Práctica 3

Anabel Gómez Ríos

```r
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.2.5
```

```r
#library(MASS)
#library(class) # Para el KNN
# Para ahorrarnos el prefijo en Auto$mpg (cada vez que queramos acceder a algo de
# Auto:
attach(Auto)
```

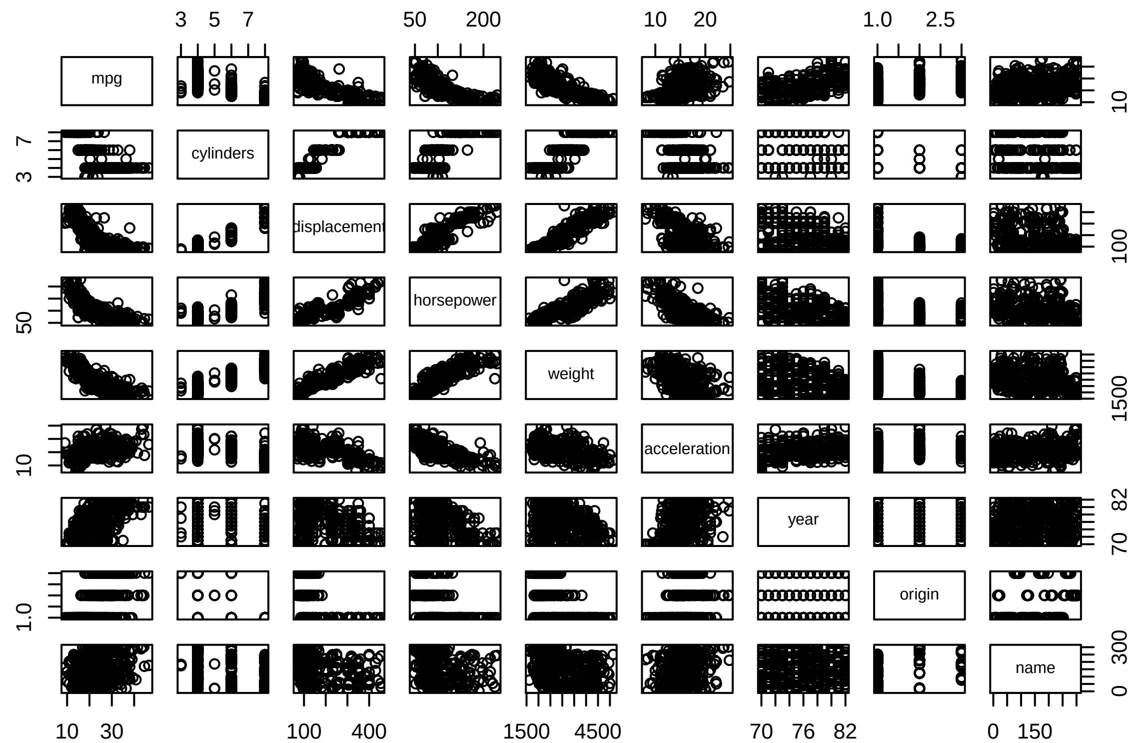
Ejercicio 1

Usar el conjunto de datos Auto que es parte del paquete ISLR. En este ejercicio desarrollaremos un modelo para predecir si un coche tiene un consumo de carburante alto o bajo usando la base de datos Auto. Se considerará alto cuando sea superior a la media de la variable mpg y bajo en caso contrario.

```r
library(ISLR)
```

a) Usar las funciones de R pairs() y boxplot() para investigar la dependencia entre mpg y las otras características. ¿Cuáles de las otras características parece más útil para predecir mpg? Justificar la respuesta.

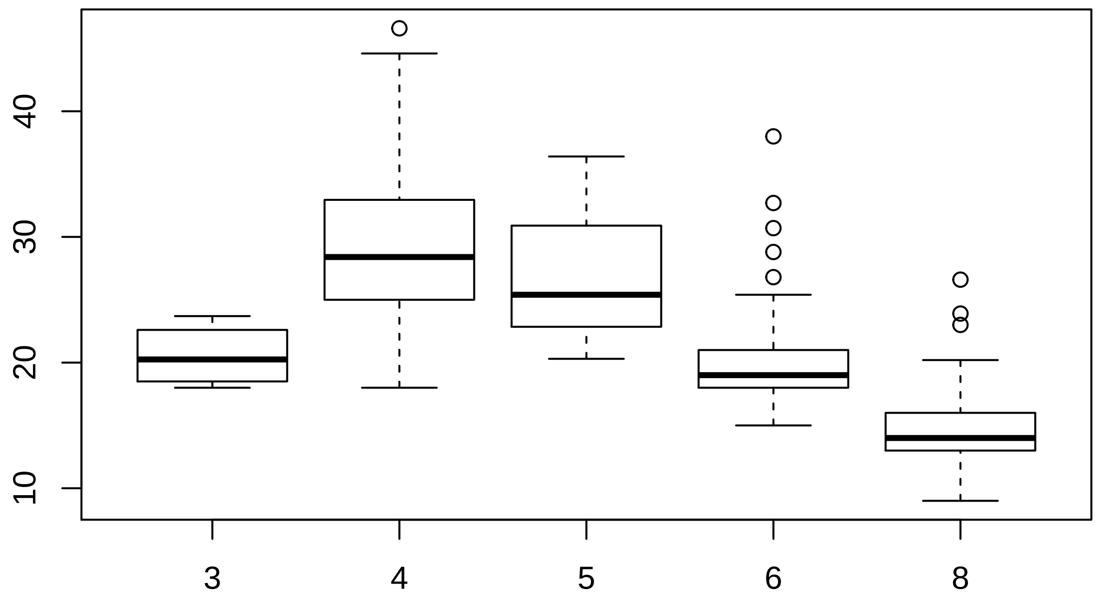
```r
pairs(Auto)
```



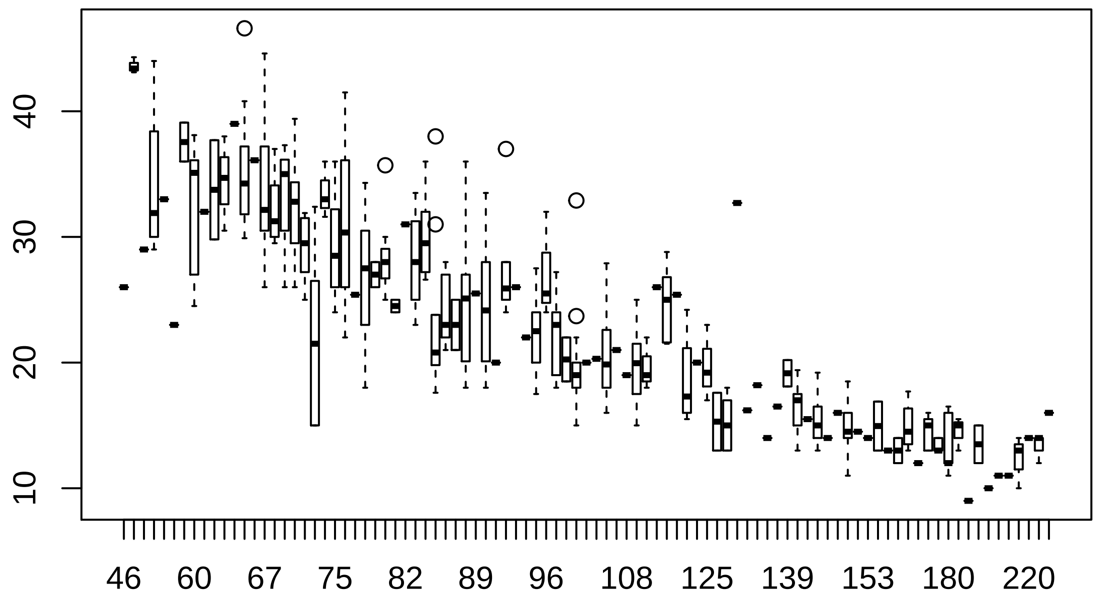
Como vemos con la función `pairs`, las características que parecen más útiles para predecir mpg (que son aquellas que tienen un patrón más o menos claro con respecto a mpg) son `displacement`, `horsepower` y `weight`.

Vamos a ver ahora las tres seleccionadas con `boxplot`:

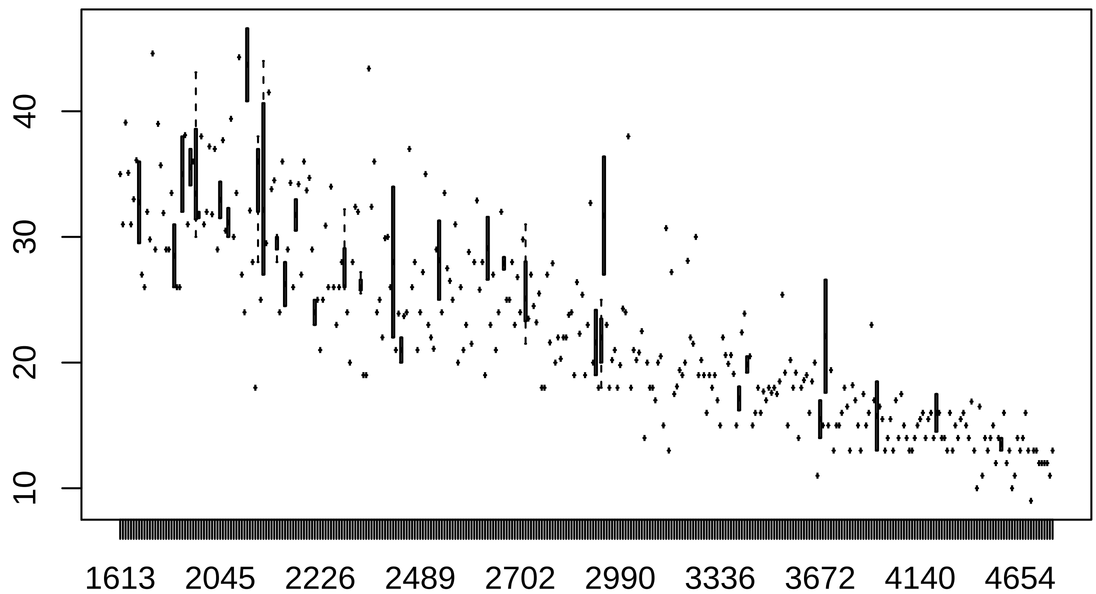
```
boxplot(Auto$mpg~Auto$cylinders)
```




```r
boxplot(Auto$mpg~Auto$horsepower)
```

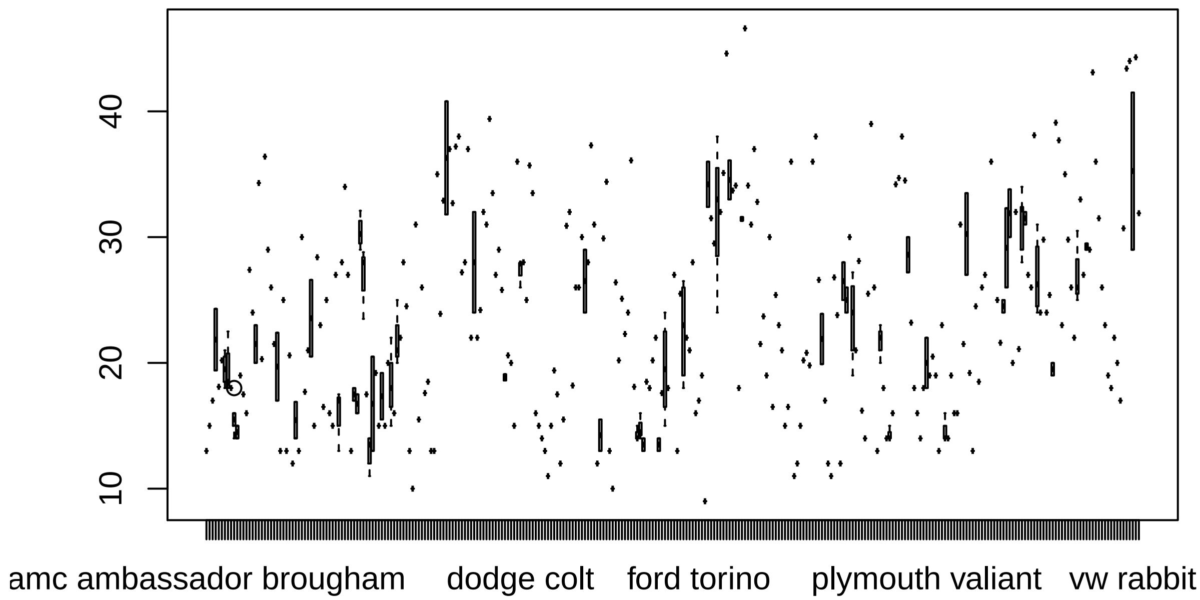



```r
boxplot(Auto$mpg~Auto$weight)
```

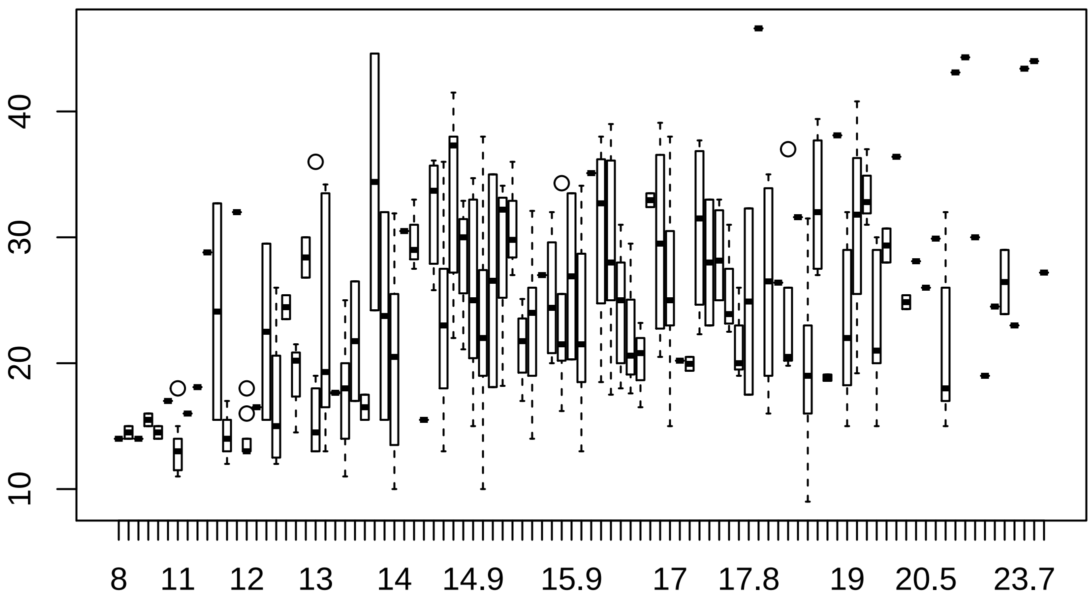


Como vemos las tres siguen más o menos una estructura clara. Veamos otras que no sean ninguna de estas tres:

```r
boxplot(Auto$mpg~Auto$name)
```




```r
boxplot(Auto$mpg~Auto$acceleration)
```



Estas por ejemplo, vemos que no tienen mucho que ver con `mpg`, puesto que las cajas, en lugar de seguir un patrón, parecen más o menos puestas de forma aleatoria.

b) Seleccionar las variables predictorias que considere más relevantes.

Lo que vamos a hacer es crear otro `data.frame` en el que vamos a eliminar el resto de variables, y vamos a dejar sólo las que hemos citado previamente.

```
# Elegimos las cinco primeras columnas de Auto, que contienen mpg, cylinders,
# displacement, horsepower y weight
AutoMod <- Auto[ ,1:5]
# Nos sobra la columna de cylinders, así que la eliminamos
AutoMod <- AutoMod[ ,-2]
```

c) Particionar el conjunto de datos en un conjunto de entrenamiento (80%) y otro de test (20%). Justificar el procedimiento usado.

Como la base de datos de Auto tiene una gran cantidad de instancias y además están ordenadas por el año de salida, lo que hacemos es elegir de forma aleatoria, con la función `sample()` el 80% de estas instancias para train, y el resto las dejaremos para test.

```
# Fijamos la semilla para el sample
set.seed(237)
train = sample(1:nrow(AutoMod), round(nrow(AutoMod)*0.8))
```

```
test = AutoMod[-train, ]
train = AutoMod[train, ]
```

d) Crear una variable binaria, mpg01, que será igual a 1 si la variable mpg contiene un valor por encima de la mediana, y -1 si mpg contiene un valor por debajo de la mediana. La mediana se puede calcular usando la función median(). (Nota: puede resultar útil usar la función data.frames() para unir en un mismo conjunto de datos la nueva variable mpg01 y las otras variables de Auto).

Lo que vamos a haces es seleccionar de los conjuntos de train y test que hemos separado previamente, las posiciones en las que la variable `mpg` está por encima de la mediana y las posiciones en las que está por debajo para después cambiar dicha variable a 1 y -1 repectivamente. No he utilizado la función `data.frame()` porque como ya tengo un data.frame train y otro test con las variables seleccionadas en b) lo voy a modificar en dichos data.frame directamente.

```
# Obtenemos las posiciones a cambiar con 1 y -1 en train
posiciones <- c(1:length(train[,1]))
pos_positivos <- posiciones[train[,1] >= median(train[,1])]
pos_negativos <- posiciones[train[,1] < median(train[,1])]
train[,1][pos_positivos] = 1
train[,1][pos_negativos] = -1

# Hacemos lo mismo para test
posiciones <- c(1:length(test[,1]))
pos_positivos <- posiciones[test[,1] >= median(test[,1])]
pos_negativos <- posiciones[test[,1] < median(test[,1])]
test[,1][pos_positivos] = 1
test[,1][pos_negativos] = -1
```

1. Ajustar un modelo de regresión logística a los datos de entrenamiento y predecir mpg01 usando las variables seleccionadas en b). ¿Cuál es el error de test del modelo? Justificar la respuesta.

Estos datos, con las variables seleccionadas y con la variable `mpg` a 1 y -1 es lo que tenemos ahora mismo en train y test, y es lo que por tanto vamos a utilizar. Para ajustar un modelo de regresión logística vamos a utilizar la función `glm()`, a la que le pasamos la variable a predecir, `train$mpg` y las variables con las que la vamos a predecir. Como hemos dicho que ya tenemos sólo las variables seleccionadas, podemos directamente poner un `.` para que tome el resto de variables presentes en los datos que le pasamos, que serán `train`.

Para calcular el error de test del modelo tenemos que predecir, con el modelo logístico que hemos obtenido, la salida que nos da para la variable `mpg` del conjunto de test. Para esto utilizo la función `predict()`, a la que hay que pasarle el modelo y los datos de test sin la variable `mpg`. Una vez tenemos las predicciones, que serán números positivos o negativos, como es un problema que hemos transformado a clasificación, nos quedamos con el signo de estas predicciones y todas aquellas que coincidan en signo con la variable `mpg` de test, estarán bien clasificadas. Contamos por tanto aquellas que no coincidan, lo dividimos por el número de instancias que tenemos en test y multiplicamos por 100 para obtener el porcentaje de error.

```
modlog1 <- glm(train$mpg~., data = train)
prediccionesRL <- predict(modlog1, test[,-1])
comp <- sign(prediccionesRL) == sign(test$mpg)
errores <- comp[comp == FALSE]
```

```r
error.regresion <- 100*(length(errores)/nrow(test))
cat("El error con regresión ha sido:", error.regresion)
```

```
## El error con regresión ha sido: 3.846154
```

El porcentaje de error, como vemos, es 3.846154

2. Ajustar un modelo K-NN a los datos de entrenamiento y predecir mpg01 usando solamente las variables seleccionadas en b). ¿Cuál es el error de test en el modelo? ¿Cuál es el valor de K que mejor ajusta los datos? Justificar la respuesta. (Usar el paquete class de R).

Para obtener el valor de K que mejor ajusta a los datos he utilizado la función `tune.knn()`, que prueba con un rango de Ks dados y devuelve aquel que tenga mejor tasa de acierto en train. Previo a esto, he normalizado los datos de test y de train (por separado, para así no infliur en los datos de test con los de train):

```r
escalado <- scale(train[,2:4])
train[,2:4] <- escalado
centro <- attr(escalado,"scaled:center")
escala <- attr(escalado, "scaled:scale")
test[,2:4] <- scale(test[,2:4], center=centro, scale=escala)
```

Vamos a obtener el mejor K entre 1 y 10. Antes de usar `tune.knn()` es importante fijar una semilla ya que cuando hay empates lo que hace es desempatar de forma aleatoria. Además es necesario pasar los datos a una matriz para que funcione.

```r
library(class)
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 3.2.5
```

```r
# Pasamos los datos con los que vamos a predecir a una matriz
x <- as.matrix(train[,-1])
# Fijamos la semilla
set.seed(237)
tune.knn(x,as.factor(train[,1]), k=1:10, tunecontrol=tune.control(sampling = "cross"))
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  9
##
## - best performance: 0.0890121
```

```r
# Utilizamos knn con el mejor k que nos ha dicho tune.knn
pred.knn <- knn(train[,-1], test[,-1], train[,1], k=9)
```

Vamos a ver ahora el error de test, para el que utilizo la función `table()` con las predicciones y la variable `mpg` real para que me devuelva la matriz de confusión. El error serán los falsos positivos y los falsos negativos entre el número de todas las instancias:

```r
confM <- table(pred.knn, test[,1])
error <- (confM[1,2] + confM[2,1])/(confM[1,1]+confM[1,2]+confM[2,1]+confM[2,2])
cat("El error en test ha sido:", error)
```

```
## El error en test ha sido: 0.05128205
```

Tenemos por tanto un error de poco más del 5% en test, lo que es muy bueno.

3. Pintar las curvas ROC (instalar paquete ROCR de R) y comparar y valorar los resultados obtenidos para ambos modelos.

En el caso del knn tenemos que obtener la probabilidad de que cada elemento en test pertenezca a la clase 1 o -1, para lo que utilizamos el parámetro `prob=T` en la función `knn()`. Sin embargo necesitamos tener la probabilidad de que todos los elementos pertenezcan a una sola clase, o a 1 o a -1, para lo que cambiamos aquellas que sean, por ejemplo, -1, y ponemos 1-la probabilidad de que pertenezcan a la clase -1, que es lo que nos devuelve `knn()` con `prob=T`, con lo que tenemos lo que necesitamos.

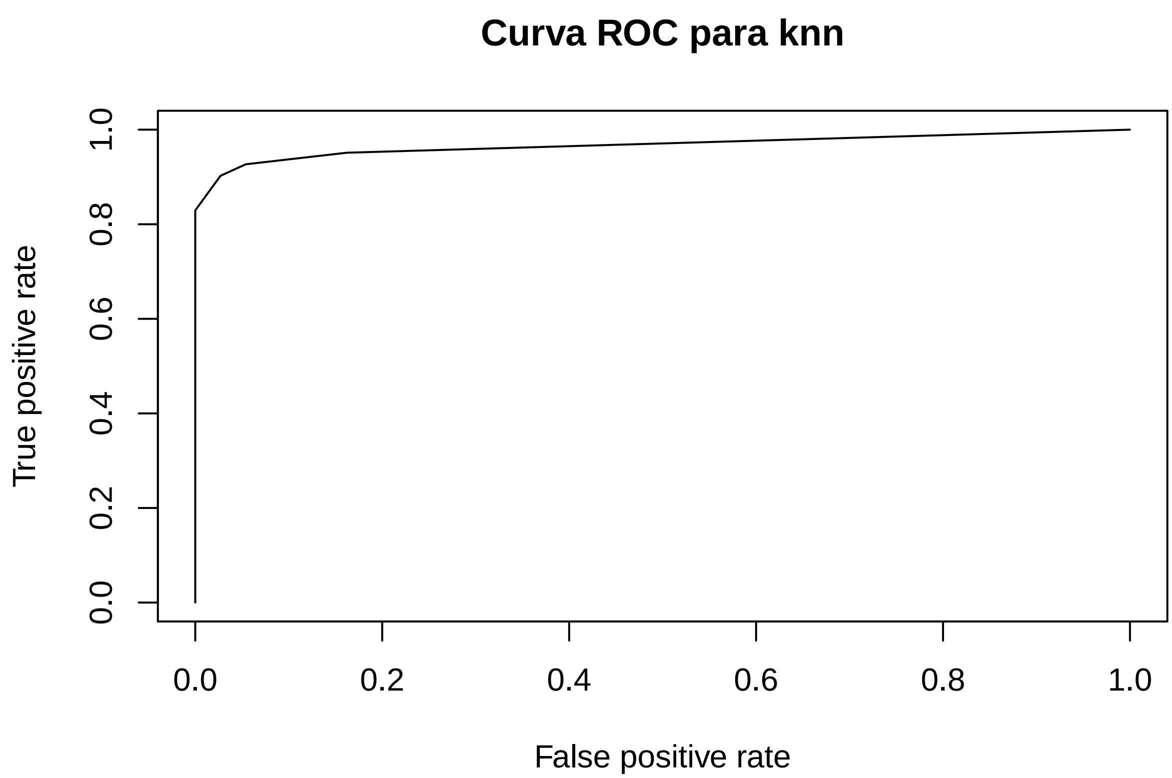
Posteriormente, y esto es común para `knn` y para regresión logística, lo que tenemos que hacer es obtener de esto un objeto de tipo `prediction` para lo que utilizamos la función del mismo nombre, pasándole por parámetros estas probabilidades, después utilzamos la función `performance()` con el objeto de tipo `prediction` y las medidas "tpr" y "fpr", que según nos dice en la ayuda de la función, son las necesarias para obtener la curva ROC. Por último, pintando el objeto performance, tenemos las curvas ROC:

```r
library(ROCR)
```

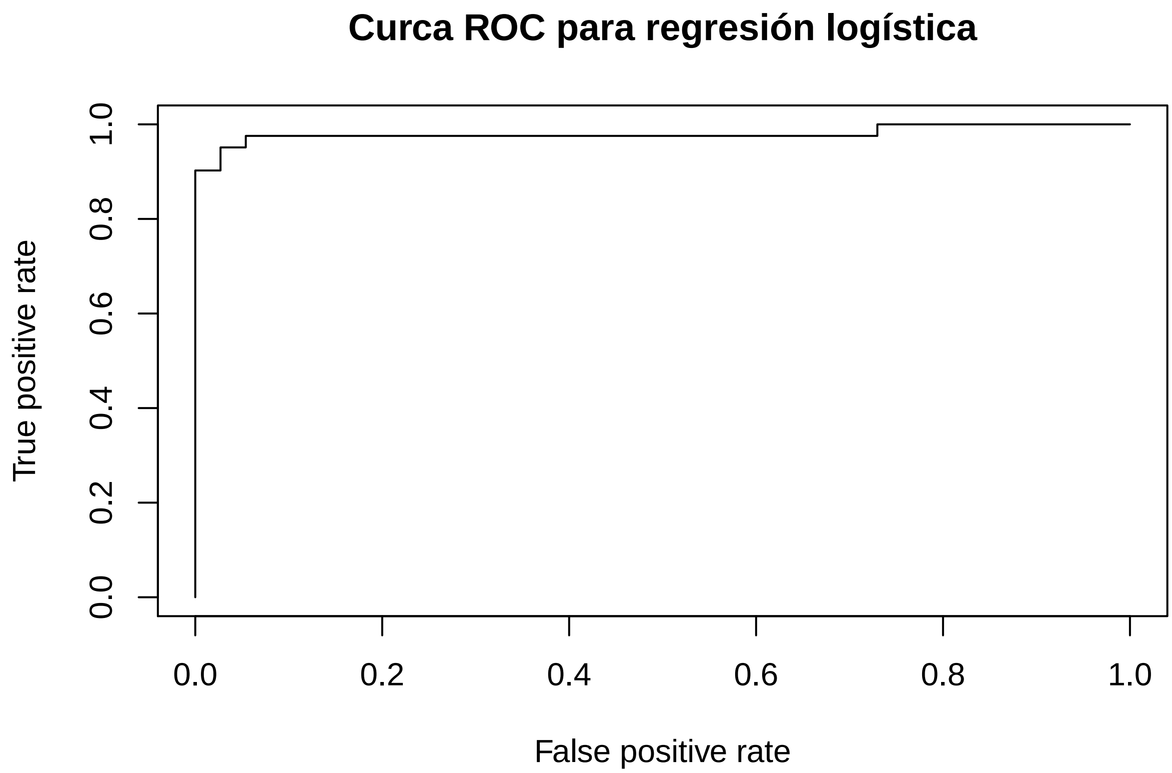
```
## Warning: package 'ROCR' was built under R version 3.2.5

## Loading required package: gplots

## Warning: package 'gplots' was built under R version 3.2.5

##
## Attaching package: 'gplots'

## The following object is masked from 'package:stats':
##
##     lowess
```

```r
knn.res <- knn(train[,-1], test[,-1], train[,1], k=5, prob=T)
prob <- attr(knn.res,"prob")
prob <- sapply(1:length(prob), function(i) {
  if(as.numeric(knn.res[i]) == 1) {
    1-prob[i]
```

```
  }
  else {
    prob[i]
  }
 })

pred <- prediction(prob, test$mpg)
perf <- performance(pred, "tpr", "fpr")
plot(perf, main="Curva ROC para knn")
```




```
pred = prediction(prediccionesRL, test$mpg)
perf = performance(pred, "tpr", "fpr")
plot(perf, main="Curca ROC para regresión logística")
```



Como vemos, ambas son muy buenas, ya que crecen hacia 1 muy rápido en el 0, aunque después knn va un poco más lento que regresión logística, por lo que esta última nos sale un poco mejor, que es algo que ya habíamos notado al calcular los errores, ya que regresión logística tiene un error de poco más del 3% y knn de poco más del 5%.

e) Bonus-1: Estimar el error de test de ambos modelos (RL, k-NN) pero usando Validación Cruzada de 5-particiones. Comparar con los resultados obtenidos en el punto anterior.

Dividimos el conjunto AutoMod, en el que ya tenemos sólo las variables seleccionadas en el apartado b), en 5 particiones de forma aleatoria haciendo uso de las funciones `sample()` y `lapply()`, de forma que obtenemos las particiones en una lista que después combinaremos con `rbind()`.

```r
set.seed(237)
x <- sample(392,392)
l <- c(0, 78, 156, 234, 313, 392)
particiones <- lapply(1:5, function(i) {
    AutoMod[x[(l[i]+1):l[i+1]], ]
})
```

Vamos a hacer un bucle de 5 iteraciones en los que iremos cambiando los conjuntos de train y test (e iremos en cada uno modificando la variable `mpg` umbralizándola a 1 y -1 según la mediana de cada conjunto de train y test que obtengamos).

```r
media_regresion <- 0
media_knn <- 0

for (i in 1:5) {
  test <- particiones[[i]]
  y <- 1:5
  y <- y[y != i]
  train <- rbind(particiones[[y[1]]], particiones[[y[2]]], particiones[[y[3]]],
                 particiones[[y[4]]])

  # Discretizamos la variable mpg en train y test
  posiciones <- c(1:length(train[,1]))
  pos_positivos <- posiciones[train[,1] >= median(train[,1])]
  pos_negativos <- posiciones[train[,1] < median(train[,1])]
  train[,1][pos_positivos] = 1
  train[,1][pos_negativos] = -1

  posiciones <- c(1:length(test[,1]))
  pos_positivos <- posiciones[test[,1] >= median(test[,1])]
  pos_negativos <- posiciones[test[,1] < median(test[,1])]
  test[,1][pos_positivos] = 1
  test[,1][pos_negativos] = -1

  # Calculamos el error con regresión logística
  modlog1 <- glm(train$mpg~., data = train)
  prediccionesRL <- predict(modlog1, test[,-1])
  comp <- sign(prediccionesRL) == sign(test$mpg)
  errores <- comp[comp == FALSE]
  error.regresion <- 100*(length(errores)/nrow(test))
  media_regresion <- media_regresion + error.regresion

  # Escalamos para knn
  escalado <- scale(train[,2:4])
  train[,2:4] <- escalado
  centro <- attr(escalado,"scaled:center")
  escala <- attr(escalado, "scaled:scale")
  test[,2:4] <- scale(test[,2:4], center=centro, scale=escala)

  # Obtemos el mejor k
  # Pasamos los datos con los que vamos a predecir a una matriz
  x <- as.matrix(train[,-1])
  # Fijamos la semilla
  set.seed(237)
  k <- tune.knn(x, as.factor(train[,1]), k=1:10, tunecontrol =
                  tune.control(sampling = "cross"))
  k <- k$best.parameters$k
  # Utilizamos knn con el mejor k que nos ha dicho tune.knn
  pred.knn <- knn(train[,-1], test[,-1], train[,1], k=k)

  # Creamos la tabla de confusión y calculamos el error
  confM <- table(pred.knn, test[,1])
  error <- (confM[1,2] + confM[2,1]) /
              (confM[1,1]+confM[1,2]+confM[2,1]+confM[2,2])
```

```r
  media_knn <- media_knn + error
}

media_regresion <- media_regresion/5
media_knn <- 100*media_knn/5

cat("El error medio por validación cruzada en regresión logística ha sido",
    media_regresion)
```

```
## El error medio por validación cruzada en regresión logística ha sido 12.53165
```

```r
cat("El error medio por validación cruzada con knn ha sido", media_knn)
```

```
## El error medio por validación cruzada con knn ha sido 10.98994
```

Vemos que los errores, aunque siguen siendo bajos, son un poco más altos que antes. Esto puede deberse a que con las particiones de los datos que teníamos en apartados anteriores hayamos tenido más suerte a la hora de predecir. Es importante decir que este error es más fiable, ya que estamos repitiendo el experimento 5 veces con conjuntos de train y test distintos, donde los datos de test no influyen en ningún momento en los de train, y estamos calculando la media de los errores, que es más fiable que hacerlo una única vez. En este caso, además, nos sale un 2% mejor el ajuste con knn que con regresión logística.

f) Bonus-2: Ajustar el mejor modelo de regresión posible considerando la variable mpg como salida y el resto como predictorias. Justificar el modelo ajustado en base al patrón de los residuos. Estimar su error de entrenamiento y test.

```r
# Borramos lo que no necesitamos
rm(AutoMod, escalado, test, train, x)
rm(centro, comp, confM, error, error.regresion, errores)
rm(escala, knn.res, modlog1, perf, pos_negativos, pos_positivos)
rm(posiciones, pred, pred.knn, prediccionesRL, prob)
rm(particiones, media_regresion, media_knn, l)
```

Ejercicio 2.

Usar la base de datos Boston (en el paquete MASS de R) para ajustar un modelo que prediga si dado un suburbio este tiene una tasa de criminalidad (crim) por encima o por debajo de la mediana. Para ello considere la variable crim como la variable salida y el resto como variables predictoras.

De nuevo dejamos un 80% de los datos para train y un 20% para test, y los extraemos de forma aleatoria.

```r
library(MASS)
set.seed(237)
# Dividimos el conjunto en train (80%) y test(20%)
train = sample(1:nrow(Boston), round(nrow(Boston)*0.8))
test = Boston[-train, ]
train = Boston[train, ]
```

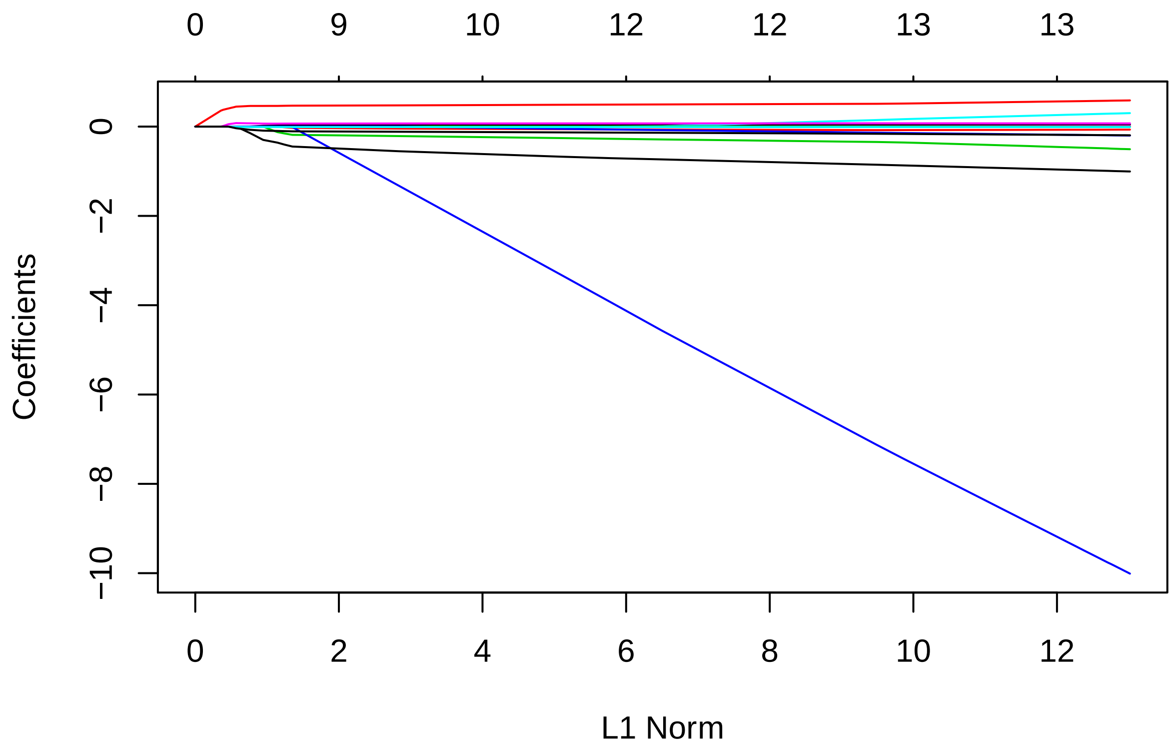
a) Encontrar el subconjunto óptimo de variables predictoras a partir de un modelo de regresión-LASSO (usar paquete glmnet de R) donde seleccionamos sólo aquellas variables con coeficiente mayor de un umbral prefijado.

Utilizando `glmnet`, le tenemos que dar un 1 al parámetro `alpha` para que sea un modelo LASSO, con el que vamos a obtener el conjunto óptimo de variables predictoras (aquellas que más correlación tengan con `crim`). A `glmnet` le tenemos que pasar los datos como matrices para que funcione, para lo que utilizo la función `as.matrix()`.

```r
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 3.2.5

## Loading required package: Matrix

## Loading required package: foreach

## Warning: package 'foreach' was built under R version 3.2.5

## Loaded glmnet 2.0-5
```

```r
# Volvemos a fijar la semilla
set.seed(237)
modelo.lasso <- glmnet(as.matrix(train[,-1]), as.matrix(train[,1]), alpha=1)
plot(modelo.lasso)
```



Como vemos, muchos de los coeficientes están cerca de cero, o son exactamente cero, lo que quiere decir que las variables correspondientes no influirán en la variable `crim`. Vamos a elegir un λ apropiado usando cross-validation, en concreto, el cross-validation de `cv.glmnet()`.

```
crossv <- cv.glmnet(as.matrix(train[,-1]), as.matrix(train[,1]), alpha=1)
lambda <- crossv$lambda.min
coeficientes <- predict(modelo.lasso, type="coefficients", s=lambda)[1:14,]
print(coeficientes)
```

```
##   (Intercept)            zn         indus          chas           nox
## 12.081390554  0.035634736 -0.065867295 -0.288254662 -4.571273619
##            rm           age           dis           rad           tax
##  0.007674630  0.001512413 -0.735387080  0.495939249  0.000000000
##       ptratio         black         lstat          medv
## -0.079012478 -0.011083919  0.070805286 -0.138844223
```

Como vemos hay muchos de estos coeficientes muy cercanos a 0. Nos vamos a quedar con aquellos que estén (en valor absoluto) por encima de un umbral 0.2, que significará que tienen cierta correlación con la variable `crim`, aunque podríamos elegir uno un poco más bajo si quisiéramos considerar más variables (aunque serían menos relevantes).

```
coeficientes <- coeficientes[abs(coeficientes)>0.2]
print(coeficientes)
```

```
## (Intercept)        chas         nox         dis         rad
##  12.0813906  -0.2882547  -4.5712736  -0.7353871   0.4959392
```

Nos quedamos sólo entonces con las variables `chas`, `nox`, `dis` y `rad`.

b) Ajustar un modelo de regresión regularizada con "weight-decay" (ridge-regression) y las variables seleccionadas. Estimar el error residual del modelo y discutir si el comportamiento de los residuos muestran algún indicio de "underfitting".

Para *weight-decay* tenemos que usar también `glmnet` pero con el parámetro $\alpha = 0$.
Nos quedamos primero con las variables seleccionadas por el método anterior, que introducimos en un data.frame nuevo para facilitar el uso posterior de la función `as.matrix()`, necesaria para `glmnet()`, y vamos a utilizar el mejor λ que nos ha salido del apartado anterior por cross-validation:

```
BostonMod.train <- data.frame(train$crim, train$chas, train$nox, train$dis,
                              train$rad)
BostonMod.test <- data.frame(test$crim, test$chas, test$nox, test$dis, test$rad)
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                       as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                        newx=as.matrix(BostonMod.test[,-1]))
```

Calculamos ahora el error residual, que será la raíz cuadrada positiva de los cuadrados de las diferencias entre nuestro valor y el predicho por el modelo.

```
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error residual ha sido:", error.res)
```

```
## El error residual ha sido: 4.948363
```

Para ver ahora si estamos o no ajustando poco el modelo (underfitting) vamos a probar distintos valores de λ, que es el parámetro que maneja la cantidad de regularización que le damos al modelo. Vamos a coger dos valores por debajo del λ que tenemos en este momento (0.0450578) y dos por encima y comprobar qué sucede:

```
lambda <- 0.08
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                       as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                        newx=as.matrix(BostonMod.test[,-1]))
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error para lambda =", lambda, "ha sido", error.res)
```

```
## El error para lambda = 0.08 ha sido 4.94845
```

```
lambda <- 1.5
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                       as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                        newx=as.matrix(BostonMod.test[,-1]))
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error para lambda =", lambda, "ha sido", error.res)
```

```
## El error para lambda = 1.5 ha sido 4.967192
```

```r
lambda <- 0.02
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                       as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                        newx=as.matrix(BostonMod.test[,-1]))
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error para lambda =", lambda, "ha sido", error.res)
```

```
## El error para lambda = 0.02 ha sido 4.950263
```

```r
lambda <- 0.0015
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                       as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                        newx=as.matrix(BostonMod.test[,-1]))
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error para lambda =", lambda, "ha sido", error.res)
```

```
## El error para lambda = 0.0015 ha sido 4.9509
```

Cuanto mayor es el λ mayor la cantidad de regularización, pero el error sube tanto si bajamos como si aumentamos λ, con lo que no estamos ajustando poco el modelo, ya que si disminuimos la cantidad de reguralización ajustamos más los datos de train pero el error residual es mayor, con lo que sobreajustamos el modelo a los datos, y por tanto estamos en el mejor valor de λ posible.

c) Definir una nueva variable con valores -1 y 1 usando el valor de la mediana de la variable crim como umbral. Ajustar un modelo SVM que prediga la nueva variable definida. (Usar el paquete e1071 de R). Describir con detalle cada uno de los pasos dados en el aprendizaje del modelo SVM. Comience ajustando un modelo lineal y argumente si considera necesario usar algún núcleo. Valorar los resultados del uso de distintos núcleos.

Comenzamos discretizando por separado los datos de train y test (lo hacemos por separado para que los datos de test no influyan en los de train).

```r
library(e1071)
```

```r
posiciones <- c(1:length(train[,1]))
pos_positivos <- posiciones[train[,1] >= median(train[,1])]
pos_negativos <- posiciones[train[,1] < median(train[,1])]
train[,1][pos_positivos] = 1
train[,1][pos_negativos] = -1

posiciones <- c(1:length(test[,1]))
pos_positivos <- posiciones[test[,1] >= median(test[,1])]
pos_negativos <- posiciones[test[,1] < median(test[,1])]
test[,1][pos_positivos] = 1
test[,1][pos_negativos] = -1
```

Vamos a hacer ahora un SVM con núcleo lineal, a ver qué error obtenemos. Para ello vamos a utilizar la función `svm()`, a la que le pasamos, como viene siendo habitual, el conjunto de train sin la variable a predecir, la variable a predecir y el tipo de núcelo que queremos utilizar, en este caso lineal. Seguidamente volvemos a utilizar la función `predict()` con el modelo `svm` obtenido y los datos de train sin la variable que intentamos predecir. Como lo hemos convertido en un problema de clasificación, lo que tenemos que hacer es lo que hemos hecho en el primer ejercicio, comparar aquellas instancias en las que el signo de la varaible predicha no coincida con la nuestra discretizada y contar aquellas en las que esto ocurra, de forma que el error será este número por 100 entre el número total de instancias.

```
# Volvemos a fijar la semilla
set.seed(237)
modelo.svm <- svm(train[,1]~., train[,-1], kernel="linear")
predicciones <- predict(modelo.svm, test[,-1])
comp <- sign(predicciones) == sign(test$crim)
errores <- comp[comp == FALSE]
error <- 100*(length(errores)/nrow(test))
cat("El error con SVM lineal ha sido:", error)
```

```
## El error con SVM lineal ha sido: 19.80198
```

El error obtenido está muy por encima del error obtenido en el apartado anterior utilizando regresión regularizada, lo que nos hace pensar que un svm lineal no ajusta lo suficientemente bien los datos. Vamos a probar con los otros tres tipos de núcleos a ver los resultados en errores que obtenemos y comprobar si podemos mejorar el error ajustando mejor los datos con otros tipos de núcleos. La predicción y el cálculo de errores se hace de forma análoga al caso lineal.

```
modelo.svm <- svm(train[,1]~., train[,-1], kernel="polynomial")
predicciones <- predict(modelo.svm, test[,-1])
comp <- sign(predicciones) == sign(test$crim)
errores <- comp[comp == FALSE]
error <- 100*(length(errores)/nrow(test))
cat("El error con SVM polinomial ha sido:", error)
```

```
## El error con SVM polinomial ha sido: 17.82178
```

```
modelo.svm <- svm(train[,1]~., train[,-1], kernel="radial")
predicciones <- predict(modelo.svm, test[,-1])
comp <- sign(predicciones) == sign(test$crim)
errores <- comp[comp == FALSE]
error <- 100*(length(errores)/nrow(test))
cat("El error con SVM radial ha sido:", error)
```

```
## El error con SVM radial ha sido: 15.84158
```

```
modelo.svm <- svm(train[,1]~., train[,-1], kernel="sigmoid")
predicciones <- predict(modelo.svm, test[,-1])
comp <- sign(predicciones) == sign(test$crim)
errores <- comp[comp == FALSE]
error <- 100*(length(errores)/nrow(test))
cat("El error con SVM sigmoidal ha sido:", error)
```

```
## El error con SVM sigmoidal ha sido: 42.57426
```

Efectivamente, tanto el modelo polinomial como el radial mejoran el error, aunque no mucho y sigue siendo alto. El mejor de ellos es el radial por dos puntos de diferencia con el polinomial.

Bonus-3: Estimar el error de entrenamiento y test por validación cruzada de 5 particiones.

Vamos a hacer las 5 particiones como en el bonus del apartado 1, haciendo una permutación del número de instancias en la base de datos Boston y partiendo esa permutación en 5 trozos, que serán los índices de las instancias en cada partición:

```r
set.seed(237)
x <- sample(405,405)
l <- c(0, 81, 162, 243, 324, 405)
particiones <- lapply(1:5, function(i) {
    Boston[x[(l[i]+1):l[i+1]], ]
})
```

Vamos a utilizar el svm que mejor error nos ha dado, que ha sido el de núcleo radial con las variables que ya habíamos seleccionado en apartados anteriores.

```r
error_test <- 0
error_train <- 0
for (i in 1:5) {
  test <- particiones[[i]]
  y <- 1:5
  y <- y[y != i]
  train <- rbind(particiones[[y[1]]], particiones[[y[2]]], particiones[[y[3]]],
                 particiones[[y[4]]])

  # Discretizamos la variable crim en train y test
  posiciones <- c(1:length(train[,1]))
  pos_positivos <- posiciones[train[,1] >= median(train[,1])]
  pos_negativos <- posiciones[train[,1] < median(train[,1])]
  train[,1][pos_positivos] = 1
  train[,1][pos_negativos] = -1

  posiciones <- c(1:length(test[,1]))
  pos_positivos <- posiciones[test[,1] >= median(test[,1])]
  pos_negativos <- posiciones[test[,1] < median(test[,1])]
  test[,1][pos_positivos] = 1
  test[,1][pos_negativos] = -1

  modelo.svm <- svm(train[,1]~., train[,-1], kernel="radial")
  predicciones.test <- predict(modelo.svm, test)
  predicciones.train <- predict(modelo.svm, train)
  comp.test <- sign(predicciones.test) == sign(test[,1])
  errores.test <- comp.test[comp.test == FALSE]
  comp.train <- sign(predicciones.train) == sign(train[,1])
  errores.train <- comp.train[comp.train == FALSE]
  error.test <- 100*(length(errores.test)/nrow(test))
  error.train <- 100*(length(errores.train)/nrow(train))
  error_test <- error_test + error.test
  error_train <- error_train + error.train
}

error_train <- error_train/5
```

```r
error_test <- error_test/5
print("El error con svm de núcelo radial para train con validación cruzada ha sido")
```

```
## [1] "El error con svm de núcelo radial para train con validación cruzada ha sido"
```

```r
print(error_train)
```

```
## [1] 11.85185
```

```r
print("El error con svm de núcelo radial para test con validación cruzada ha sido")
```

```
## [1] "El error con svm de núcelo radial para test con validación cruzada ha sido"
```

```r
print(error_test)
```

```
## [1] 12.59259
```

```r
# Borramos lo que no necesitamos
rm(BostonMod.test, BostonMod.train, test, train)
rm(coeficientes, comp, crossv, error, error.res, errores)
rm(lambda, modelo.lasso, modelo.ridge, predicciones, modelo.svm)
rm(pos_negativos, pos_positivos, posiciones, error.test, error.train)
rm(comp.test, comp.train, crim, crim.test, error_test, error_train)
```

```
## Warning in rm(comp.test, comp.train, crim, crim.test, error_test,
## error_train): objeto 'crim' no encontrado
```

```
## Warning in rm(comp.test, comp.train, crim, crim.test, error_test,
## error_train): objeto 'crim.test' no encontrado
```

```r
rm(l, particiones, predicciones.test, predicciones.train, y)
rm(errores.test, errores.train)
```

Ejercicio 3.

Usar el conjunto de datos Boston y las librerías randomForest y gbm de R.

```r
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 3.2.5
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```r
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 3.2.5
```

```
## Loading required package: survival
```

```
## Loading required package: lattice
```

```
## Loading required package: splines
```

```
## Loading required package: parallel
```

```
## Loaded gbm 2.1.1
```

```r
library(MASS)
```

a) Dividir la base de datos en dos conjuntos de entrenamiento (80%) y test (20%).

```r
set.seed(237)
train = sample(1:nrow(Boston), round(nrow(Boston)*0.8))
test = Boston[-train, ]
train = Boston[train, ]
```

b) Usando la variable medv como salida y el resto como predictoras, ajustar un modelo de regresión usando bagging. Explicar cada uno de los parámetros usados. Calcular el error del test.

Veamos cuántas variables hay, ya que va a ser necesario especificar el total para hacer bagging:

```r
summary(train)
```

```
##       crim                zn            indus            chas
##  Min.   : 0.00632   Min.   :  0.0   Min.   : 0.46   Min.   :0.00000
##  1st Qu.: 0.08199   1st Qu.:  0.0   1st Qu.: 5.13   1st Qu.:0.00000
##  Median : 0.26838   Median :  0.0   Median : 8.56   Median :0.00000
##  Mean   : 3.70216   Mean   : 11.6   Mean   :11.06   Mean   :0.07407
##  3rd Qu.: 3.67822   3rd Qu.: 17.5   3rd Qu.:18.10   3rd Qu.:0.00000
##  Max.   :88.97620   Max.   :100.0   Max.   :27.74   Max.   :1.00000
##       nox               rm             age              dis
##  Min.   :0.385   Min.   :3.863   Min.   :  2.90   Min.   : 1.130
##  1st Qu.:0.449   1st Qu.:5.875   1st Qu.: 45.70   1st Qu.: 2.079
##  Median :0.538   Median :6.211   Median : 76.50   Median : 3.272
##  Mean   :0.555   Mean   :6.296   Mean   : 68.76   Mean   : 3.807
##  3rd Qu.:0.631   3rd Qu.:6.649   3rd Qu.: 94.40   3rd Qu.: 5.118
##  Max.   :0.871   Max.   :8.780   Max.   :100.00   Max.   :12.127
##       rad              tax           ptratio          black
```

```
##  Min.   : 1.00   Min.   :187.0   Min.   :12.60   Min.   :  0.32
##  1st Qu.: 4.00   1st Qu.:277.0   1st Qu.:17.40   1st Qu.:374.71
##  Median : 5.00   Median :330.0   Median :19.00   Median :391.43
##  Mean   : 9.64   Mean   :406.7   Mean   :18.46   Mean   :355.82
##  3rd Qu.:24.00   3rd Qu.:666.0   3rd Qu.:20.20   3rd Qu.:395.99
##  Max.   :24.00   Max.   :711.0   Max.   :22.00   Max.   :396.90
##      lstat            medv
##  Min.   : 1.73   Min.   : 5.00
##  1st Qu.: 6.93   1st Qu.:16.70
##  Median :11.50   Median :21.20
##  Mean   :12.88   Mean   :22.68
##  3rd Qu.:17.21   3rd Qu.:25.00
##  Max.   :37.97   Max.   :50.00
```

Como vemos son 14 variables en total, si quitamos la que vamos a predecir, son en total 13. Vamos a empezar con un número de árboles 100 y ver el error que tiene, e iremos subiendo hasta que el error se quede más o menos estable, ya que sabemos que bagging no sobreajusta los datos.

Los parámetros utilizados son la variable a estimar, el resto del conjunto de train y el número de variables del conjunto de entrenamiento. Después hacemos una predicción con la función `predict()` y calculamos el error cuadrado medio.

```
bagging10 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 100)
pred10 <- predict(bagging10, newdata = test)
cat("El error con 10 árboles es:", mean((pred10-test$medv)^2))
```

```
## El error con 10 árboles es: 6.96409
```

```
bagging300 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 300)
pred300 <- predict(bagging300, newdata = test)
cat("El error con 300 árboles es:", mean((pred300-test$medv)^2))
```

```
## El error con 300 árboles es: 7.064586
```

```
bagging500 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 500)
pred500 <- predict(bagging500, newdata = test)
cat("El error con 500 árboles es:", mean((pred500-test$medv)^2))
```

```
## El error con 500 árboles es: 6.952598
```

```
bagging600 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 600)
pred600 <- predict(bagging600, newdata = test)
cat("El error con 600 árboles es:", mean((pred600-test$medv)^2))
```

```
## El error con 600 árboles es: 6.921371
```

```
bagging700 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 700)
pred700 <- predict(bagging700, newdata = test)
cat("El error con 700 árboles es:", mean((pred700-test$medv)^2))
```

```
## El error con 700 árboles es: 6.811936
```

```r
bagging650 <- randomForest(train$medv~., data = train, mtry = 13, ntree = 650)
pred650 <- predict(bagging650, newdata = test)
cat("El error con 650 árboles es:", mean((pred650-test$medv)^2))
```

```
## El error con 650 árboles es: 6.843684
```

Como vemos ya en valores cercanos a 650 el error se estabiliza y no varía mucho. El mejor de ellos es en 650, donde da 6.84%.

c) Ajustar un modelo de regresión usando Random Forest. Obtener una estimación del número de árboles necesario. Justificar el resto de parámetros usados en el ajuste. Calcular el error de test y compararlo con el obtenido con bagging.

Como es un modelo de regresión, vamos a usar una cantidad de variables igual a $p/3$, donde p es el número de variables, que en este caso es 13, luego utilizamos el redondeo por arriba de $13/3 = 4,33$, que es 5. Vamos a hacer lo mismo que antes para elegir el número de árboles, ya que random forest tampoco sobreajusta los datos.

```r
# Fijamos la semilla de nuevo
set.seed(237)
randomF10 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 10)
pred10 <- predict(randomF10, newdata=test)
cat("El error con 10 árboles es:", mean((pred10-test$medv)^2))
```

```
## El error con 10 árboles es: 7.815575
```

```r
randomF30 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 30)
pred30 <- predict(randomF30, newdata=test)
cat("El error con 30 árboles es:", mean((pred30-test$medv)^2))
```

```
## El error con 30 árboles es: 7.487246
```

```r
randomF50 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 50)
pred50 <- predict(randomF50, newdata=test)
cat("El error con 50 árboles es:", mean((pred50-test$medv)^2))
```

```
## El error con 50 árboles es: 7.254418
```

```r
randomF100 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 100)
pred100 <- predict(randomF100, newdata=test)
cat("El error con 100 árboles es:", mean((pred100-test$medv)^2))
```

```
## El error con 100 árboles es: 6.912036
```

```r
randomF200 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 200)
pred200 <- predict(randomF200, newdata=test)
cat("El error con 200 árboles es:", mean((pred200-test$medv)^2))
```

```
## El error con 200 árboles es: 6.869719
```

```r
randomF400 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 400)
pred400 <- predict(randomF400, newdata=test)
cat("El error con 400 árboles es:", mean((pred400-test$medv)^2))
```

```
## El error con 400 árboles es: 6.573803
```

```r
randomF600 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 600)
pred600 <- predict(randomF600, newdata=test)
cat("El error con 600 árboles es:", mean((pred600-test$medv)^2))
```

```
## El error con 600 árboles es: 6.923628
```

```r
randomF500 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 500)
pred500 <- predict(randomF500, newdata=test)
cat("El error con 100 árboles es:", mean((pred500-test$medv)^2))
```

```
## El error con 100 árboles es: 6.896802
```

```r
randomF450 <- randomForest(train$medv~., data = train, mtry = 5, ntree = 450)
pred450 <- predict(randomF450, newdata=test)
cat("El error con 450 árboles es:", mean((pred450-test$medv)^2))
```

```
## El error con 450 árboles es: 6.828355
```

Podemos ver que el mejor número de árboles es 400, que da un error de 6.87%, y es por tanto con el que nos vamos a quedar.

d) Ajustar un modelo de regresión usando Boosting (usar gbm con distribution = 'gaussian'). Calcular el error de test y compararlo con el obtenido con bagging y Random Forest.

```r
library(gbm)
set.seed(237)
```

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 400, shrinkage = 0.001)
pred <- predict(boosting, newdata = test, n.trees=400)
print("Error para 400 árboles y shrinkage = 0.001")
```

```
## [1] "Error para 400 árboles y shrinkage = 0.001"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 37.85734
```

El error sale muy alto, vamos a intentar cambiar los párametros (subir tanto el número de árboles como shrinkage, que se recomienda que esté entre 0.1 y 0.001) a ver si conseguimos bajarlo:

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 400, shrinkage = 0.01)
pred <- predict(boosting, newdata = test, n.trees=400)
print("Error para 400 árboles y shrinkage = 0.01")
```

```
## [1] "Error para 400 árboles y shrinkage = 0.01"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 13.58642
```

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 500, shrinkage = 0.01)
pred <- predict(boosting, newdata = test, n.trees=500)
print("Error para 500 árboles y shrinkage = 0.01")
```

```
## [1] "Error para 500 árboles y shrinkage = 0.01"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 13.16294
```

Vemos que subir ambos parámetros ayuda. Continuamos:

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 500, shrinkage = 0.1)
pred <- predict(boosting, newdata = test, n.trees=500)
print("Error para 500 árboles y shrinkage = 0.1")
```

```
## [1] "Error para 500 árboles y shrinkage = 0.1"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 11.19801
```

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 600, shrinkage = 0.1)
pred <- predict(boosting, newdata = test, n.trees=600)
print("Error para 500 árboles y shrinkage = 0.1")
```

```
## [1] "Error para 500 árboles y shrinkage = 0.1"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 10.82985
```

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 600, shrinkage = 0.2)
pred <- predict(boosting, newdata = test, n.trees=600)
print("Error para 500 árboles y shrinkage = 0.1")
```

```
## [1] "Error para 500 árboles y shrinkage = 0.1"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 11.52691
```

Aquí vemos que subir a 0.2 shrinkage no ayuda, con lo que vamos a dejarlo en 0.1 y continuar con el número de árboles.

```r
boosting <- gbm(medv~., data = train, distribution = "gaussian", n.trees = 700, shrinkage = 0.1)
pred <- predict(boosting, newdata = test, n.trees=700)
print("Error para 500 árboles y shrinkage = 0.1")
```

```
## [1] "Error para 500 árboles y shrinkage = 0.1"
```

```r
print(mean((pred - test$medv)^2))
```

```
## [1] 11.47879
```

Subir el número de árboles ya tampoco ayuda, con lo que lo dejamos en 600 árboles. El error por tanto con boosting es 10.82%, 4 puntos peor que con bagging y randomForest.

Ejercicio 4.

Usar el conjunto de datos OJ que es parte del paquete ISLR.

a) Crear un conjunto de entrenamiento conteniendo una muestra aleatoria de 800 observaciones, y un conjunto de test conteniendo el resto de las observaciones. Ajustar un árbol a los datos de entrenamiento, con Purchase como la variable respuesta y las otras variables como predictoras (paquete tree de R).

Hacemos lo mismo que en el resto de ejercicios para dividir en train y test: seleccionamos una muestra aleatoria de, en este caso, 800 instancias para train y dejamos el resto para test.

```r
# Fijamos de nuevo la semilla
set.seed(237)

library(ISLR)
train.idx = sample(1:nrow(OJ), 800)
test = OJ[-train.idx, ]
train = OJ[train.idx, ]

# Usamos la librería tree
library(tree)
```

```
## Warning: package 'tree' was built under R version 3.2.5
```

Para saber si usar clasificación o regresión vamos a ver de qué tipo es la variable `Purchase` con la función `summary()`

```r
summary(OJ)
```

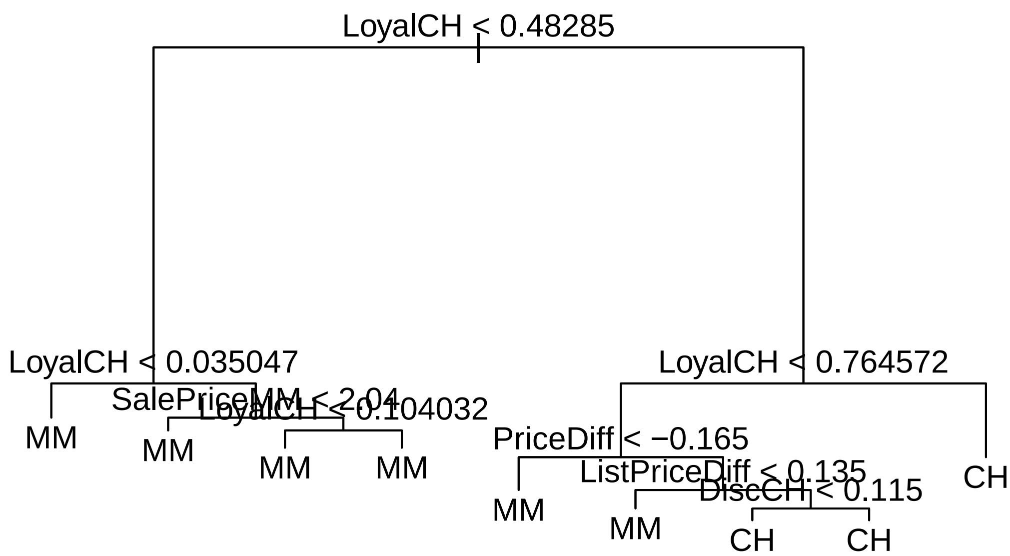
```
##  Purchase WeekofPurchase     StoreID        PriceCH         PriceMM     
##  CH:653   Min.   :227.0   Min.   :1.00   Min.   :1.690   Min.   :1.690  
##  MM:417   1st Qu.:240.0   1st Qu.:2.00   1st Qu.:1.790   1st Qu.:1.990  
##           Median :257.0   Median :3.00   Median :1.860   Median :2.090  
##           Mean   :254.4   Mean   :3.96   Mean   :1.867   Mean   :2.085  
##           3rd Qu.:268.0   3rd Qu.:7.00   3rd Qu.:1.990   3rd Qu.:2.180  
##           Max.   :278.0   Max.   :7.00   Max.   :2.090   Max.   :2.290  
##      DiscCH            DiscMM         SpecialCH        SpecialMM     
##  Min.   :0.00000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000  
##  1st Qu.:0.00000   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.0000  
##  Median :0.00000   Median :0.0000   Median :0.0000   Median :0.0000  
##  Mean   :0.05186   Mean   :0.1234   Mean   :0.1477   Mean   :0.1617  
##  3rd Qu.:0.00000   3rd Qu.:0.2300   3rd Qu.:0.0000   3rd Qu.:0.0000  
##  Max.   :0.50000   Max.   :0.8000   Max.   :1.0000   Max.   :1.0000  
##     LoyalCH          SalePriceMM     SalePriceCH      PriceDiff      
##  Min.   :0.000011   Min.   :1.190   Min.   :1.390   Min.   :-0.6700  
##  1st Qu.:0.325257   1st Qu.:1.690   1st Qu.:1.750   1st Qu.: 0.0000  
##  Median :0.600000   Median :2.090   Median :1.860   Median : 0.2300  
##  Mean   :0.565782   Mean   :1.962   Mean   :1.816   Mean   : 0.1465  
##  3rd Qu.:0.850873   3rd Qu.:2.130   3rd Qu.:1.890   3rd Qu.: 0.3200  
##  Max.   :0.999947   Max.   :2.290   Max.   :2.090   Max.   : 0.6400  
##  Store7      PctDiscMM        PctDiscCH       ListPriceDiff  
##  No :714   Min.   :0.0000   Min.   :0.00000   Min.   :0.000  
##  Yes:356   1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:0.140  
##            Median :0.0000   Median :0.00000   Median :0.240  
##            Mean   :0.0593   Mean   :0.02731   Mean   :0.218  
##            3rd Qu.:0.1127   3rd Qu.:0.00000   3rd Qu.:0.300  
##            Max.   :0.4020   Max.   :0.25269   Max.   :0.440  
##      STORE      
##  Min.   :0.000  
##  1st Qu.:0.000  
##  Median :2.000  
##  Mean   :1.631  
##  3rd Qu.:3.000  
##  Max.   :4.000
```

Como vemos, `Purchase` toma sólo dos valores: `CH` y `MM`, con lo que vamos a utilizar clasificación. Utilizamos la función `tree()`, pasándole como parámetro la variable a predecir y el resto del conjunto de train, y nos devuelve un árbol de clasificación para dicho conjunto de entrenamiento:

```r
tree.oj <- tree(train$Purchase~., train[,-1])
```

Vamos a ver el resultado de este árbol:

```r
plot(tree.oj)
text(tree.oj, pretty = 0)
```



Dicho árbol tiene 9 nodos terminales y 17 nodos en total, contando el nodo raíz. Sin embargo vemos que podría haber menos, ya que en la parte de la izquierda, por ejemplo, todos los nodos van a `MM`.

c) Usar la función summary() para generar un resumen estadístico acerca del árbol y describir los resultados obtenidos: tasa de error de training, número de nodos del árbol, etc.

```r
summary(tree.oj)
```

```
##
## Classification tree:
## tree(formula = train$Purchase ~ ., data = train[, -1])
## Variables actually used in tree construction:
## [1] "LoyalCH"       "SalePriceMM"   "PriceDiff"     "ListPriceDiff"
## [5] "DiscCH"
## Number of terminal nodes:  9
## Residual mean deviance:  0.7303 = 577.6 / 791
## Misclassification error rate: 0.1638 = 131 / 800
```

Como vemos, el número de nodos hoja son 9. El número total de nodos lo podemos ver en el árbol anterior y es 17. Las variables que se han utilizado en los nodos internos (aquellos que no son hoja) nos lo dice también la función `summary()` y son `LoyalCH`, `SalePriceMM`, `PriceDiff`, `ListPriceDiff` y `DiscCH`.
También nos dice el error en training: 0.1638 (16%) y la desviación residual hasta la media, que en este caso

es la desviación normal a la media entre 800 (total de instancias en train) - 9 (número de nodos hoja) = 791, lo que da 0.7303. Como es lógico, a menor desviación, mejor ajusta el árbol a los datos de train.

d) Predecir la respuesta de los datos de test, y generar e interpretar la matriz de confusión de los datos de test. ¿Cuál es la tasa de error del test? ¿Cuál es la precisión del test?

Para esto podemos usar de nuevo la función `predict()` pasándole el árbol que hemos entrenado con train y por otro lado los datos de test. Le ponemos el argumento `type=class` porque estamos con un árbol de clasificación y así obligamos a utilziar la predicción con la variable `Purchase`. Para calcular el error de test y su precisión vamos a utilizar la función `table()`, que nos devuelve la matriz de confusión.

```r
tree.predict <- predict(tree.oj, test[,-1], type="class")
table(tree.predict, test[,1])
```

```
##
## tree.predict  CH  MM
##           CH 125  11
##           MM  39  95
```

Esta matriz devuelve en la primera fila aquellas instancias que eran `CH` y efectivamente el árbol ha predicho `CH` y las instancias que lo eran pero el árbol predijo `MM`, que son 11. En la segunda fila devuelve las instancias en las que era `MM` y el árbol predijo `CH` (39) y aquellas que eran `CH` y acertó, 95.
El error en test es (11+39)/(125+11+39+95) = 0.1851852
La precisión es (125+95)/(125+11+39+95) = 0.8148148
Es decir, hay un error en test del 18.5% y una precisión del 81.4%.

e) Aplicar la función cv.tree() al conjunto de training y determinar el tamaño óptimo del árbol. ¿Qué hace cv.tree?

La misión de `cv.tree()` es utilizar cross-validation para obtener el mejor nivel de complejidad para el árbol que se obtiene. Este "mejor nivel" se puede obtener en base a diferentes criterios. Por ejemplo, si usamos `cv.tree()` con los parámetros por defecto, nos devolverá aquel que tenga menor desvianza. Si queremos que nos devuelva aquel que tenga menor error en la validación cruzada tenemos que usar el parámetro `FUN = prune.misclass`.

```r
set.seed(237)
cv.oj <- cv.tree(tree.oj, FUN = prune.misclass)
print(cv.oj)
```

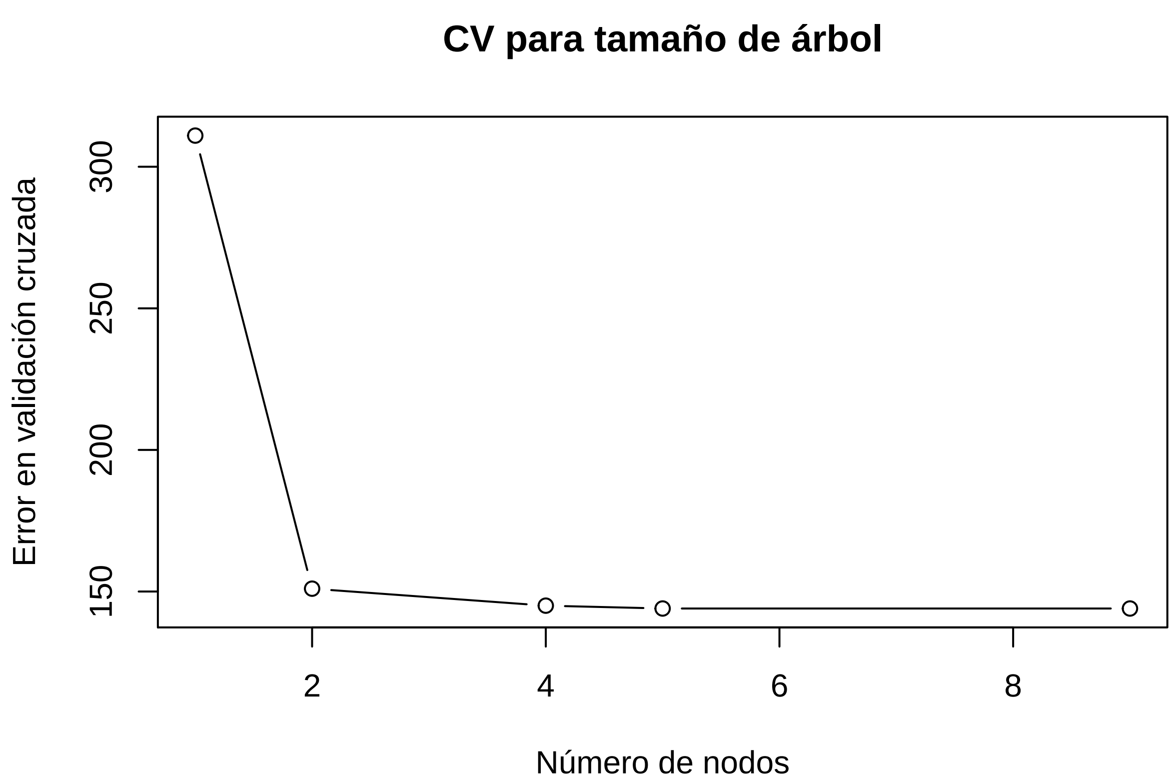
```
## $size
## [1] 9 5 4 2 1
##
## $dev
## [1] 144 144 145 151 311
##
## $k
## [1]  -Inf   0.0   2.0   7.5 163.0
##
## $method
```

```
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Como podemos ver, los árboles con 9 y 5 nodos terminales son los que tienen el menor error, 144. El árbol que habíamos ajustado en el apartado c) tenía justo 9 nodos terminales, pero habíamos visto que tenía nodos que le sobraban, con lo que el árbol óptimo es el que tiene 5 nodos terminales, que es el que tiene menos error y menos nodos simultáneamente.

Bonus-4. Generar un gráfico con el tamaño del árbol en el eje x (número de nodos) y la tasa de error de validación cruzada en el eje y. ¿Qué tamaño de árbol corresponde a la tasa más pequeña de error de clasificación por validación cruzada?

```
plot(cv.oj$size, cv.oj$dev, type = "b", main = "CV para tamaño de árbol",
     ylab = "Error en validación cruzada", xlab = "Número de nodos")
```



El tamaño de árbol que corresponde a la tasa más pequeña de error de clasificación es, como hemos comentado antes, es 5 (y todos a partir de ahí).

```
# Borramos lo que no necesitamos
rm(test, train, cv.oj, train.idx, tree.oj, tree.predict)
```