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.

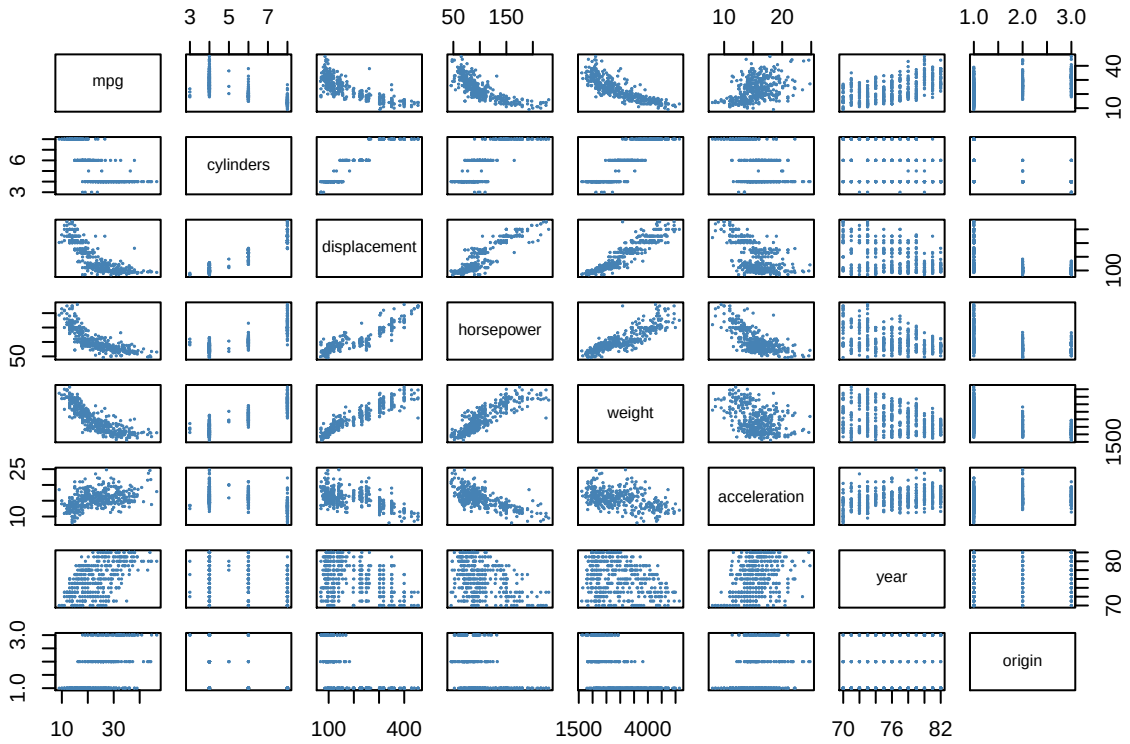
Podemos eliminar la última columna, que es el nombre del vehículo y la única variable no numérica, para poder trabajar con comodidad más adelante:

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

Apartado a

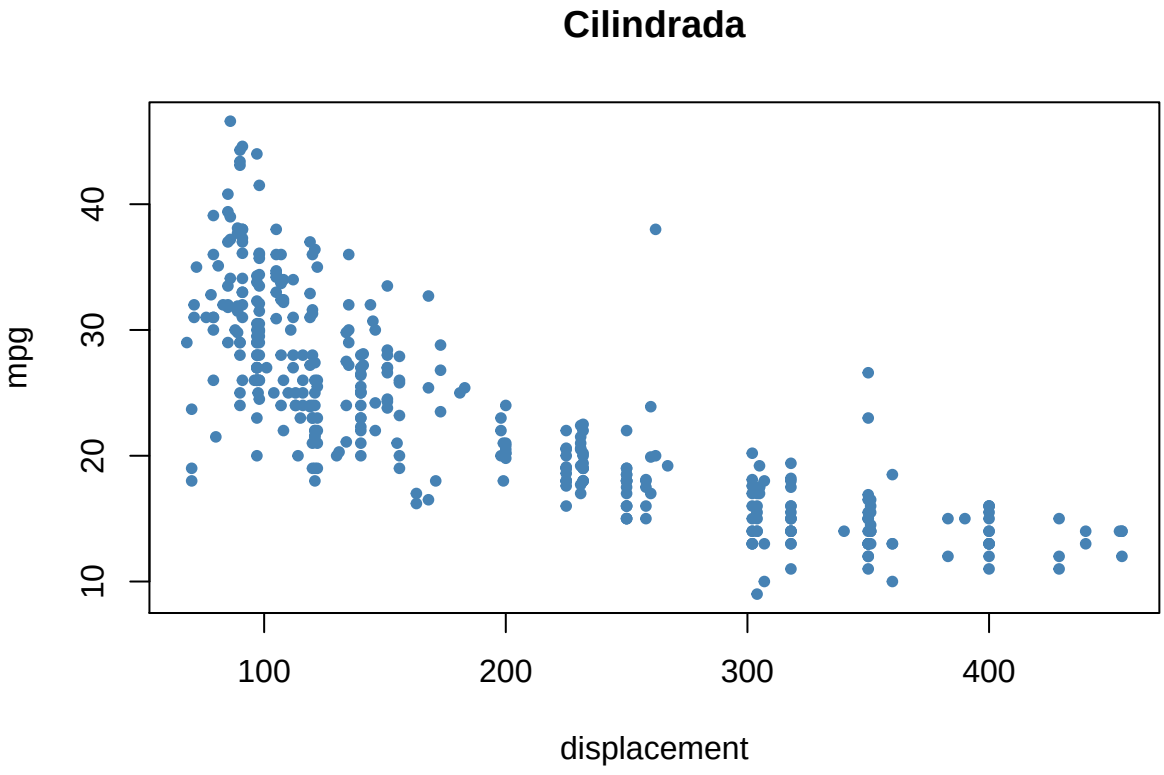
Para visualizar las dependencias entre `mpg` y las otras características podemos usar las funciones `pairs()` y `boxplot()`:

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

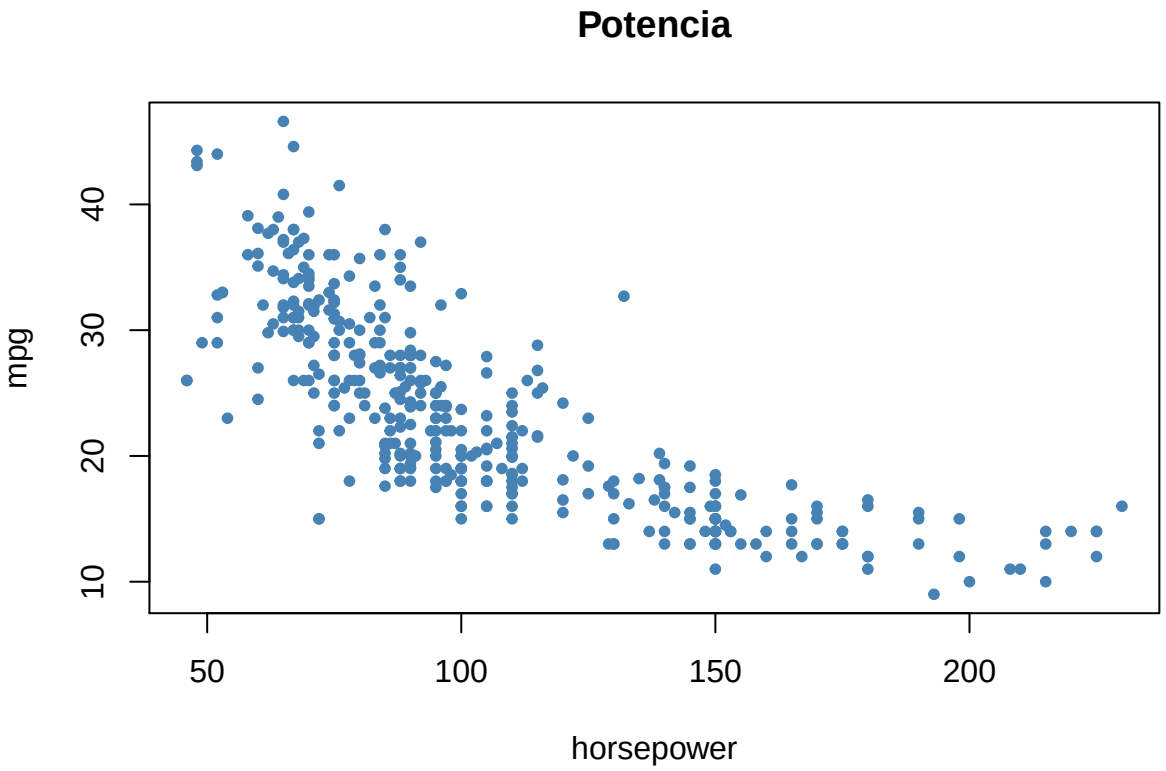


Podemos visualizar más de cerca las variables que parecen más relevantes para predecir la variable `mpg`:

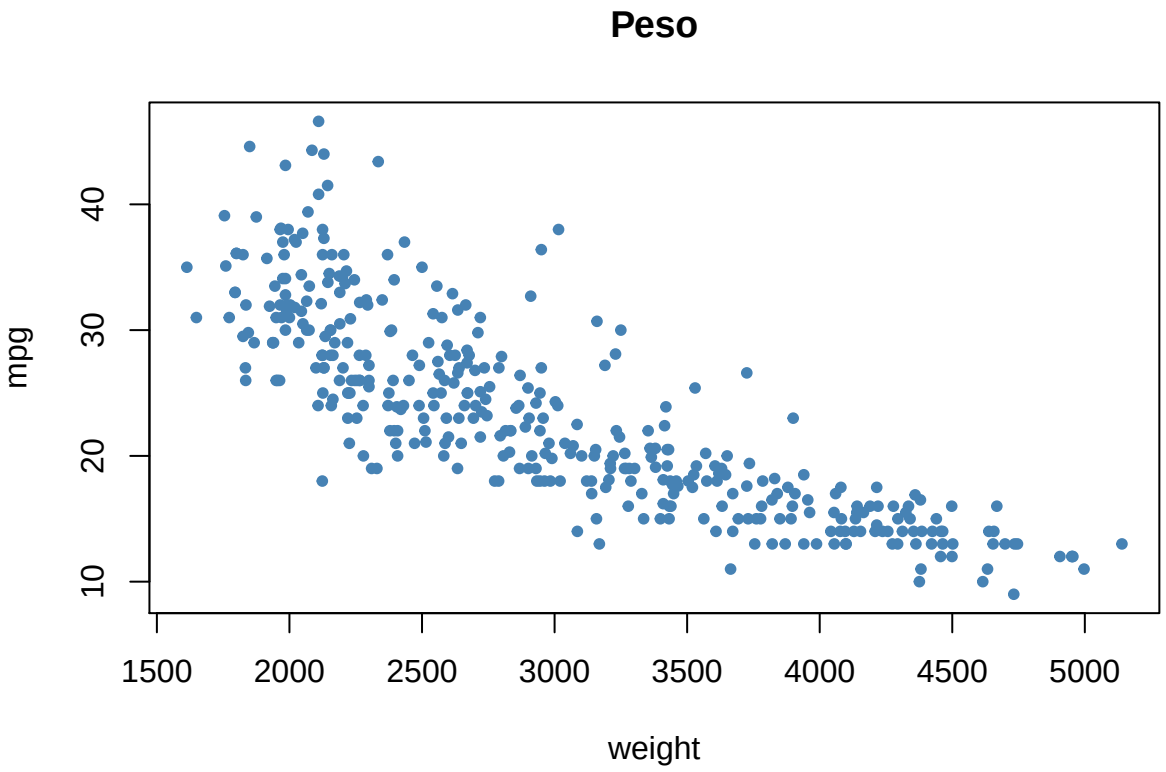
```
# Plot para mpg-displacement
plot(displacement, mpg, pch=20, col="steelblue",
     main="Cilindrada")
```



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



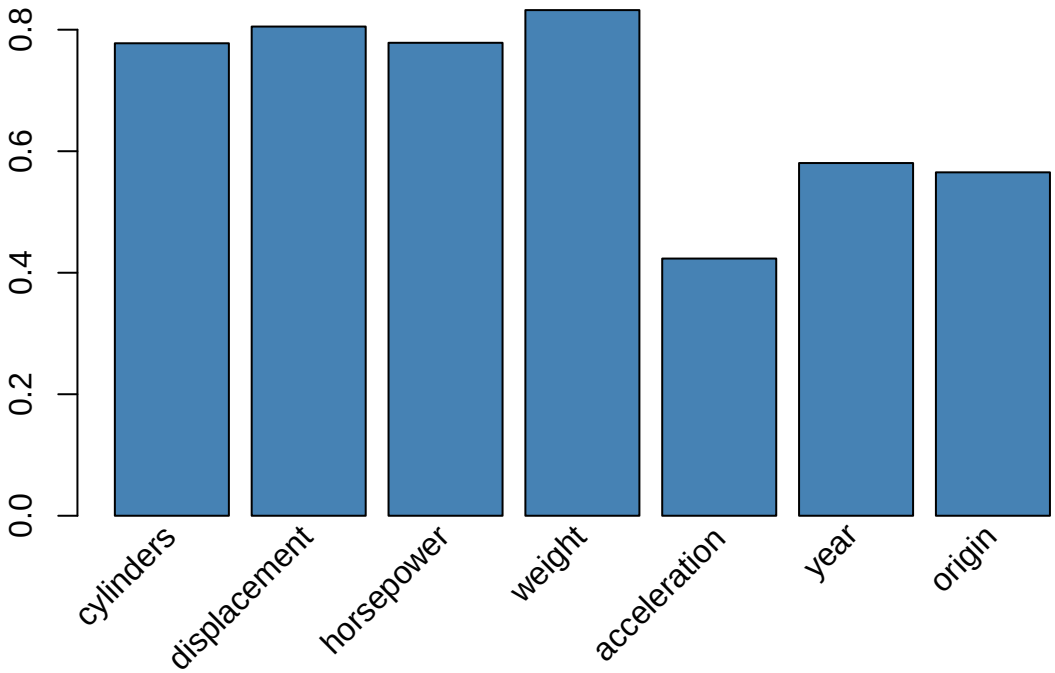

```r
# Plot para mpg-weight
plot(weight, mpg, pch=20, col="steelblue",
     main="Peso")
```



Apartado b

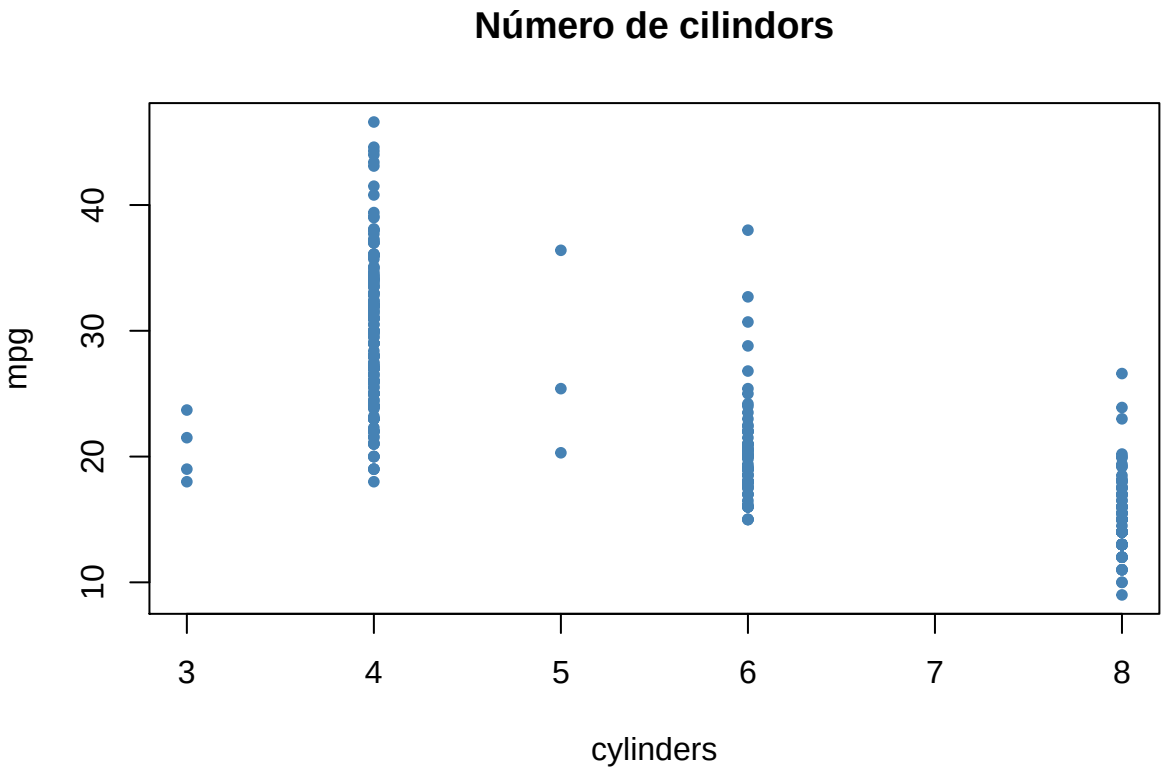
Podemos estudiar de forma numérica la correlación entre `mpg` y las demás variables:

```
# 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:

```
# 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 lineal vamos a usar la función `lm`, que devuelve un modelo lineal `Y~X`, donde `Y` es la variable a predecir y `X` el conjunto de variables predictoras.

Este modelo lineal lo ajustaremos con la muestra de test:

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

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 `Auto.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, 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 11.3924051%. 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	30	7
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.
- Escalamos la muestra de test con esos valores.

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

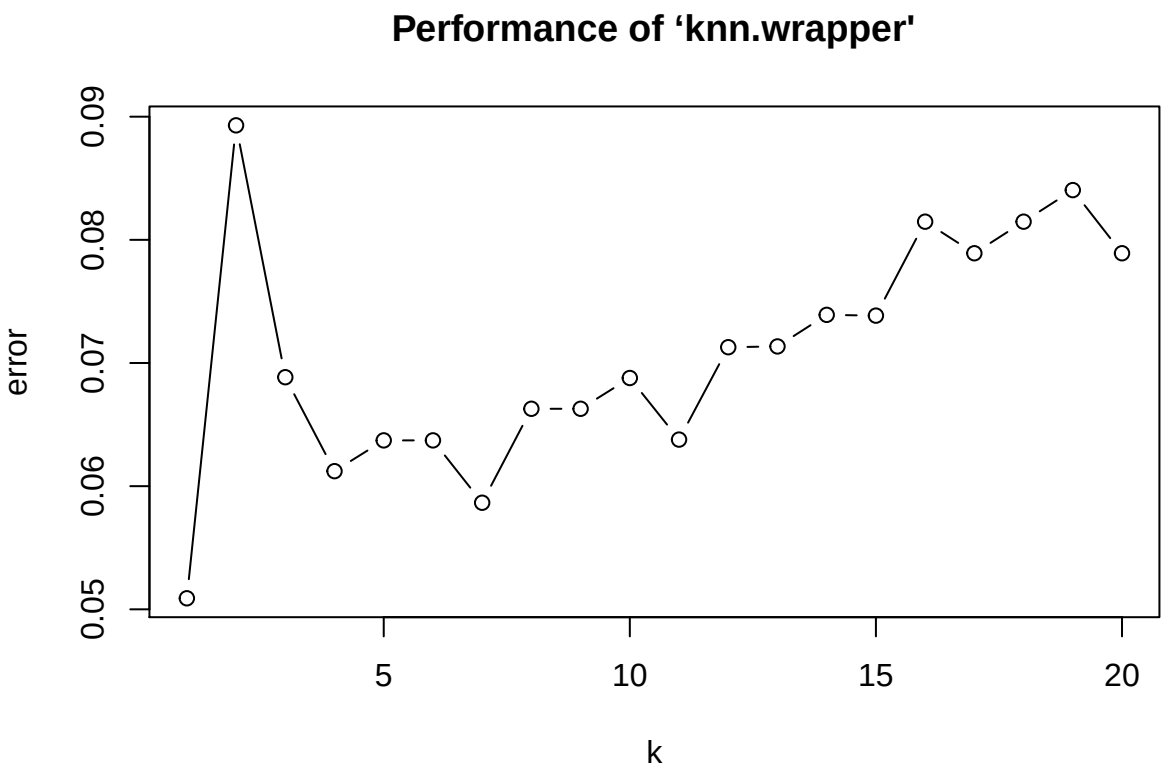
```r
# Escalado de la muestra de entrenamiento sin la variable mpg01
norm.train <- scale(Auto.train[,-1])

# 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 no más atributos que las muestras de entrenamiento y test y las clases:

```r
# 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, Auto.train$mpg01, k=3)

# 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:

```r
# 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 = 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, Auto.train$mpg01, k = knn.k, prob = T)

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

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 == 0, 1-prob, prob)
```

```
  }

  prediction = prediction(probabilities, labels)
  performance = performance(prediction, "tpr", "fpr")
  print(performance)
  plot(performance, ...)
}

curveROC(mod.lin.pred, Auto.test$mpg01, col="darkolivegreen", lwd=1.5, main="Curvas ROC")
```

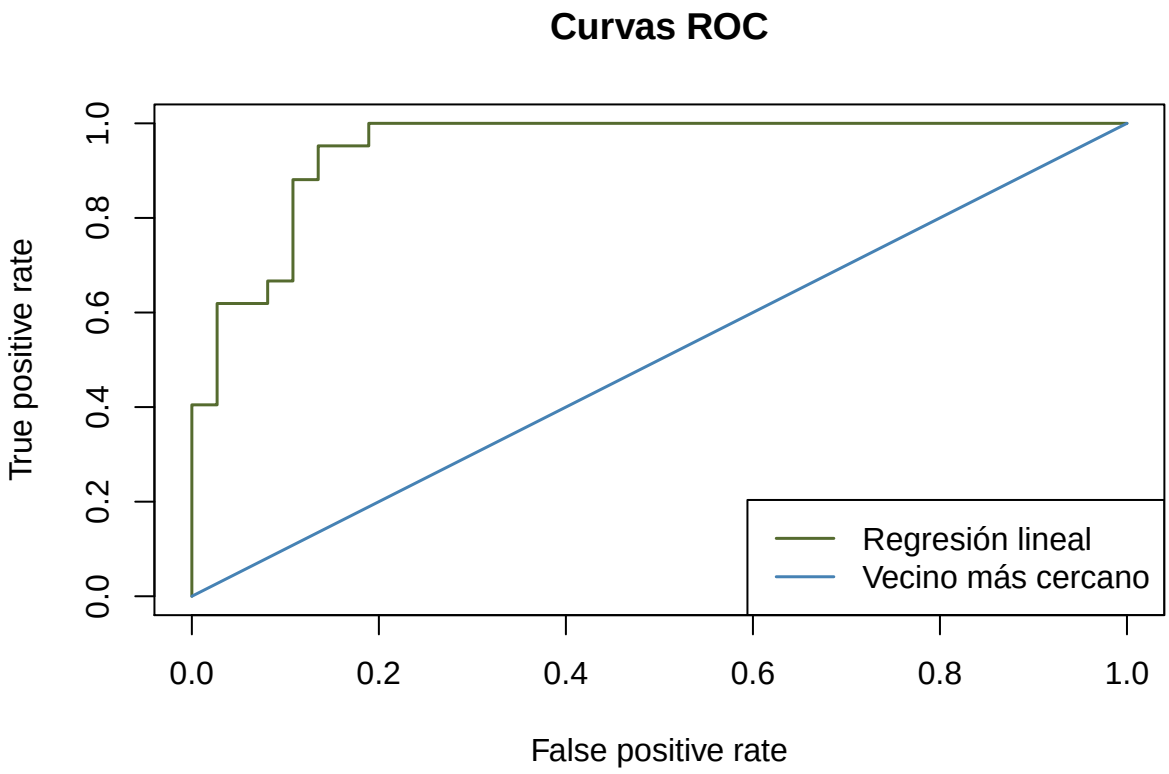
```
## An object of class "performance"
## Slot "x.name":
## [1] "False positive rate"
##
## Slot "y.name":
## [1] "True positive rate"
##
## Slot "alpha.name":
## [1] "Cutoff"
##
## Slot "x.values":
## [[1]]
##  [1] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
##  [7] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
## [13] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
## [19] 0.02702703 0.02702703 0.02702703 0.02702703 0.02702703 0.02702703
## [25] 0.02702703 0.02702703 0.02702703 0.02702703 0.05405405 0.08108108
## [31] 0.08108108 0.08108108 0.10810811 0.10810811 0.10810811 0.10810811
## [37] 0.10810811 0.10810811 0.10810811 0.10810811 0.10810811 0.10810811
## [43] 0.13513514 0.13513514 0.13513514 0.13513514 0.16216216 0.18918919
## [49] 0.18918919 0.18918919 0.21621622 0.24324324 0.27027027 0.29729730
## [55] 0.32432432 0.35135135 0.37837838 0.40540541 0.43243243 0.45945946
## [61] 0.48648649 0.51351351 0.54054054 0.56756757 0.59459459 0.62162162
## [67] 0.64864865 0.67567568 0.70270270 0.72972973 0.75675676 0.78378378
## [73] 0.81081081 0.83783784 0.86486486 0.89189189 0.91891892 0.94594595
## [79] 0.97297297 1.00000000
##
##
## Slot "y.values":
## [[1]]
##  [1] 0.00000000 0.02380952 0.04761905 0.07142857 0.09523810 0.11904762
##  [7] 0.14285714 0.16666667 0.19047619 0.21428571 0.23809524 0.26190476
## [13] 0.28571429 0.30952381 0.33333333 0.35714286 0.38095238 0.40476190
## [19] 0.40476190 0.42857143 0.45238095 0.47619048 0.50000000 0.52380952
## [25] 0.54761905 0.57142857 0.59523810 0.61904762 0.61904762 0.61904762
## [31] 0.64285714 0.66666667 0.66666667 0.69047619 0.71428571 0.73809524
## [37] 0.76190476 0.78571429 0.80952381 0.83333333 0.85714286 0.88095238
## [43] 0.88095238 0.90476190 0.92857143 0.95238095 0.95238095 0.95238095
## [49] 0.97619048 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000
## [55] 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000
## [61] 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000
## [67] 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000
## [73] 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000
## [79] 1.00000000 1.00000000
```

```
##
##
## Slot "alpha.values":
## [[1]]
##  [1]          Inf  0.965949678  0.949852459  0.936415670  0.936203981
##  [6]  0.930216014  0.929795435  0.921367966  0.920484559  0.915132876
## [11]  0.912327529  0.903482882  0.893242267  0.873738240  0.868610284
## [16]  0.865066368  0.859615506  0.859572122  0.848772375  0.848058335
## [21]  0.845057270  0.844204153  0.838376620  0.835818926  0.831787809
## [26]  0.829281371  0.820866463  0.812280827  0.806594200  0.793662404
## [31]  0.782047084  0.769272931  0.719778876  0.717701129  0.715444237
## [36]  0.701712401  0.692266144  0.673197618  0.650648914  0.646127031
## [41]  0.642111282  0.634433661  0.614865660  0.609604598  0.607188117
## [46]  0.599039561  0.566803232  0.518653145  0.497462794  0.452886461
## [51]  0.438368575  0.397360748  0.393885269  0.354728842  0.299779454
## [56]  0.285191538  0.233820755  0.192942687  0.152378774  0.102463074
## [61]  0.092987451  0.042521040  0.025437825 -0.004038042 -0.009979748
## [66] -0.010561074 -0.025094374 -0.046120288 -0.046557899 -0.066960405
## [71] -0.074421796 -0.095670368 -0.124017213 -0.132054551 -0.146747393
## [76] -0.291483938 -0.293188889 -0.316468430 -0.333187325 -0.428701457
```

```
curveROC(knn.pred, Auto.test$mpg01, model.knn=T, add=T, col="steelblue", lwd=1.5)
```

```
## An object of class "performance"
## Slot "x.name":
## [1] "False positive rate"
##
## Slot "y.name":
## [1] "True positive rate"
##
## Slot "alpha.name":
## [1] "Cutoff"
##
## Slot "x.values":
## [[1]]
## [1] 0 1
##
##
## Slot "y.values":
## [[1]]
## [1] 0 1
##
##
## Slot "alpha.values":
## [[1]]
## [1] Inf   1
```

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



Ejercicio 2

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

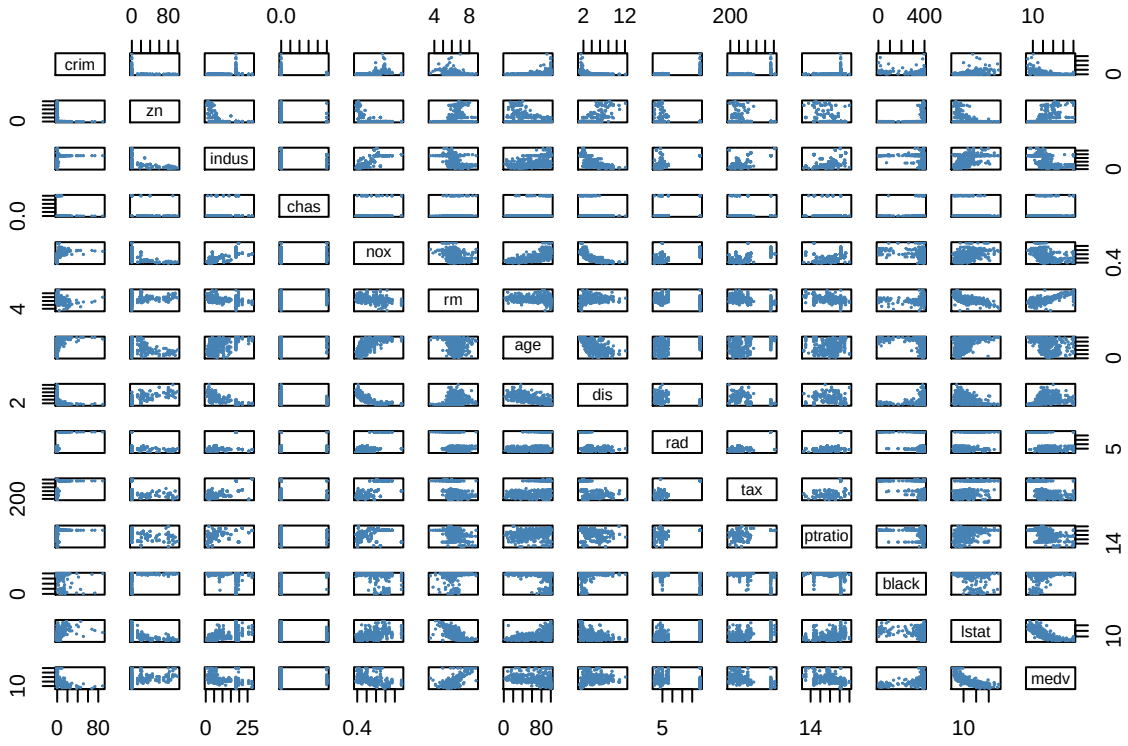
```
# Cargamos la librería necesaria para usar la base de datos Boston
# Para usarla, hay que instalar con la orden
# install.packages('MASS')
library(MASS)

# 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

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

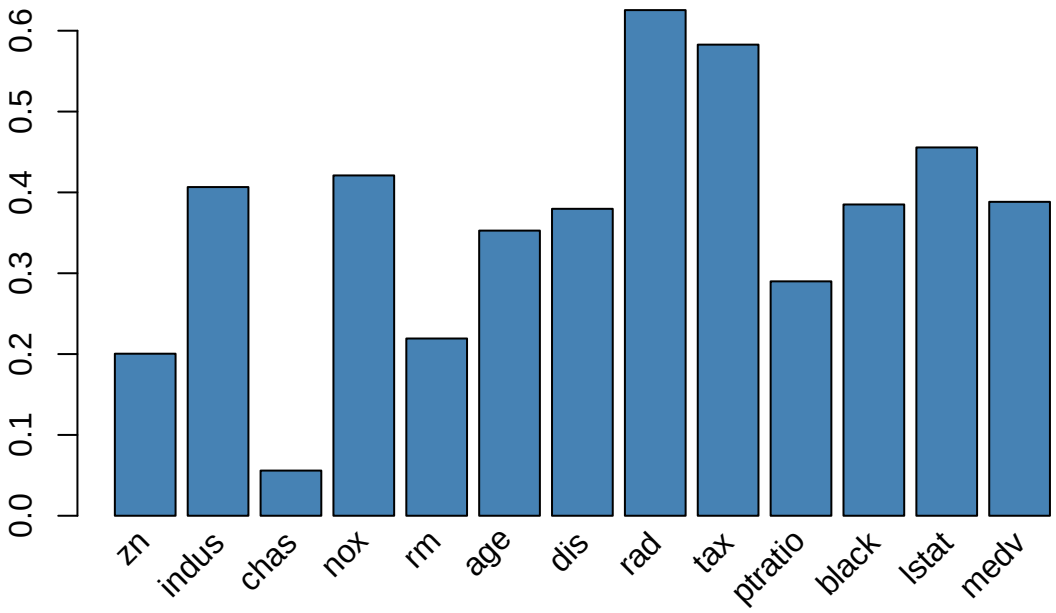
podemos obtener información de la forma que tiene nuestra base de datos. Vemos así que tiene forma de data.frame, con 506 filas y 14 columnas, cuyos nombres son los siguientes: crim, zn, indus, chas, nox, rm, age, dis, rad, tax, ptratio, black, lstat, medv.

```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")

# 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:

```
# 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

```
# 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

# Calculamos ahora los coeficientes solicitados:
```

```r
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,]
lasso.umbral <- 0.4

# 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"])))
}

# 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
```

Las variables seleccionadas por el método LASSO son: zn, indus, chas, nox, rm, dis, rad, ptratio, black, lstat, medv.

Apartado b

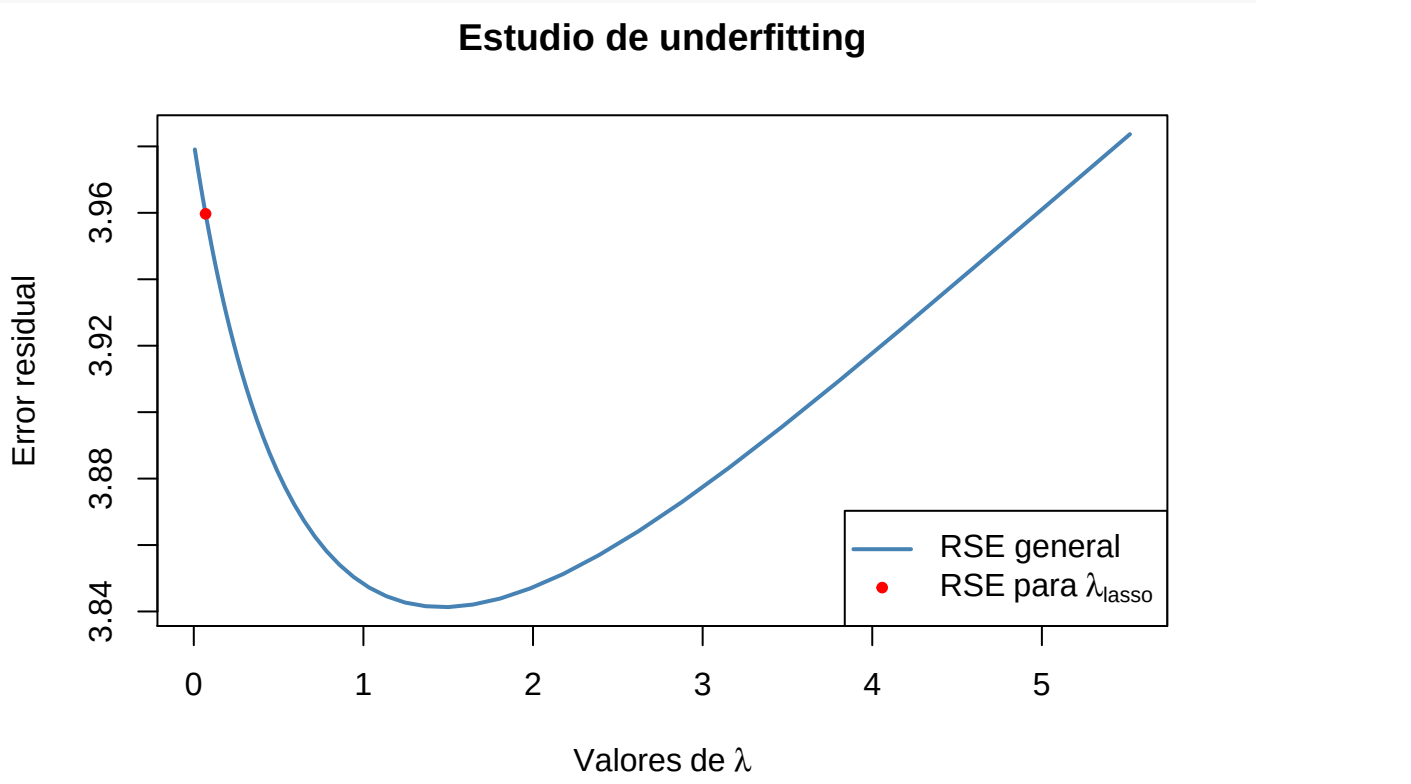
```r
# Vamos a encapsular este estudio en una función dependiente del parámetro
# lambda. Esto nos permitirá después hacer un estudio del underfitting
weight.decay.rse <- function(param.lambda){
  # Hacemos la regresión, apuntando que es weight decay con el parámetro alpha = 0
  ridge <- glmnet(Boston.train[, 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 $\lambda = 0.0696473$ obtenido con el modelo LASSO, La estimación del error residual estándar es de 3.9596933.

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

```
# 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,]
```

```
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"]))
}

svm.linear <- mean(replicate(100,svm.error("linear")))
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 son los siguientes:

- Núcleo lineal: 17.7745098%
- Núcleo polinómico: 16.1176471%
- Núcleo de base radial: 10.9803922%
- Núcleo sigmoidal: 40.254902%