Trabajo 3

Alejandro García Montoro

2 de junio de 2016

Ejercicio 1

Carguemos primero los datos necesarios para realizar el ejercicio y echémosle un primer vistazo a la base de datos:

```
# Cargamos la librería necesaria para usar la base de datos Auto
# Para usarla, hay que instalar con la orden
# install.packages('ISLR')
library(ISLR)

# Usamos Auto por defecto, evitando así poner el prefijo Auto$
# siempre que queramos acceder a una característica de esa base de datos
attach(Auto)
```

Si ejecutamos las órdenes siguientes

```
class(Auto)
dim(Auto)
colnames(Auto)
```

podemos obtener información de la forma que tiene nuestra base de datos. Vemos así que tiene forma de data.frame, con 392 filas y 9 columnas, cuyos nombres son los siguientes: mpg, cylinders, displacement, horsepower, weight, acceleration, year, origin, name.

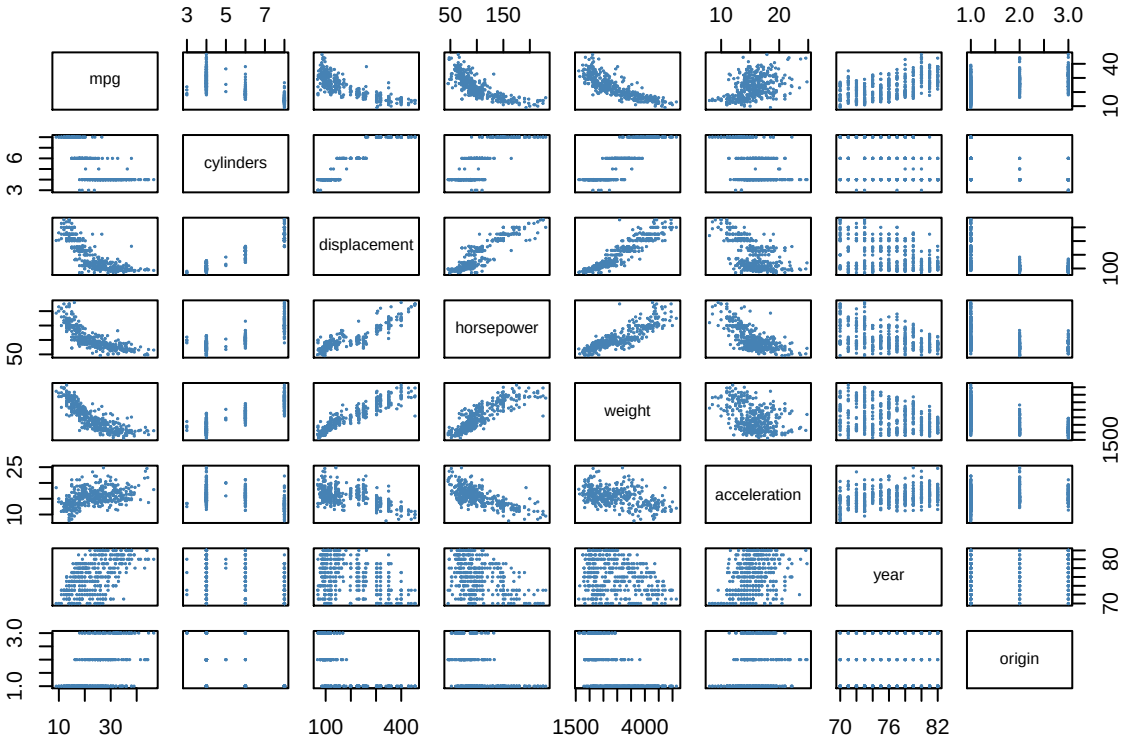
Podemos eliminar la última columna, que es el nombre del vehículo y la única variable no numérica, para poder trabajar con comodidad más adelante:

```
# Eliminamos la última columna
Auto <- Auto[,seq(ncol(Auto)-1)]
```

Apartado a

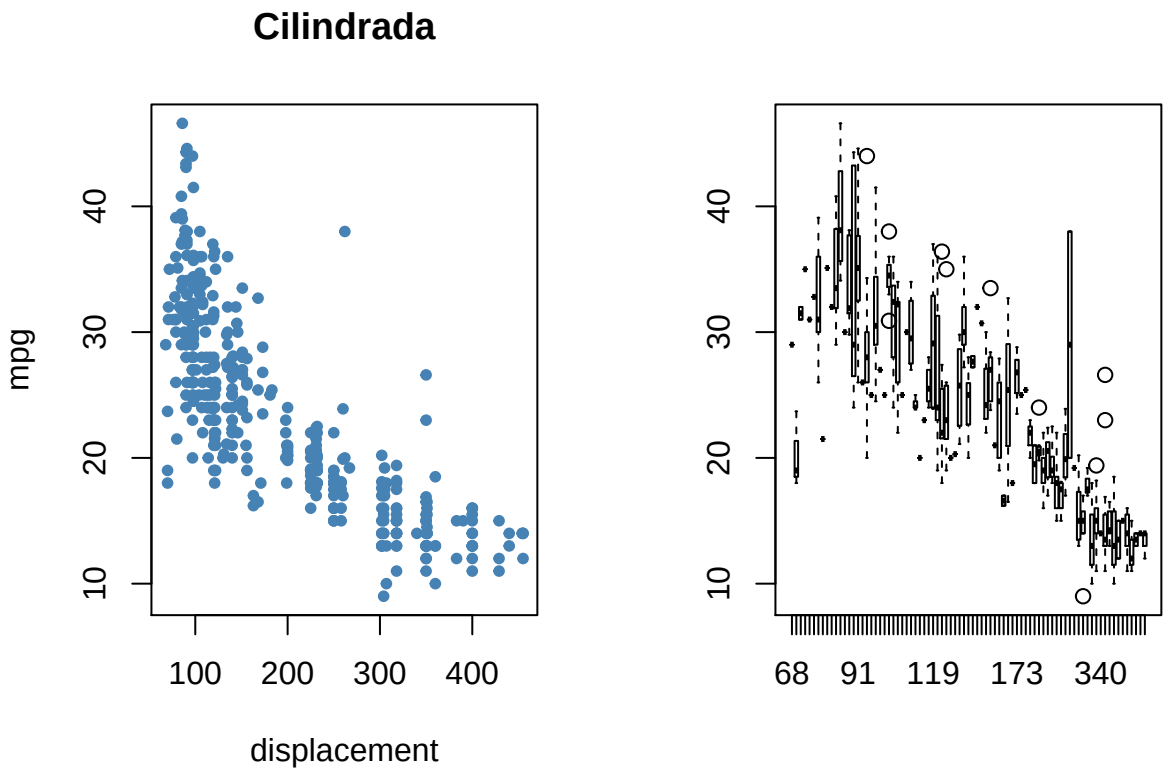
Para visualizar las dependencias entre `mpg` y las otras características podemos usar las funciones `pairs()` y `boxplot()`:

```
# Visualizamos la relación entre todos los pares de variables
pairs(Auto, pch=20, cex=0.2, col="steelblue")
```

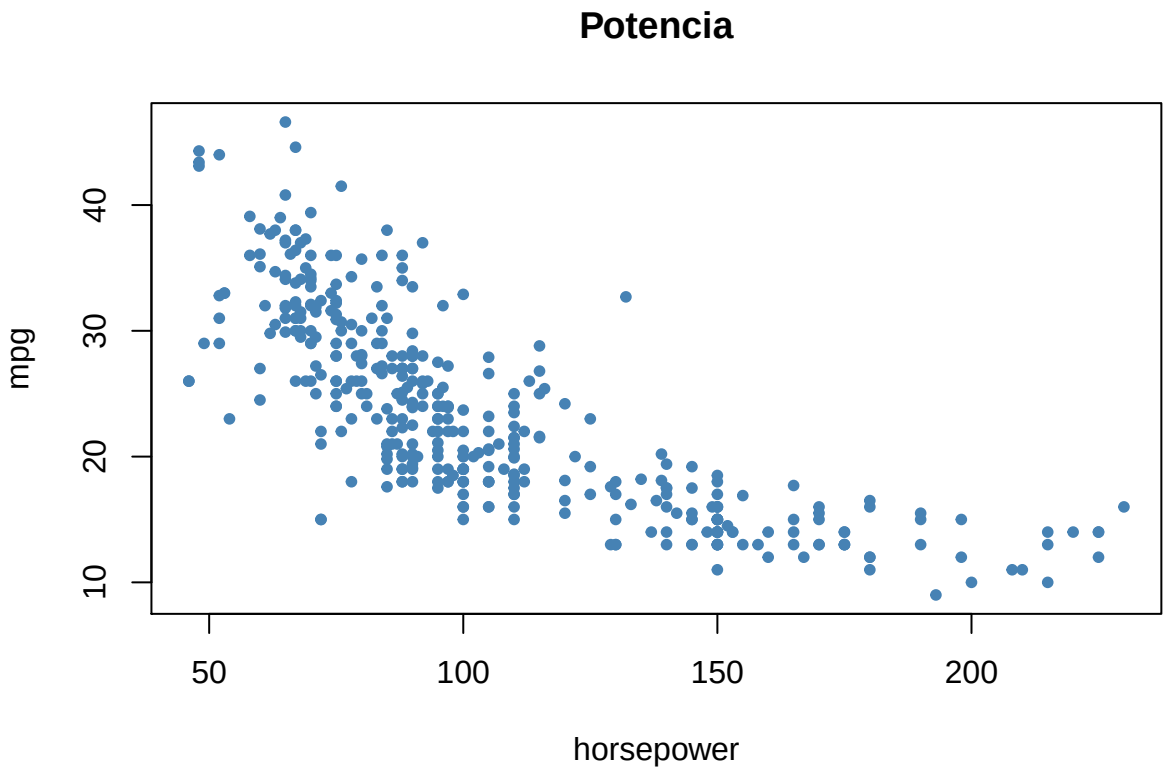


Podemos visualizar más de cerca las variables que parecen más relevantes para predecir la variable `mpg` y visualizar un gráfico de cajas. Seleccionamos estas porque en la gráfica se ve correlación; mientras que en las demás hay nubes de puntos sin forma o columnas/filas con las que no vamos a poder predecir nada.

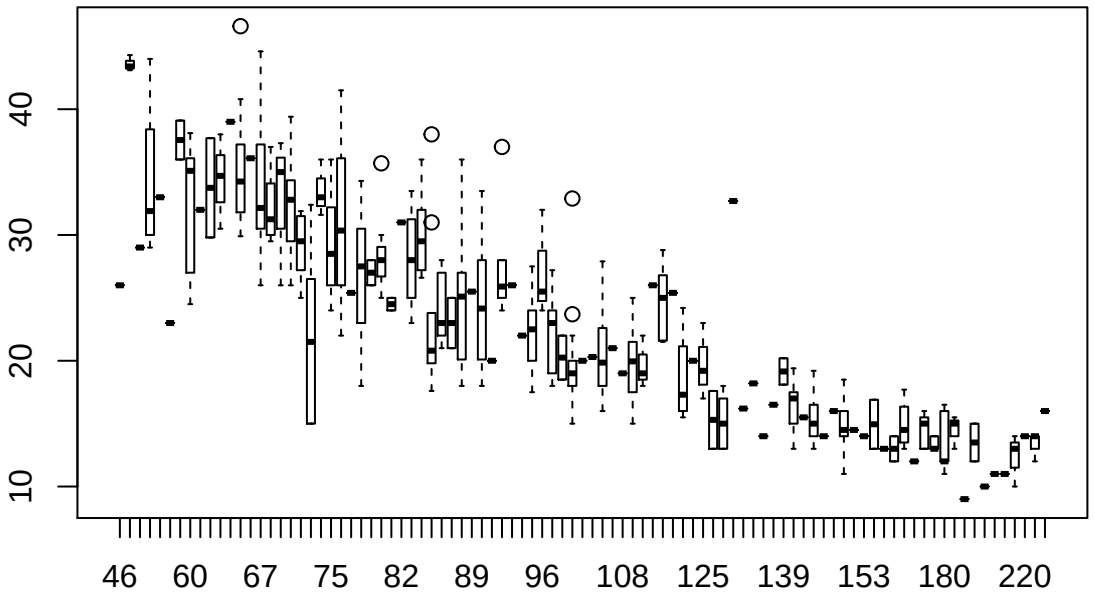
```
# Plot para mpg-displacement
par(mfrow=c(1,2))
plot(displacement, mpg, pch=20, col="steelblue",
     main="Cilindrada")
boxplot(mpg~displacement)
```



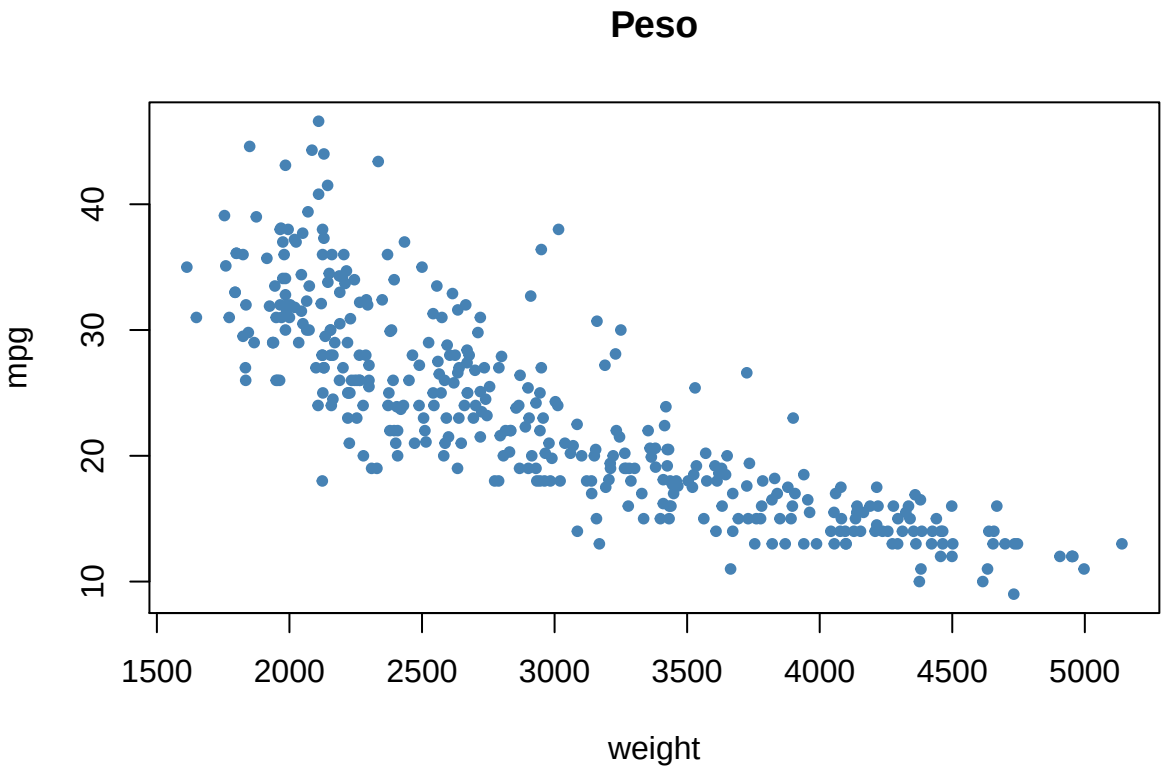

```
# Plot para mpg-horsepower
plot(horsepower, mpg, pch=20, col="steelblue",
     main="Potencia")
```



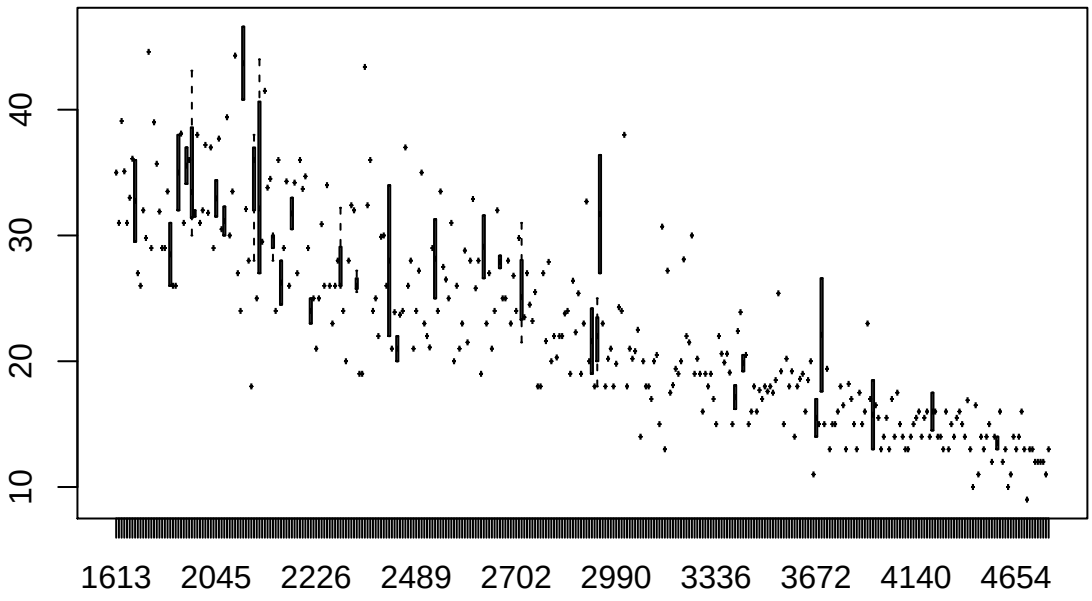
```r
boxplot(mpg~horsepower)
```




```r
# Plot para mpg-weight
plot(weight, mpg, pch=20, col="steelblue",
     main="Peso")
```




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

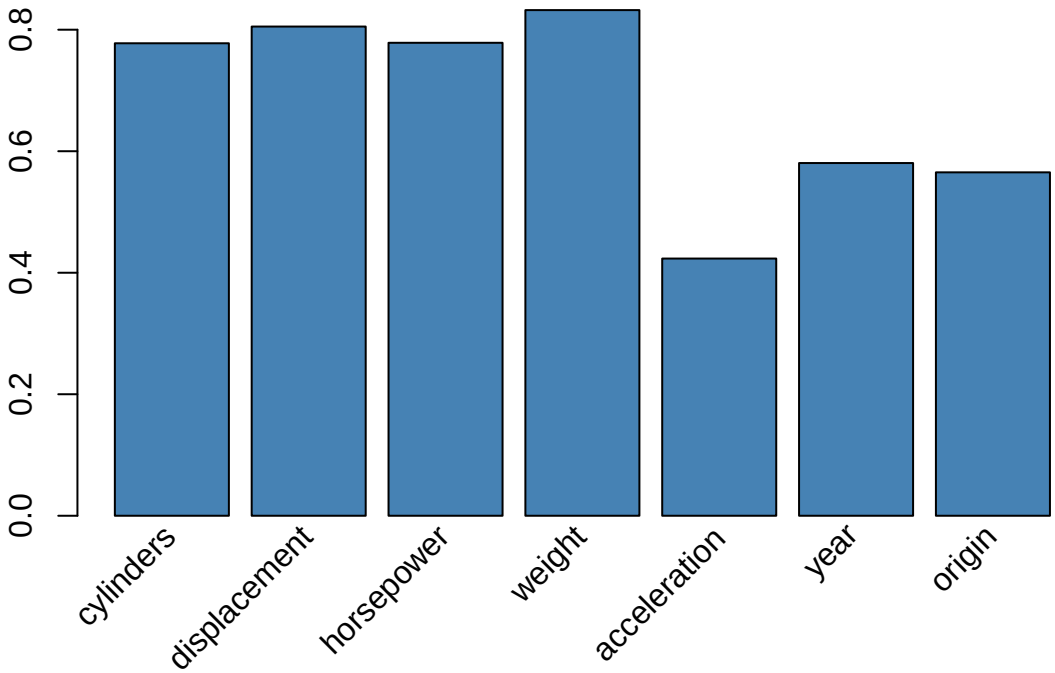



```r
par(mfrow=c(1,1))
```

Apartado b

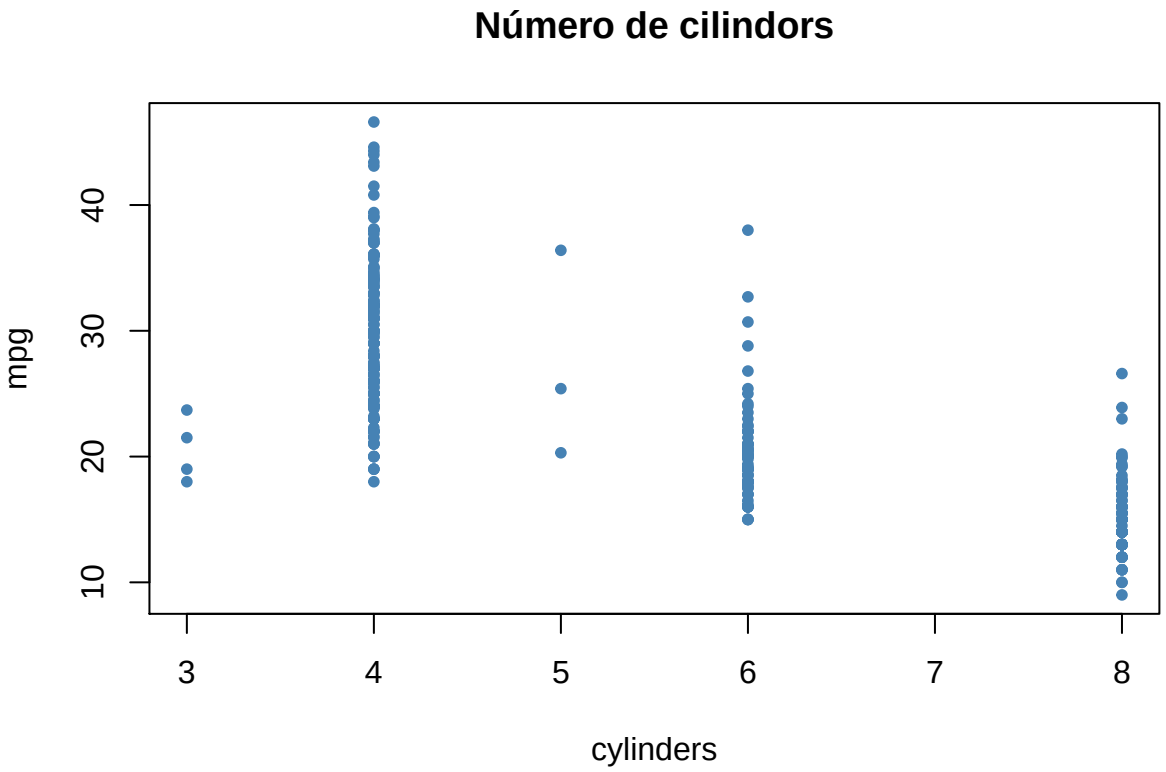
Podemos estudiar de forma numérica la correlación entre `mpg` y las demás variables:

```r
# Tomamos los valores absolutos de la correlación entre mpg y todas las demás variables,
# sin incluirse a sí misma
corr <- abs(cor(Auto))["mpg",-1]

# Visualizamos el grado de correlación en un gráfico de barras.
# Creamos el gráfico.
bp <- barplot(corr, axes = FALSE, axisnames = FALSE, col = "steelblue",
              main="Correlación entre mpg y las demás variables")

# Añadimos el texto, girado 45 grados.
text(bp, par("usr")[3]-0.02, labels = colnames(Auto[-1]),
     srt = 45, adj = 1, xpd = TRUE)

# Dibujamos los ejes.
axis(2)
```



Como vemos son `cylinders`, `displacement`, `horsepower` y `weight` las variables que más correlación presentan con `mpg`. Sin embargo, vamos a estudiar las tres últimas; la correlación entre la primera y `mpg` es un dato arbitrario que tiene más que ver con la forma de la variable que con la relación entre ambas. Esto se ve claro si ampliamos el gráfico `mpg-cylinders`:

```
# Plot para mpg-cylinders
plot(cylinders, mpg, pch=20, col="steelblue",
     main="Número de cilindors")
```



Seleccionamos las variables escogidas:

```
selec <- c("displacement", "horsepower", "weight")
```

Apartado c

Vamos a crear dos vectores que usaremos para indexar las muestras de entrenamiento y de test. Tomamos primero una muestra de índices cuyo tamaño será del 80% del número de filas de la base de datos y que constituirá el conjunto de índices de la muestra de entrenamiento; la diferencia entre el conjunto de índices total y el de la muestra de entrenamiento será el conjunto de índices de la muestra de test.

```
# Vector de índices para la muestra de entrenamiento (80%)
trainIdx  <- sample(nrow(Auto), size=0.8*nrow(Auto))

# Vector de índices para la muestra de test
testIdx   <- setdiff(1:nrow(Auto), trainIdx)
```

Apartado d

Creamos la variable booleana `mpg01`. En vez de usar los valores simbólicos 1 y -1, usamos unos mucho más expresivos: `True` y `False`. Una vez añadida la variable, partimos los datos en las muestras de entrenamiento y test con los índices calculados anteriormente:

```
# Creamos una nueva variable booleana, mpg01, en función de la mediana,
# y la añadimos a la base de datos
mpg01 <- ifelse(mpg > median(mpg), T, F)
Auto <- data.frame(mpg01, Auto)

# Obtenemos las muestras de entrenamiento y de test
```

```
Auto.train <- Auto[trainIdx,]
Auto.test  <- Auto[testIdx,]
```

Regresión lineal

Para hacer la regresión logística vamos a usar la función `glm`, que devuelve un modelo lineal `Y~X`, donde `Y` es la variable a predecir y `X` el conjunto de variables predictoras. Tenemos que pasarle el argumento `family=binomial` para que haga regresión logística en vez de un tipo general de modelo lineal.

Este modelo lo ajustaremos con la muestra de entrenamiento:

```
# Ajustamos el modelo con los datos de entrenamiento
mod.lin = glm(mpg01~displacement+horsepower+weight, data=Auto.train, family = binomial)
```

Para ver la efectividad del modelo, intentamos predecir la variable `mpg01` con la función `predict` sobre la muestra de test:

```
# Usamos el modelo lineal ajustado para predecir con la muestra de test
mod.lin.pred <- predict(mod.lin, Auto.test, type = "response")
```

Esto nos devuelve en la variable `mod.lin.pred` las probabilidades predecidas para cada punto de la muestra de test. Por tanto, vamos a asignar el valor `True` a aquellos puntos donde la probabilidad sea mayor al 50% y `False` a los demás:

```
# Si la predicción tiene probabilidad mayor que el 50%,
# asignamos el valor Verdad; en otro caso, asignamos el valor Falso
mod.lin.mpg01 <- ifelse(mod.lin.pred > 0.5, T, F)
```

Para calcular el error ya basta, tan sólo, con contar el porcentaje de puntos mal clasificados en la muestra de test:

```
# Calculamos el error como el porcentaje de muestras mal clasificadas
mod.lin.error <- mean(Auto.test$mpg01 != mod.lin.mpg01)
```

De aquí obtenemos que el error producido por este modelo es del 10.1265823%. Podemos ver exactamente cuántos falsos positivos y falsos negativos tenemos; la siguiente tabla, cuyas columnas son los valores predecidos y cuyas filas los reales, resume la efectividad del método:

```
tab <- table(Real=Auto.test$mpg01, Predecido=mod.lin.mpg01)
kable(tab, caption="Regresión lineal")
```

Table 1: Regresión lineal

	FALSE	TRUE
FALSE	31	6
TRUE	2	40

Vecino más cercano

Para ajustar el modelo de los k vecinos más cercanos necesitamos, primero, normalizar las variables predictoras. Para ello usamos la función `scale()`, que devuelve el conjunto de variables introducido de manera que todas ellas tengan media 0 y varianza 1. Omitimos en este escalado la variable que queremos predecir, `mpg01`, ya que su valor no influirá en las distancias entre los datos, que se mide con las variables predictoras.

El escalado, además lo vamos a hacer de la siguiente manera:

- Escalamos la muestras de entrenamiento.
- Tomamos los valores de medias y varianzas del escalado anterior.
- Escalamos la muestra de test con esos valores.

Esto lo hacemos para que los datos de test no influyan en el proceso de aprendizaje.

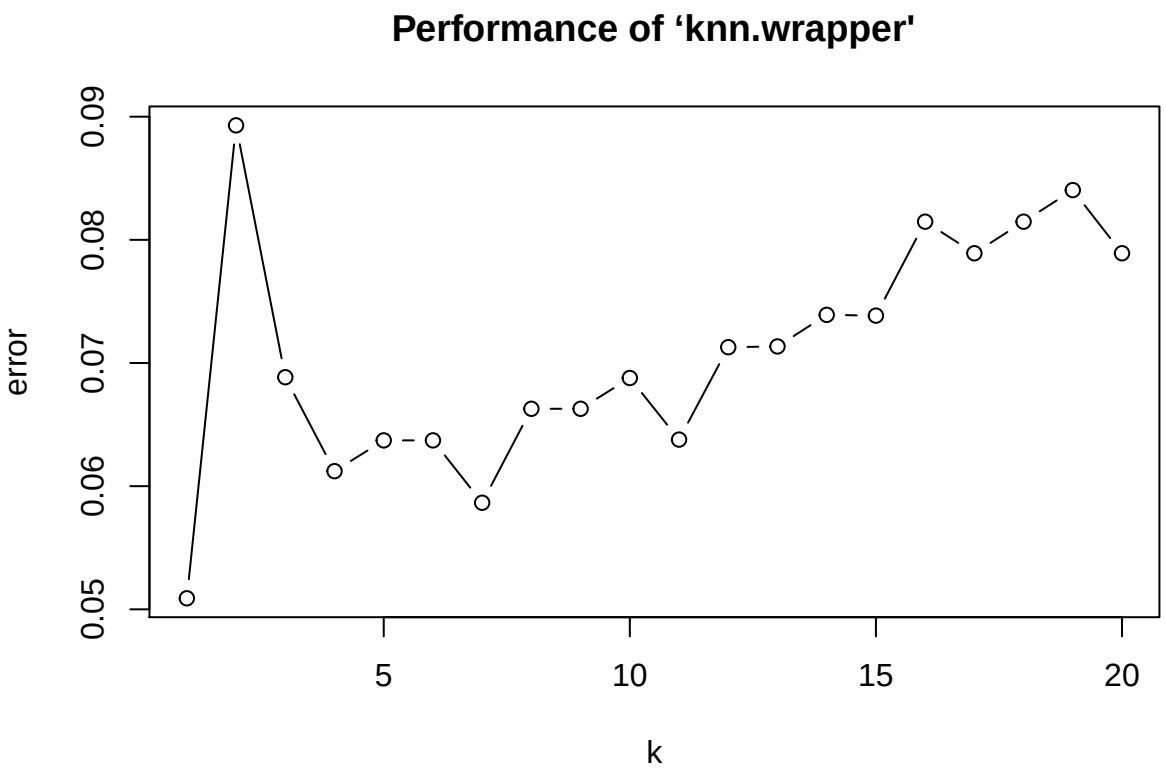
```
# Escalado de la muestra de entrenamiento sin la variable mpg01
norm.train <- scale(Auto.train[,-1])

# Escalado de la muesrta de test sin mpg01 y en base al escalado anterior
norm.test <- scale(Auto.test[,-1],
                   center = attributes(norm.train)$"scaled:center",
                   scale = attributes(norm.train)$"scaled:scale")

# Creamos una variable con ambas muestras y la reordenamos según el nombre (número) de las filas
norm.full <- rbind(norm.train, norm.test)
norm.full <- norm.full[order(as.numeric(row.names(norm.full))),]
```

La forma más sencilla de ajustar el modelo y predecir con él es llamar a la función `knn` con los atributos de las muestras de entrenamiento y test y las clases:

```
# Cargamos la librería class, que contiene los modelos del knn
library(class)

# Predecimos de la forma más directa posible y con el típico valor de k=3
knn.pred <- knn(norm.train, norm.test, as.factor(Auto.train$mpg01), k=3, prob=T)

# Calculamos el error
knn.error <- mean(Auto.test$mpg01 != knn.pred)
```

Tenemos así un error de 8.8607595%, que mejora al anterior de regresión lineal. Sin embargo, podemos intentar mejorarlo, ajustando el parámetro k a su óptimo. Esta regularización la podemos hacer con la función `tune.knn` y con el método de validación cruzada:

```
# Necesitamos la librería e1071, que podemos instalar con la orden
# install.packages("e1071")
library(e1071)

# Para la validación cruzada usamos el conjunto completo de datos
# y el conjunto completo de clases
knn.cross <- tune.knn(x = norm.full, y = as.factor(Auto$mpg01), k = 1:20,
                      tunecontrol = tune.control(sampling = "cross"), cross=10)

# Podemos estudiar visualmente la búsqueda del mejor k:
plot(knn.cross)
```




```
# Nos quedamos con el mejor parámetro
knn.k <- knn.cross$best.parameters$k
```

Ajustamos de nuevo el modelo con el valor de $k = 1$ obtenido del anterior estudio:

```
# Predecimos con el k calculado
knn.pred <- knn(norm.train, norm.test, as.factor(Auto.train$mpg01), k = knn.k, prob = T)

# Calculamos de nuevo el error
knn.k.error <- mean(Auto.test$mpg01 != knn.pred)

# Vemos una tabla que resume los resultados
tab <- table(Real=Auto.test$mpg01, Predecido=knn.pred)
kable(tab, caption="kNN")
```

Table 2: kNN

	FALSE	TRUE
FALSE	35	2
TRUE	4	38

consiguiendo ahora un 7.5949367% de error, 1.2658228 puntos mejor que el anterior.

Curvas ROC

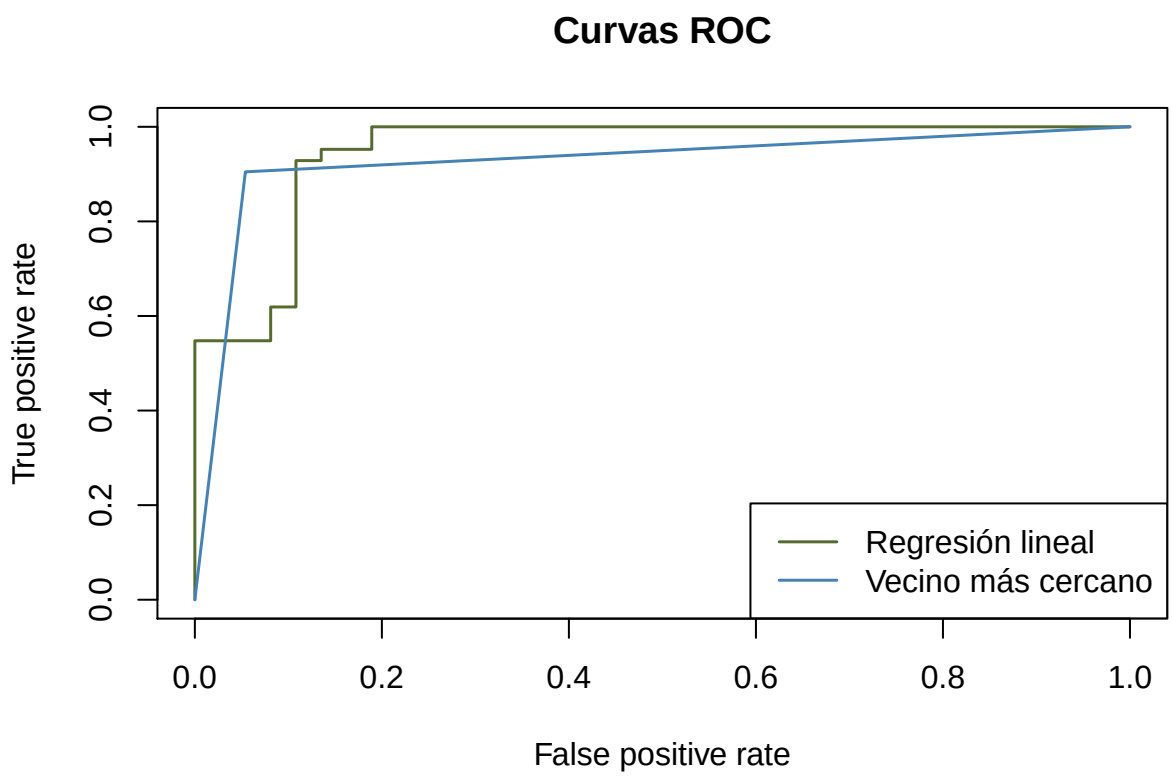
```
# install.packages("ROCR")
library ( ROCR )
```

```
## Loading required package: gplots
```

```
##
## Attaching package: 'gplots'
```

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

```
curveROC <- function (probabilities, labels, model.knn=F, ...) {
  if(model.knn){
    prob <- attributes(probabilities)$"prob"
    probabilities <- ifelse(probabilities == F, 1-prob, prob)
  }

  prediction <- prediction(probabilities, labels)

  # Calculamos la curva para las tasas de verdaderos y falsos positivos
  performance <- performance(prediction, "tpr", "fpr")
  plot(performance, ...)

  # Devolvemos el estimador AUC - el área bajo la curva
  return(attributes(performance(prediction, "auc"))$"y.values"[[1]])
}

mod.lin.auc <- curveROC(mod.lin.pred, Auto.test$mpg01, col="darkolivegreen", lwd=1.5, main="Curvas ROC")
knn.auc <- curveROC(knn.pred, Auto.test$mpg01, model.knn=T, add=T, col="steelblue", lwd=1.5)

legend("bottomright", c("Regresión lineal", "Vecino más cercano"),
       col=c("darkolivegreen", "steelblue"), lwd=1.5)
```



A la vista de gráfico no parece muy claro qué modelo tiene un comportamiento más ajustado. Esto lo podemos comprobar observando que el área bajo la curva es mayor en en regresión lineal que en kNN:

$AUC_{kNN} = 0.9253539$, $AUC_{reg} = 0.9485199$. Concluimos así que el ajuste de regresión lineal es algo mejor, aunque no por mucho, que el vecino más cercano.

Bonus - 1

Regresión logística

Para usar validazión cruzada con regresión logística vamos a utilizar la función `cv.glm` de la librería `boot`. Esta función hace justo lo que queremos: estima el error de predicción con validación cruzada para modelos lineales generalizados. Este valor viene como el primer elemento de un vector llamado `delta`, componente de la lista devuelta por la función.

La sintaxis es análoga a la usada en la función `glm`, aunque ahora hay que esepecificar el modelo ya ajustado y el número de particiones que queremos hacer.

```
# Cargamos la librería boot, que puede ser instalada con
# install.packages("boot")
library(boot)

# Hacemos validación cruzada
mod.lin.cv <- cv.glm(mod.lin, data=Auto.train, K=5)

# Tomamos la estimación del error de predicción
mod.lin.cv.err <- mod.lin.cv$delta[1]
```

Obtenemos así que el error de predicción es de un 8.3842027%, `I(100*(mod.lin.error-mod.lin.cv.err))` puntos mejor que el obtenido sin validación cruzada.

Vecino más cercano

Para el vecino más cercano la técnica cambia un poco, ya que aunque tenemos una función `knn.cv`, esta no devuelve un error, sino directamente las predicciones. Además, recibe el conjunto completo de datos con todas las muestras. La validación cruzada consiste en realizar la técnica del *leave-one-out*, es decir, calcular las distancias a cada muestra sin contar la propia muestra. Esto permite no influir y sobreajustar, lo que previsiblemente nos dará un mejor error de test.

Veámoslo:

```
# Predecimos con el k calculado anteriormente y con leave-one-out cross-validation
knn.cv.pred <- knn.cv(norm.full, as.factor(Auto$mpg01), k = knn.k)

# Calculamos de nuevo el error
knn.cv.error <- mean(Auto$mpg01 != knn.cv.pred)
```

Obtenemos así que el error de predicción es de un 5.6122449%, `I(100*(knn.k.error-knn.cv.error))` puntos mejor que el obtenido sin validación cruzada.

En ambos casos, tanto en el modelo de regresión como en el del vecino más cercano, hemos mejorado el error. Vemos así que la validación cruzada es una técnica potente que da buenos resultados.

Ejercicio 2

Cargamos la librería necesaria y estudiamos un poco la base de datos como hicimos antes:

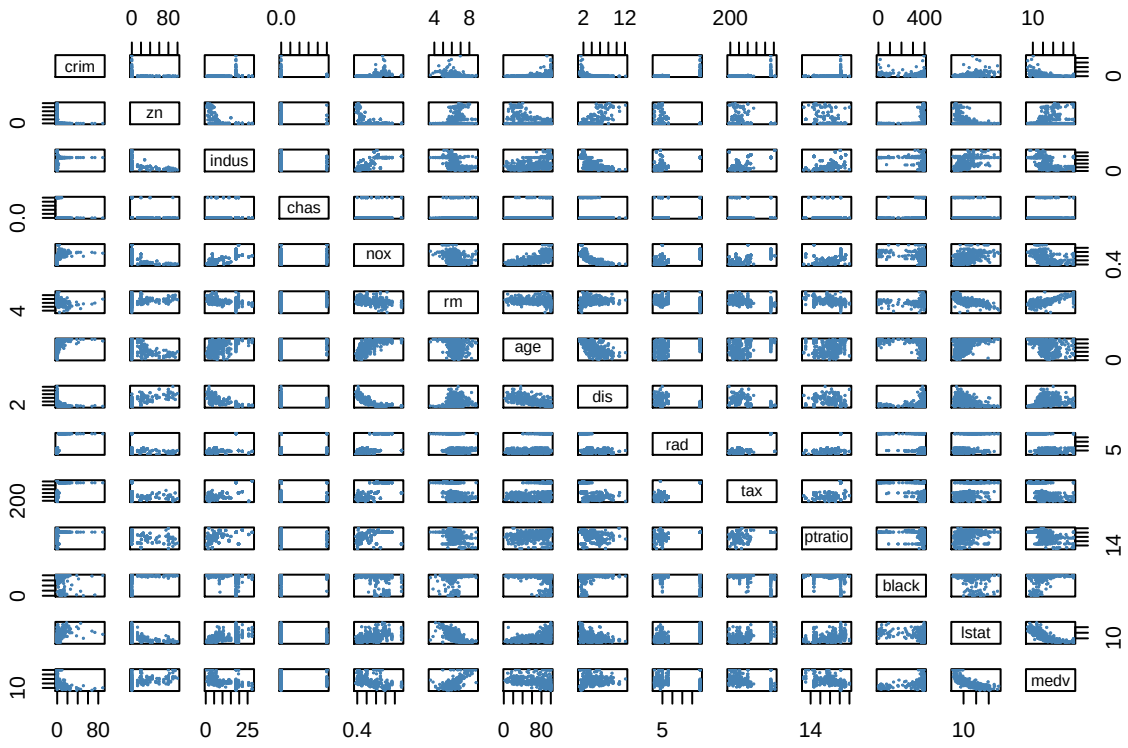
```r
# Cargamos la librería necesaria para usar la base de datos Boston
# Para usarla, hay que instalar con la orden
# install.packages('MASS')
library(MASS)

# Usamos Boston por defecto, evitando así poner el prefijo Boston$
# siempre que queramos acceder a una característica de esa base de datos
attach(Boston)
```

Si ejecutamos las órdenes siguientes

```r
class(Boston)
dim(Boston)
colnames(Boston)
```

podemos obtener información de la forma que tiene nuestra base de datos. Vemos así que tiene forma de data.frame, con 506 filas y 14 columnas, cuyos nombres son los siguientes: crim, zn, indus, chas, nox, rm, age, dis, rad, tax, ptratio, black, lstat, medv.

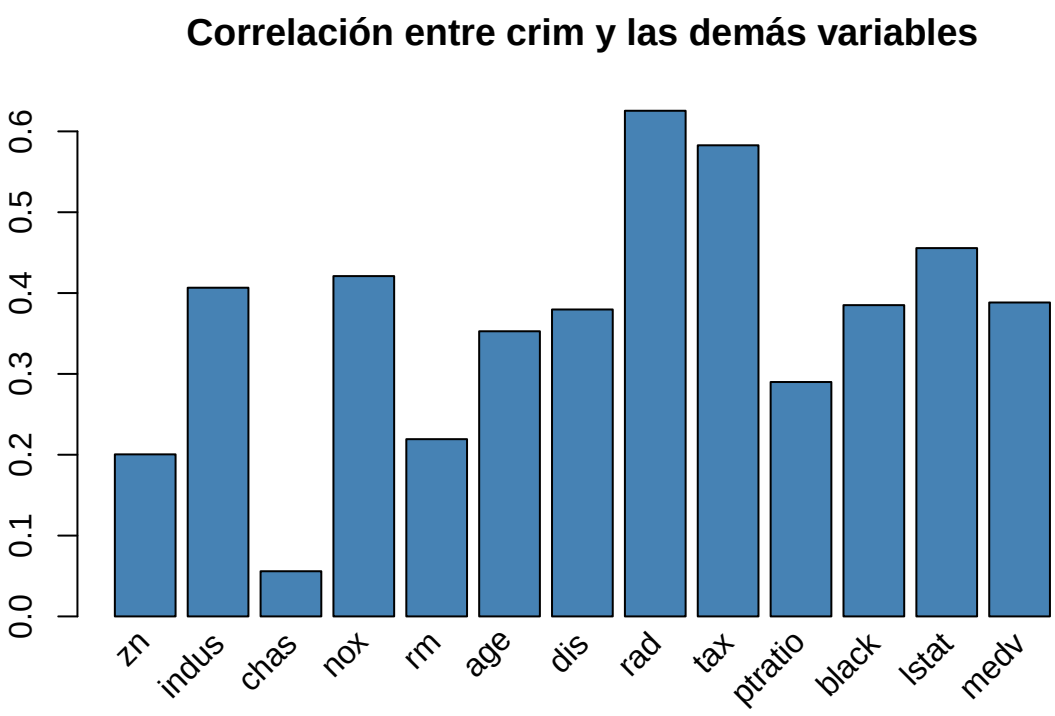
```r
# Visualizamos la relación entre todos los pares de variables
pairs(Boston, pch=20, cex=0.2, col="steelblue")
```



Podemos estudiar de forma numérica la correlación entre `crim` y las demás variables:

```r
# Tomamos los valores absolutos de la correlación entre crim y todas las demás variables,
# sin incluirse a sí misma
corr <- abs(cor(Boston))["crim",-1]

# Visualizamos el grado de correlación en un gráfico de barras.
# Creamos el gráfico.
bp <- barplot(corr, axes = FALSE, axisnames = FALSE, col = "steelblue",
              main="Correlación entre crim y las demás variables")
```

```r
# Añadimos el texto, girado 45 grados.
text(bp, par("usr")[3]-0.02, labels = colnames(Boston)[-1],
     srt = 45, adj = 1, xpd = TRUE)

# Dibujamos los ejes.
axis(2)
```



Por último, antes de entrar con el ejercicio en sí, dividimos la muestra en entrenamiento y test como hicimos antes:

```r
# Vector de índices para la muestra de entrenamiento (80%)
trainIdx  <- sample(nrow(Boston), size=0.8*nrow(Boston))

# Vector de índices para la muestra de test
testIdx   <- setdiff(1:nrow(Boston), trainIdx)

# Obtenemos las muestras de entrenamiento y de test
Boston.train <- as.matrix(Boston[trainIdx, ])
Boston.test  <- as.matrix(Boston[testIdx, ])
```

Apartado a

Vamos ahora a usar el método LASSO para hacer una selección de variables. Tras cargar la librería necesaria, hacemos validación cruzada con la función `glmnet` y el atributo `alpha = 1`, que indica que vamos a usar un método LASSO. Esto nos devuelve un valor para el parámetro de regularización:

```r
# Cargamos la librería glmnet, que debe ser instalada con la orden
# install.packages("glmnet")
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-5
```

```
# Hacemos validación cruzada para encontrar el mínimo lambda (el parámetro de regularización
# que minimiza el error medio de validación cruzada). Tomamos alpha = 1 para realizar un método
# LASSO
lasso.cv <- cv.glmnet(Boston.train[, -1], Boston.train[,"crim"], alpha = 1)
lasso.lambda <- lasso.cv$lambda.min
```

Podemos ahora calcular los coeficientes solicitados usando la función `predict` con el parámetro `type = "coefficients"`. Esto nos dará una serie de coeficientes para cada variable, y cuanto mayor sea este en valor absoluto, mayor correlación habrá con la variable a predecir.

```
# Calculamos ahora los coeficientes solicitados:
lasso.coefs <- predict(lasso.cv, Boston.test[,-1], s = lasso.lambda, alpha = 1,
                       type = "coefficients")

# Eliminamos la primera fila, que contiene el valor de Intercept; en este momento no nos interesa
lasso.coefs <- lasso.coefs[-1,]
```

Para elegir el umbral que determinará qué variables seleccionaremos y cuáles dejaremos atrás, vamos a encapsular en una función el cálculo del error que produce la selección de variables dado un umbral.

```
# Función para calcular el error residual estándar (raíz del error cuadrático medio)
# entre las variables predecidas y las reales
RSE <- function(pred, real){
  return(sqrt(mean((pred - real)^2)))
}

# Hacemos la selección de variables dependiente de un umbral pasado como parámetro
lasso.seleccionar <- function(lasso.umbral){
  # Extraemos los nombres de las variables cuyos coeficientes (en valor absoluto)
  # superan el umbral prefijado
  lasso.selec <- attributes(lasso.coefs[abs(lasso.coefs) > lasso.umbral])$names

  # Por último, hacemos las predicciones con las variables seleccionadas
  lasso.glm <- glmnet(as.matrix(Boston.train[,lasso.selec]), Boston.train[,"crim"], alpha = 1, lambda =
  lasso.pred <- predict(lasso.glm, Boston.test[,lasso.selec], s=lasso.lambda, alpha=1)

  # Devolvemos el error residual estándar (raíz del error cuadrático medio)
  #con las variables seleccionadas
  return(c(lasso.umbral, RSE(lasso.pred, Boston.test[,"crim"])))
}
```

Podemos así iterar sobre los umbrales y ver cuál produce un error menor, de manera que nos quedemos con el umbral óptimo:

```
# Calculamos los errores para todos los umbrales de 0.1 a 0.7, con saltos de 0.05
lasso.errores <- t(sapply(seq(0.0,0.7,0.05), lasso.seleccionar))

# Recuperamos el umbral para el que el error calculado es mínimo
lasso.umbral <- lasso.errores[which.min(lasso.errores[,2]),1]

# Nos quedamos con las mejores variales según el umbral devuelto
lasso.selec <- attributes(lasso.coefs[abs(lasso.coefs) > lasso.umbral])$names
```

Tras todo este trabajo, las variables seleccionadas por el método LASSO son: chas, nox, rm, dis, rad, ptratio,

lstat, medv.

Apartado b

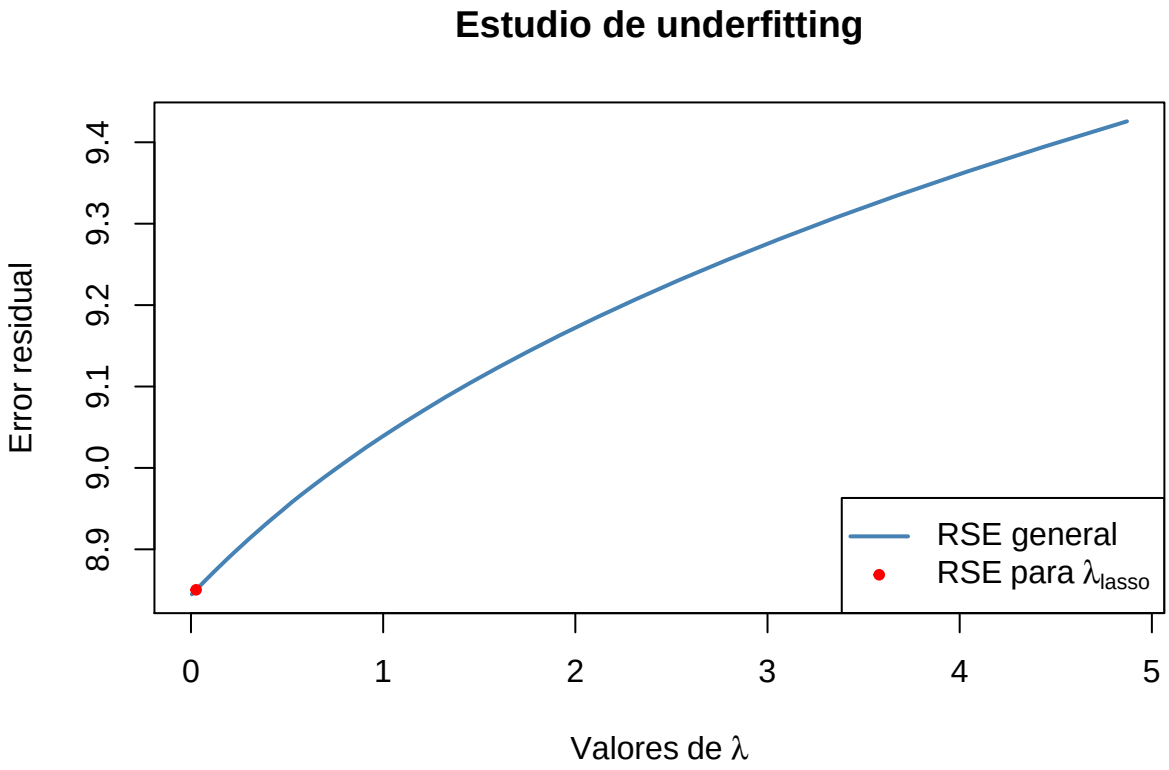
Al igual que en el apartado anterior, vamos a encapsular en una función el cálculo del error de un modelo de regresión dado un parámetro de regularización con *weight-decay*:

```
# Función dependiente del parámetro lambda que nos permitirá después
# hacer un estudio del underfitting
weight.decay.rse <- function(param.lambda){
  # Hacemos la regresión, apuntando que es weight decay con el parámetro alpha = 0
  ridge <- glmnet(Boston.train[, lasso.selec], Boston.train[,"crim"],
                  alpha = 0, lambda = param.lambda)

  # Calculamos las predicciones del modelo ajustado con el lambda calculado anteriormente
  ridge.pred <- predict(ridge, Boston.test[,lasso.selec], s=param.lambda, alpha=0)

  # Calculamos el RSE de estas predicciones
  ridge.rse <- RSE(ridge.pred, Boston.test[,"crim"])

  return(ridge.rse)
}
```

Usando esta función y el parámetro $\lambda = 0.0266038$ obtenido con el modelo LASSO, La estimación del error residual estándar es de 8.8502678.

Para comprobar si hay *underfitting* vamos a analizar el modelo de *bias-variance* estudiado en teoría. Vamos a calcular los errores asociados a la regresión usando los valores de λ testeados en la validación cruzada que hicimos con LASSO:

```
# Estudio de underfitting
ridge.errors <- sapply(lasso.cv$lambda, weight.decay.rse)

# Generación de la gráfica
plot(lasso.cv$lambda, ridge.errors, pch=20, type="l", cex=0.5, lwd=2, col="steelblue",
     xlab=expression(Valores~de~lambda), ylab="Error residual", main="Estudio de underfitting")
points(x=lasso.lambda, y=weight.decay.rse(lasso.lambda), pch=20, col="red")
legend("bottomright", c("RSE general", expression(RSE~para~lambda[lasso])), col=c("steelblue","red"),
       pch=c(NA,20), lwd=c(2,NA))
```



A la vista de este resultado podemos concluir que el comportamiento de los residuos no presenta ningún indicio de *underfitting* con respecto al parámetro λ. En todo caso podríamos hablar de sobreajuste, ya que el error residual en la muestra de test puede ser mejorado suavizando el modelo; es decir, aumentando el valor del parámetro de regularización.

Apartado c

Tal y como hicimos para la variable binaria `mpg01`, creamos una nueva variable `crim1` en la base de datos de Boston en función de la mediana.

```
# Creamos una nueva variable booleana, crim1, en función de la mediana,
# y la añadimos a la base de datos
crim1 <- ifelse(crim > median(crim), 1, -1)
Boston <- data.frame(crim1, Boston)

# Obtenemos las muestras de entrenamiento y de test
Boston.train <- Boston[trainIdx,]
Boston.test  <- Boston[testIdx,]
```

Vamos a escribir una función que devuelva el error de un modelo SVM dado un kernel de los aceptados por la función proporcionada por R e intentemos probarla con un núcleo lineal. La función escrita genera aleatoriamente particiones de entrenamiento y test, así que la llamaremos un número alto de veces para poder tomar la media de todos los errores devueltos:

```
# Devuelve el porcentaje de muestras mal clasificadas que produce
# un modelo SVM con el kernel del tipo kernel.name.
svm.error <- function(kernel.name){
  # Vector de índices para la muestra de entrenamiento (80%)
  trainIdx  <- sample(nrow(Boston), size=0.8*nrow(Boston))
  # Vector de índices para la muestra de test
```

```
  testIdx   <- setdiff(1:nrow(Boston), trainIdx)

  # Obtenemos las muestras de entrenamiento y de test
  Boston.train <- Boston[trainIdx,]
  Boston.test  <- Boston[testIdx,]

  # Ajustamos el modelo
  svm <- svm(crim1~., data = Boston.train, kernel = kernel.name)

  # Hacemos la predicción
  svm.pred <- predict(svm, Boston.test, type = "response")

  # Calculamos la variable según las predicciones
  pred <- ifelse(svm.pred > 0, 1, -1)

  # Devolvemos el porcentaje de muestras mal clasificadas
  return(mean(pred != Boston.test[,"crim1"]))
}

# Ajustamos un modelo de SVM con un núcleo lineal
svm.linear <- mean(replicate(100,svm.error("linear")))
```

Tras repetir el experimento 100 veces con particiones aleatorias de entrenamiento y test, vemos que el SVM con núcleo lineal devuelve un error de $E_{test} = 17.4215686\%$. A la vista de un error tan alto, vamos a ajustar el modelo con otros núcleos, estudiando cuál es el mejor.

```
# Ajustamos SVM con los núcleos disponibles
svm.polynomial <- mean(replicate(100,svm.error("polynomial")))
svm.radial <- mean(replicate(100,svm.error("radial")))
svm.sigmoid <- mean(replicate(100,svm.error("sigmoid")))
```

Los errores, obtenidos con la misma técnica de repetir 100 veces las particiones y calcular el porcentaje de muestras mal clasificadas, son los siguientes:

- Núcleo lineal: 17.4215686%
- Núcleo polinómico: 16.2843137%
- Núcleo de base radial: 11.4901961%
- Núcleo sigmoidal: 39.2647059%

Podemos concluir por tanto que el mejor modelo de entre los SVM analizados es aquel que usa el núcleo de base radial, consiguiendo una tasa de error de sólo el 11.4901961%.

Ejercicio 3

Apartado 1

Cargamos las librerías necesarias y dividimos, como antes, la base de datos en conjuntos de entrenamiento y test:

```
# Cargamos las librerías randomForest y gbm, que se pueden instalar con las órdenes
# install.packages(c("randomForest","gbm"))
library(randomForest)
```

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

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

```
library(gbm)
```

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

```
##
## Attaching package: 'survival'
```

```
## The following object is masked from 'package:boot':
##
##     aml
```

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

```
##
## Attaching package: 'lattice'
```

```
## The following object is masked from 'package:boot':
##
##     melanoma
```

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

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

```
## Loaded gbm 2.1.1
```

```
# Devolvemos la base de datos a su estado original
Boston <- Boston[, !(names(Boston) %in% "crim1")]

# Vector de índices para la muestra de entrenamiento (80%)
trainIdx  <- sample(nrow(Boston), size=0.8*nrow(Boston))

# Vector de índices para la muestra de test
testIdx   <- setdiff(1:nrow(Boston), trainIdx)

# Obtenemos las muestras de entrenamiento y de test
Boston.train <- Boston[trainIdx, ]
Boston.test  <- Boston[testIdx, ]
```

Apartado 2

Sabemos que *bagging* no es más que un caso particular de *random forest* con los parámetros $m = p$; es decir, el número de variables usadas es el número total de variables disponibles.

```
# Ajustamos bagging
bag <- randomForest(medv ~ ., data = Boston, subset = trainIdx, mtry = ncol(Boston)-1, importance = T)
```

El significado de cada parámetro usado en la llamada a `randomForest` es el siguiente:

- `medv ~ .` Fórmula en la que indicamos que `medv` es la variable predecir y todas las demás (`.`) se usen como predictoras
- `data = Boston` Conjunto completo de datos, con todas las variables (incluso la que se quiere predecir, ya que `randomForest` ya lo tendrá en cuenta) y ambos subconjuntos: los de entrenamiento y los de test.
- `subset = trainIdx` Vector de índices que indican qué filas del parámetro `data` son las correspondientes al subconjunto de entrenamiento.
- `mtry = ncol(Boston)-1` Número de variables usadas en el cómputo. Para poder hacer bagging, tenemos que indicar que se usen todas las variables disponibles; es decir, el número de columnas menos uno: todas las variables menos la que se quiere predecir.

- `importance = T` Parámetro de configuración para indicar que se evalúe la importancia de los predictores.

Podemos ya usar el modelo para predecir la variable `medv` y calcular así el error cuadrático medio con respecto a los valores reales:

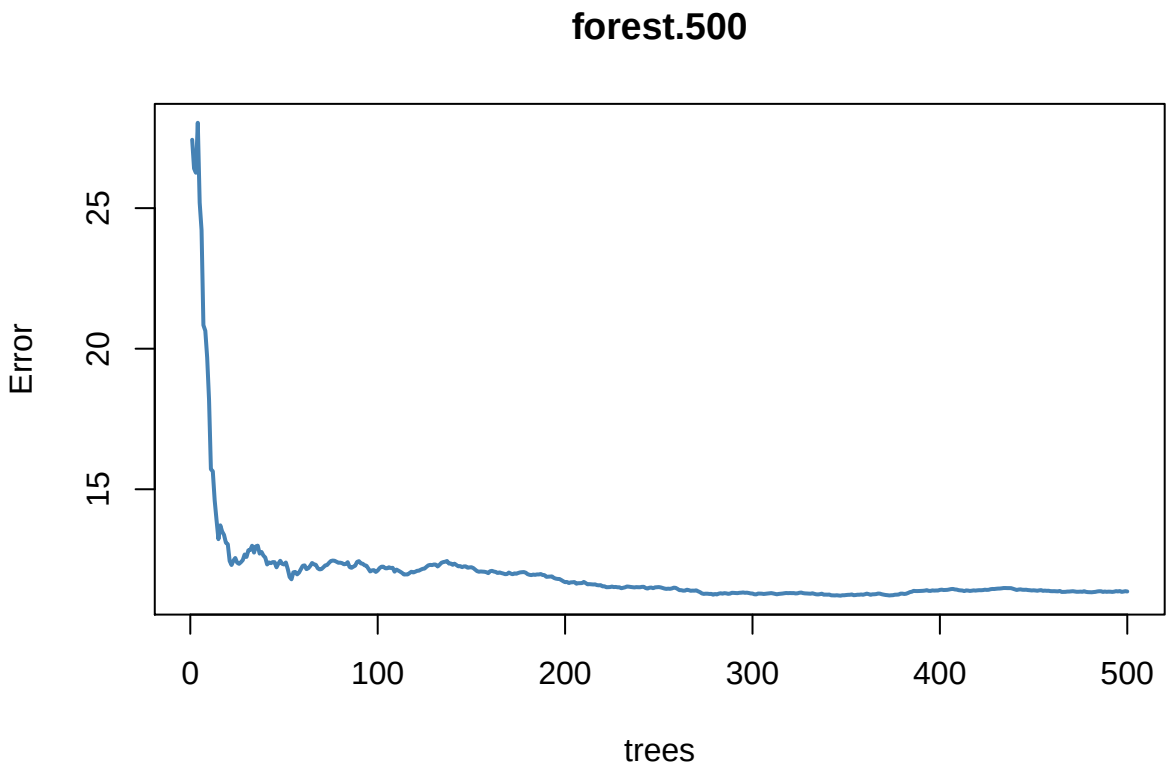
```
# Predecimos la variable medv con el modelo ajustado
bag.pred <- predict(bag, Boston.test)

# Calculamos el error cuadrático medio
bag.error <- mean((bag.pred - Boston.test[,"medv"])^2)
```

Este modelo nos da un error de 10.6299382.

Apartado 3

Vamos a ajustar el modelo con los parámetros por defecto y estudiar su comportamiento:

```
# Ajustamos randomForest con el valor por defecto de número árboles: 500
forest.500 <- randomForest(medv ~ ., data = Boston, subset = trainIdx, ntrees=500, importance = T)

# Generamos el gráfico del error producido con cada valor de ntrees de 1a 500
plot(forest.500, col="steelblue", lwd=2)
```



En la gráfica anterior se puede observar cómo el error se estabiliza a partir de 50 o 60 árboles. Los datos referentes a esta gráfica se encuentran en el componente `mse` de la lista devuelta por `randomForest`, así que podemos buscar directamente cuál es el valor de `ntrees` para el que se alcanza el mínimo error cuadrático medio:

```
# Tomamos el valor de ntrees en el que se minimiza el error
forest.best.N <- which(forest.500$mse == min(forest.500$mse))

# Reajustamos el modelo con este valor como ntrees
```

```
forest.best <- randomForest(medv ~ ., data = Boston, subset = trainIdx,
                            ntrees=forest.best.N, importance = T)
```

Analicemos ahora si este procedimiento ha mejorado el error anterior. Calculemos entonces el error cuadrático medio para el ajuste por defecto y para este último, en el que hemos optimizado el número de árboles:

```
# Calculamos las predicciones con ambos modelos
forest.500.pred <- predict(forest.500, Boston.test)
forest.best.pred <- predict(forest.best, Boston.test)

# Calculamos el error cuadrático medio con respecto a los valores reales
forest.500.err <- mean((forest.500.pred - Boston.test[,"medv"])^2)
forest.best.err <- mean((forest.best.pred - Boston.test[,"medv"])^2)
```

Tenemos unos errores de 11.3426263 para el valor por defecto y de 11.3254717 para el valor optimizado. Como vemos, el valor optimizado es algo mayor, así que parece que estamos cayendo en un sobreajuste.

Intentemos controlar esto con un método de validación cruzada. Para ello implementamos una función que devuelve el error del *random forest* con un número de árboles fijado y usand para ello validación cruzada:

```
# Calcula el error cuadrático medio de randomForest, con el parámetro
# ntrees especificado, con 10 particiones aleatorias
errorNtree <- function(param.ntrees){

  # Calcula el error cuadrático medio en una partición 80-20 aleatoria
  forest.fold <- function(){
    # Vector de índices para la muestra de entrenamiento (80%)
    trainIdx  <- sample(nrow(Boston), size=0.8*nrow(Boston))

    # Vector de índices para la muestra de test
    testIdx   <- setdiff(1:nrow(Boston), trainIdx)

    # Obtenemos las muestras de entrenamiento y de test
    Boston.train <- Boston[trainIdx, ]
    Boston.test  <- Boston[testIdx, ]

    forest <- randomForest(medv ~ ., data = Boston, subset = trainIdx,
                           ntrees=param.ntrees, importance = T)

    pred <- predict(forest, Boston.test)

    return(mean((pred - Boston.test[,"medv"])^2))
  }

  # Calcula el error medio en 10 particiones aleatorias
  error <- mean(replicate(10, forest.fold()))

  return(c(param.ntrees,error))
}
```

Podemos ahora iterar sobre algunos valores del número de árboles e intentar buscar el mínimo:

```
# Calculamos los errores para ntrees igual a 30, 50, 70, 90, ..., 350
errores <- t(sapply(seq(30, 350, 20), errorNtree))

# Obtenemos el ntree tal que minimiza los errores obtenidos
```

```
best.ntree <- errores[which(errores[,2] == min(errores[,2])), 1]
```

Intentaremos así mejorar el error reajustando el modelo con el parámetro que acabamos de obtener:

```
# Reajustamos, recalculamos predicciones y vemos el error cuadrático
forest.cv <- randomForest(medv ~ ., data = Boston, subset = trainIdx,
                          ntrees=best.ntree, importance = T)
forest.cv.pred <- predict(forest.cv, Boston.test)
forest.cv.err <- mean((forest.cv.pred - Boston.test[,"medv"])^2)
```

Tenemos por tanto los siguientes errores:

- Valor por defecto: $ntrees = 500 \rightarrow E_{test} = 11.3426263$
- Valor que minimiza los `mse` devueltos por el modelo: $ntrees = 347 \rightarrow E_{test} = 11.3254717$
- Valor encontrado con validación cruzada: $ntrees = 350 \rightarrow E_{test} = 11.8069902$

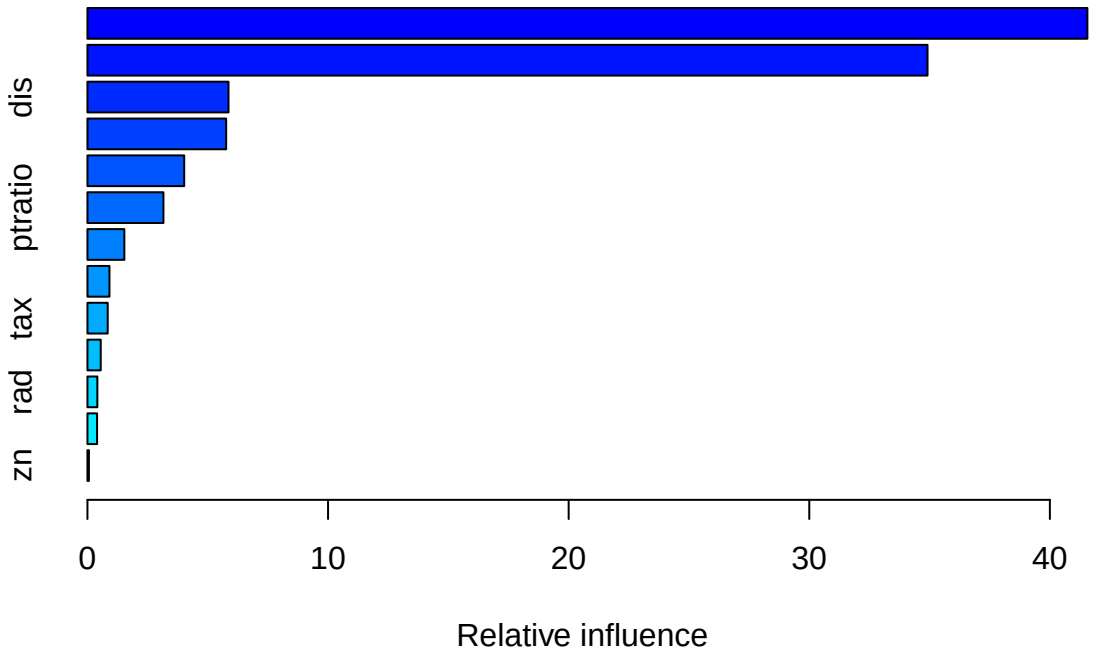
Es claro entonces que en el caso de la minimización de los `mse` hemos caído en el sobreajuste, ya que el valor del error en la muestra de test aumenta. En el último caso, calculado con validación cruzada, hemos tenido en cuenta varias particiones y hemos conseguido un error ligeramente menor, aunque esencialmente igual.

Sería sensato quedarse con este último valor, que teóricamente tiene un poder de generalización mayor, que el tomado por defecto. Sin embargo, y aunque el razonamiento seguido es correcto, no encontramos en este estudio, con estos datos concretos, conclusiones fuertes que respalden con contundencia la elección del parámetro `ntrees` =350.

Apartado 4

Ajustamos el modelo de boosting con la función `gbm` y los parámetros usuales:

```
boost <- gbm(medv ~ ., data = as.data.frame(Boston.train),
             distribution = "gaussian" , n.trees=5000, interaction.depth=4)

# Resumimos la información devuelta por el modelo y
# generamos un gráfico de la influencia de cada variable
summary(boost)
```



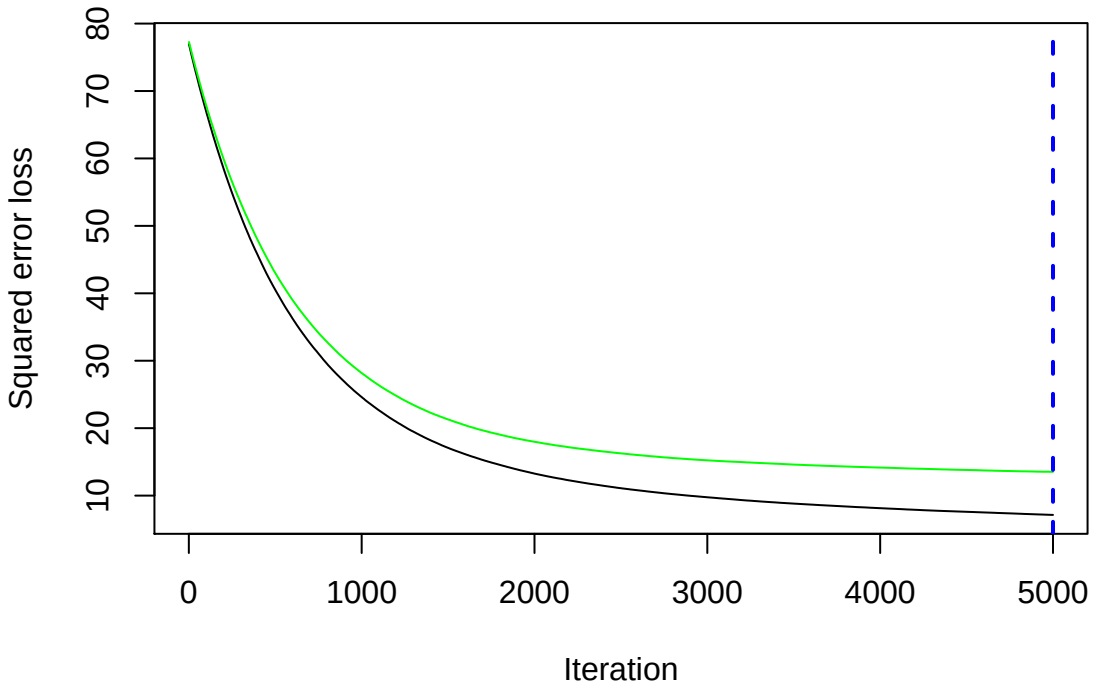

```
##             var     rel.inf
## lstat     lstat 41.55897219
## rm           rm 34.91462787
## dis         dis  5.86353262
## nox         nox  5.76522240
## crim       crim  4.02159212
## ptratio ptratio  3.15883614
## age         age  1.53374893
## black     black  0.91517715
## tax         tax  0.84202271
## chas       chas  0.55346905
## rad         rad  0.41194184
## indus     indus  0.39964278
## zn           zn  0.06121421
```

Podemos usar ahora el modelo para predecir la variable `medv`:

```
boost.pred <- predict(boost, Boston.test, n.trees=5000)
boost.err <- mean((boost.pred - Boston.test[,"medv"])^2)
```

El error de boosting con los parámetros fijados es de `I(boost.err)`, mayor que el conseguido hasta ahora con *bagging* y *random forest*. Podemos intentar mejorar este error ajustando los parámetros con validación cruzada:

```
# Reajustamos con 5cv
boost.cv <- gbm(medv ~ ., data = as.data.frame(Boston.train), cv.folds=5,
                distribution = "gaussian" , n.trees=5000, interaction.depth=4)

# Buscamos el valor de n.trees óptimo con validación cruzada
boost.cv.n <- gbm.perf(boost.cv,method="cv")
```




```
# Hacemos las predicciones con este valor
boost.cv.pred <- predict(boost.cv, Boston.test, n.trees=boost.cv.n)
boost.cv.err <- mean((boost.cv.pred - Boston.test[,"medv"])^2)
```

De nuevo, la validación cruzada nos da un error igual o peor al anterior. Mientras que el error de boosting sin optimizar era de 11.0287793, el optimizado mediante validación cruzada es de 10.8881804. Esta diferencia es tan pequeña que no aporta nada sobre el modelo por defecto, así que sería sensato quedarse con el primero, que es menos costoso computacionalmente que el optimizado y devuelve un error menor.

Haciendo un análisis global, y si queremos elegir entre *bagging*, *random forest* y *boosting*, lo más inteligente en este caso sería usar *random forest* con validación cruzada para asegurar un error menor y un poder de generalización mayor.

Ejercicio 4

Generamos las muestras de entrenamiento y test, anclamos la base de datos OJ al espacio de nombres para evitar escribir siempre el prefijo y cargamos la librería tree:

```
# Vector de índices para la muestra de entrenamiento (80%)
trainIdx  <- sample(nrow(OJ), size=800)

# Vector de índices para la muestra de test
testIdx   <- setdiff(1:nrow(OJ), trainIdx)

# Obtenemos las muestras de entrenamiento y de test
OJ.train <- OJ[trainIdx, ]
OJ.test  <- OJ[testIdx, ]

attach(OJ)

# Incluimos la librería tree, que se puede instalar con la orden
```

```r
# install.packages("tree")
library(tree)
```

Ajustamos ahora un modelo de árbol con la función `tree`, que sigue la misma sintaxis que los modelos usados hasta ahora:

```r
arbol <- tree(Purchase ~ ., OJ.train)
```

Apartado 2

Estudiamos ahora los resultados obtenidos con la órden `summary`:

```r
# Generamos un resumen del modelo obtenido
summary(arbol)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
## Number of terminal nodes:  7
## Residual mean deviance:  0.7941 = 629.7 / 793
## Misclassification error rate: 0.1675 = 134 / 800
```

Observamos que el modelo tiene una tasa de error de entrenamiento del 16.75%, bastante pobre.

El número de nodos terminales es de 7, aunque el número de variables usadas es únicamente 3, que son las siguientes: LoyalCH, PriceDiff, ListPriceDiff.

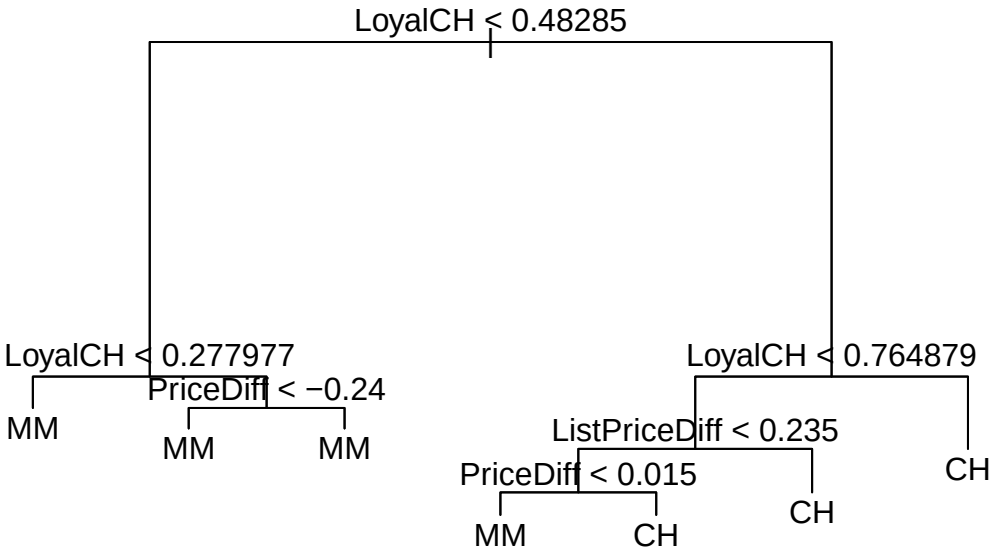
Con un error tan alto ya podemos intuir que este árbol no es el óptimo. Aunque es muy compacto, su poder de predicción es pobre, quizás por el bajo número de variables usadas.

Vemos por último que el modelo tiene una varianza residual media de 0.7940937.

Apartado 3

Si dibujamos el árbol con sus etiquetas

```r
# Generación del gráfico del árbol
plot(arbol)

#Le añadimos los nombres de las variables y los umbrales
text(arbol, pretty=0)
```



vemos que este árbol, aunque compacto, no es nada bueno. Lo primero que observamos es que toda la rama izquierda tiene la misma clase `MM`, así que podríamos ahorrarnos hasta dos subramas asignando a ese subárbol un nodo terminal que fuera `MM`. A la derecha se ve rápido que si lo primero que estudiáramos fuera la variable `PriceDiff` los cómputos serían más rápidos, ya que sólo de él depende que sea `MM` y no `CH`. Este árbol, en definitiva, no es en absoluto un buen resultado.

Apartado 4

Con el modelo ajustado en los apartados anteriores, podemos ya predecir la variable y generar la matriz de confusión, que nos dará mucha información sobre la efectividad del modelo:

```
# Cargamos la librería caret para poder generar la matriz de confusión.
# Se puede instalar con la orden
# install.packages("caret")
library(caret)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

```
## The following object is masked from 'Auto':
##
##     mpg
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:survival':
##
##     cluster
```

```
# Predecimos y calculamos la matriz de confusión
arbol.pred <- predict(arbol, OJ.test, type="class")
arbol.conf <- confusionMatrix(arbol.pred, OJ.test[,"Purchase"])
```

```r
# Visualizamos la matriz de confusión
arbol.conf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 134  12
##         MM  31  93
##
##                Accuracy : 0.8407
##                  95% CI : (0.7915, 0.8823)
##     No Information Rate : 0.6111
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6756
##  Mcnemar's Test P-Value : 0.006052
##
##             Sensitivity : 0.8121
##             Specificity : 0.8857
##          Pos Pred Value : 0.9178
##          Neg Pred Value : 0.7500
##              Prevalence : 0.6111
##          Detection Rate : 0.4963
##    Detection Prevalence : 0.5407
##       Balanced Accuracy : 0.8489
##
##        'Positive' Class : CH
##
```

En la matriz de confusión lo primero que vemos es una tabla resumiendo los falsos positivos y los falsos negativos. Como vemos, es mucho más frecuente el error de clasificar las muestras `CH` como `MM`; esto no debe sorprendernos, ya que la mayoría de nodos terminales en el árbol tienen la última clase.

Después tenemos información estadística de la precisión del test, que vemos que es del 84.0740741%. Por lo tanto, concluimos que la tasa de error del test es de un 15.9259259%.

La matriz de confusión, por último, nos da diversos estimadores como la especificidad o los porcentajes de positivos y negativos predecidos (donde la variable positiva es, como se indica al final, `CH`).

Apartado 5

Podemos optimizar el tamaño óptimo del árbol buscando los parámetros que hacen de la complejidad del árbol resultante la óptima. Esta búsqueda la vamos a hacer con `cv.tree`, una función que hace validación cruzada sobre el número de particiones indicadas y que mide el número de muestras mal clasificadas según el parámetro buscado. Vamos a tomar, como es usual, 5 particiones.

Además, vamos a usar el argumento `FUN = prune.misclass` para indicar que queremos que sea la tasa de error de clasificación la que guíe la validación cruzada y la poda.

```r
# Hacemos validación cruzada
arbol.cv <- cv.tree(arbol, K = 5, FUN=prune.misclass)

# Visualizamos la salida
arbol.cv
```

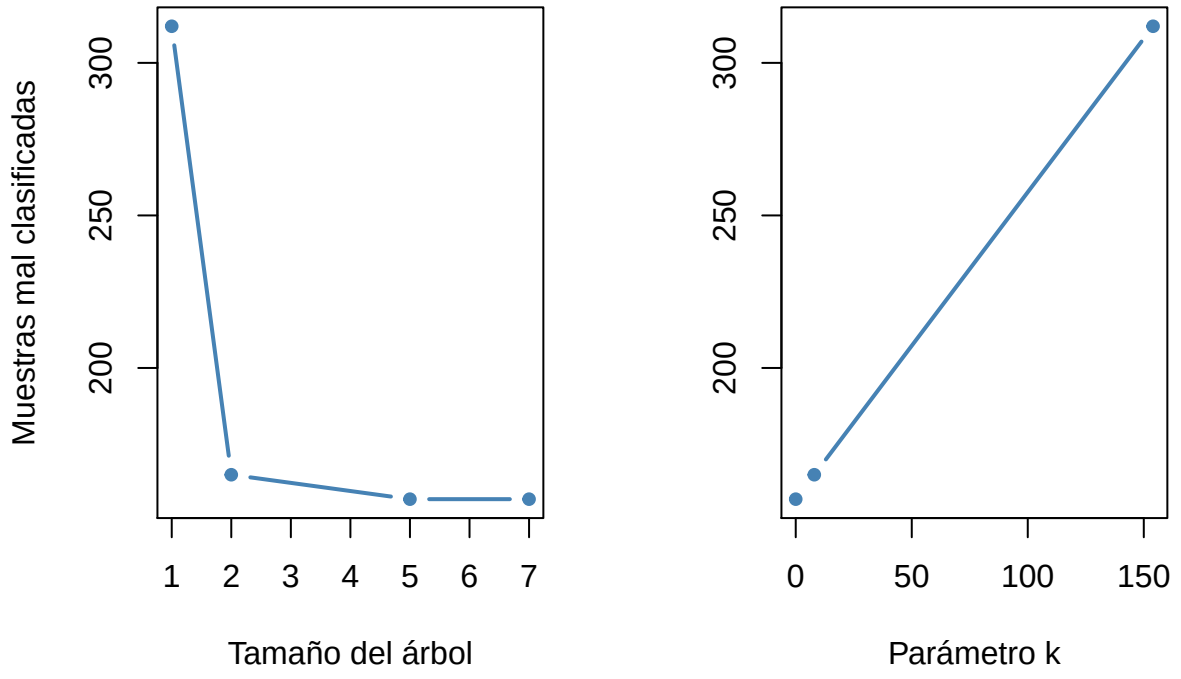
```
## $size
## [1] 7 5 2 1
##
## $dev
## [1] 157 157 165 312
##
## $k
## [1] -Inf    0    8  154
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Como vemos, el mejor tamaño para este árbol (es decir, el que produce un error -variable dev- menor) es de 5.

Bonus - 4

Podemos ahora ver dos gráficos: el tamaño del árbol contra el error de validación cruzada y el parámetro k contra ese mismo error:

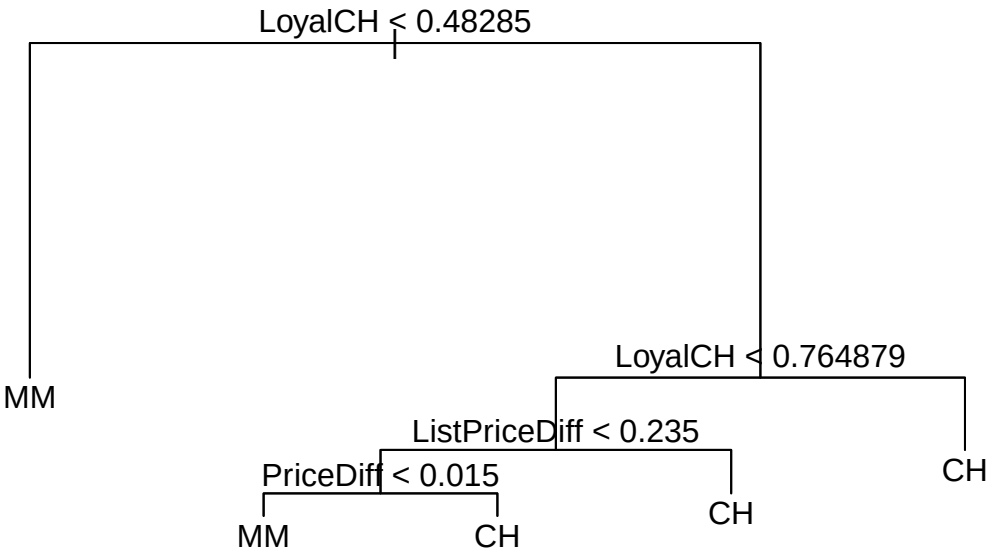
```
# Estudiamos el número de muestras mal clasificadas por cada tamaño de árbol considerado
# y por cada parámetro k
par(mfrow = c(1,2))
plot(arbol.cv$size, arbol.cv$dev, type ="b", pch=20, col="steelblue", lwd=2,
     xlab="Tamaño del árbol", ylab="Muestras mal clasificadas")
plot(arbol.cv$k, arbol.cv$dev, type ="b", pch=20, col="steelblue", lwd=2,
     xlab="Parámetro k", ylab="")
```




```
par(mfrow=c(1,1))
```

Una vez calculado y visualizado el tamaño óptimo de árbol, podemos mejorar con él el modelo presente:

```r
# Tomamos el mejor valor de tamaño
arbol.n <- min(arbol.cv$size[which(arbol.cv$dev == min(arbol.cv$dev))])

# Podamos el árbol
arbol.podado = prune.misclass(arbol, best=arbol.n)

# Dibujamos de nuevo el árbol
plot(arbol.podado)
text(arbol.podado, pretty=0)
```



Este árbol es mucho más compacto, pero veamos qué error de clasificación tiene en la muestra de test:

```r
# Predecimos y calculamos matriz de confusión
arbol.podado.pred <- predict(arbol.podado, OJ.test, type="class")
arbol.podado.conf <- confusionMatrix(arbol.podado.pred, OJ.test[,"Purchase"])

# Estudiamos matriz de confusión
arbol.podado.conf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 134  12
##         MM  31  93
##
##                Accuracy : 0.8407
##                  95% CI : (0.7915, 0.8823)
##     No Information Rate : 0.6111
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6756
##  Mcnemar's Test P-Value : 0.006052
##
##             Sensitivity : 0.8121
##             Specificity : 0.8857
##          Pos Pred Value : 0.9178
```

```
##          Neg Pred Value : 0.7500
##              Prevalence : 0.6111
##          Detection Rate : 0.4963
##    Detection Prevalence : 0.5407
##       Balanced Accuracy : 0.8489
##
##        'Positive' Class : CH
##
```

La tasa de error es ahora de 15.9259259%, igual a la anterior. Aunque no hay mejora, hay que notar que ahora el árbol es más compacto, legible y eficiente, así que hemos ganado en eficiencia y no hemos perdido en efectividad. En este caso la validación cruzada ha merecido la pena.