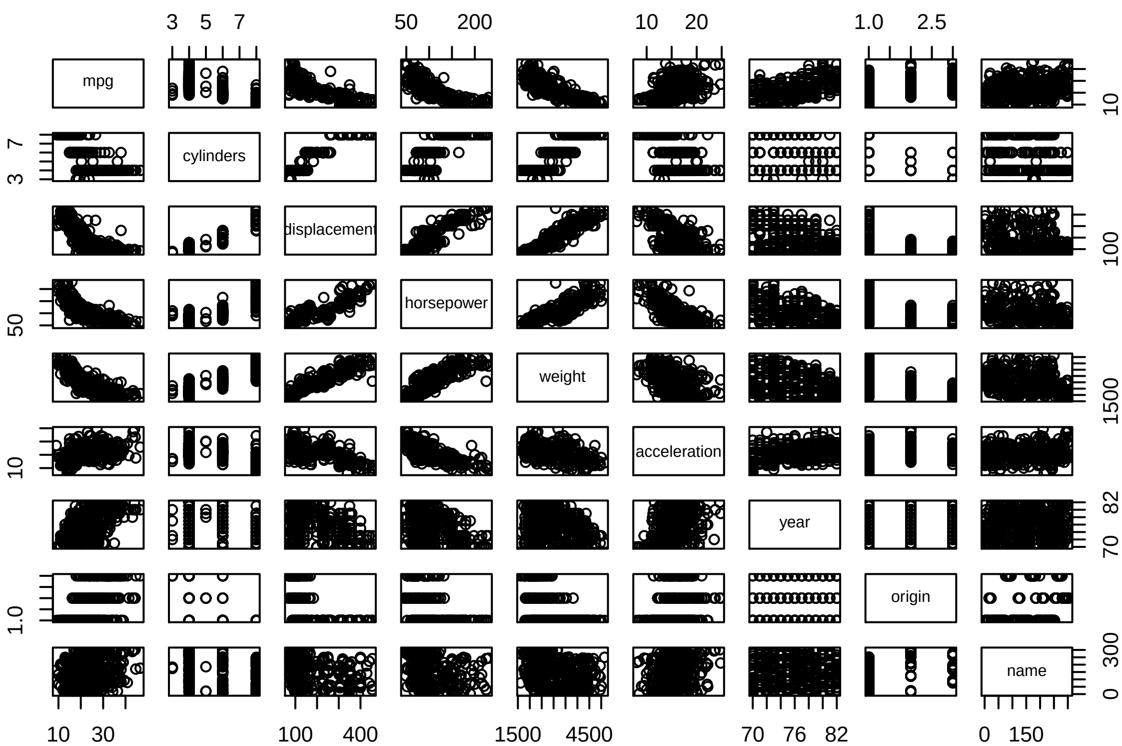
Trabajo3

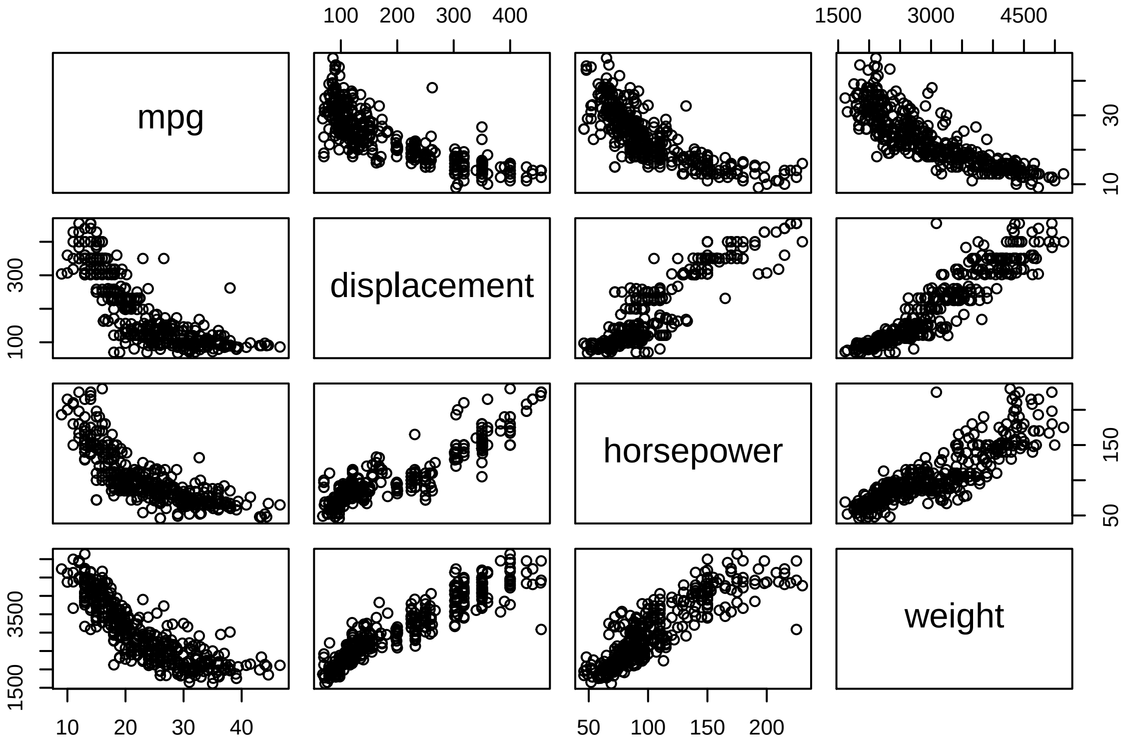
Ejercicio 1

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.

```
set.seed(41192)
attach(Auto)
pairs(Auto)
```

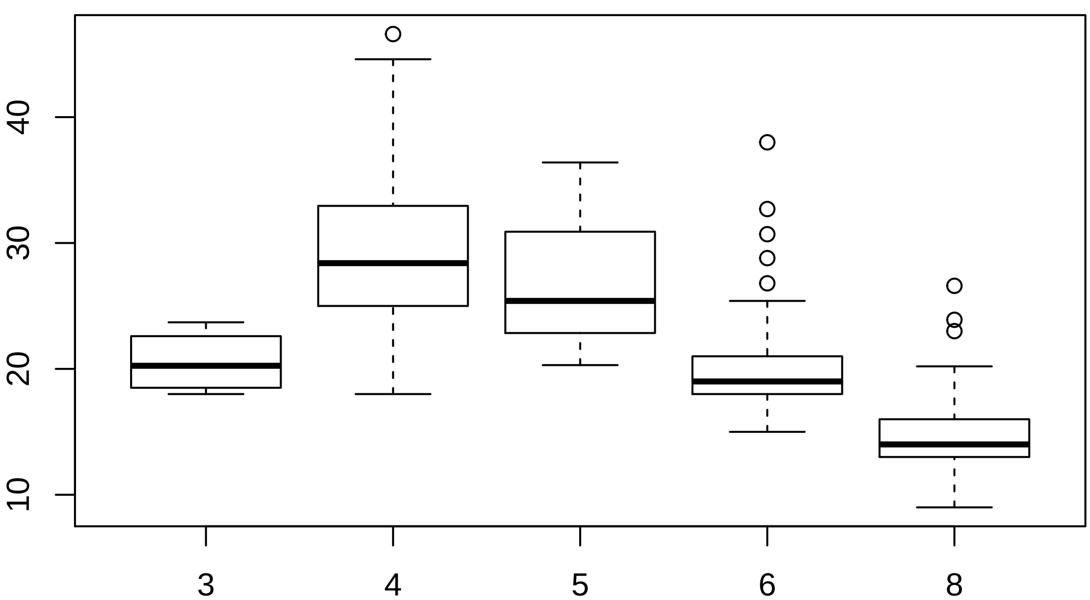



```
pairs(mpg ~ displacement + horsepower + weight)
```

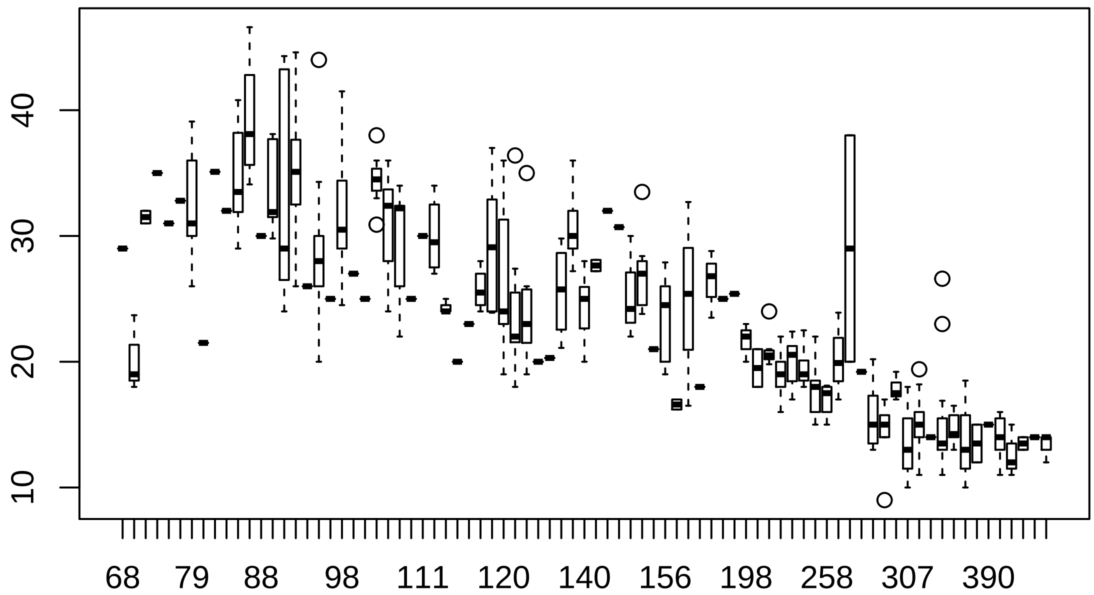


Como podemos ver las "gráficas de dependencias" de mpg con respecto a *displacement*, *horsepower* y *weight* son las gráficas que presentan un patrón más parecido entre ellas indicando que mpg tiene una relación fuerte con estas variables ya que se ajusta a ellas de un modo similar. Por ejemplo si vemos la gráfica con respecto a *acceleration* lo que tenemos es una nube de puntos mucho más dispersa. En cambio estas gráficas sí que tienen un aspecto de ser ajustables linealmente.

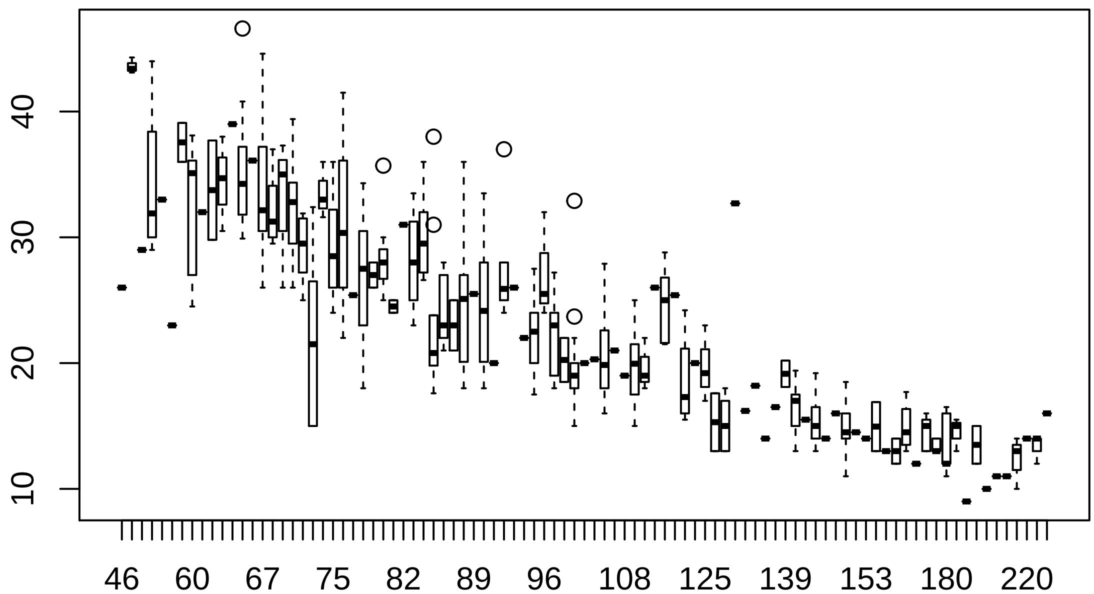
```r
boxplot(mpg ~ cylinders)
```



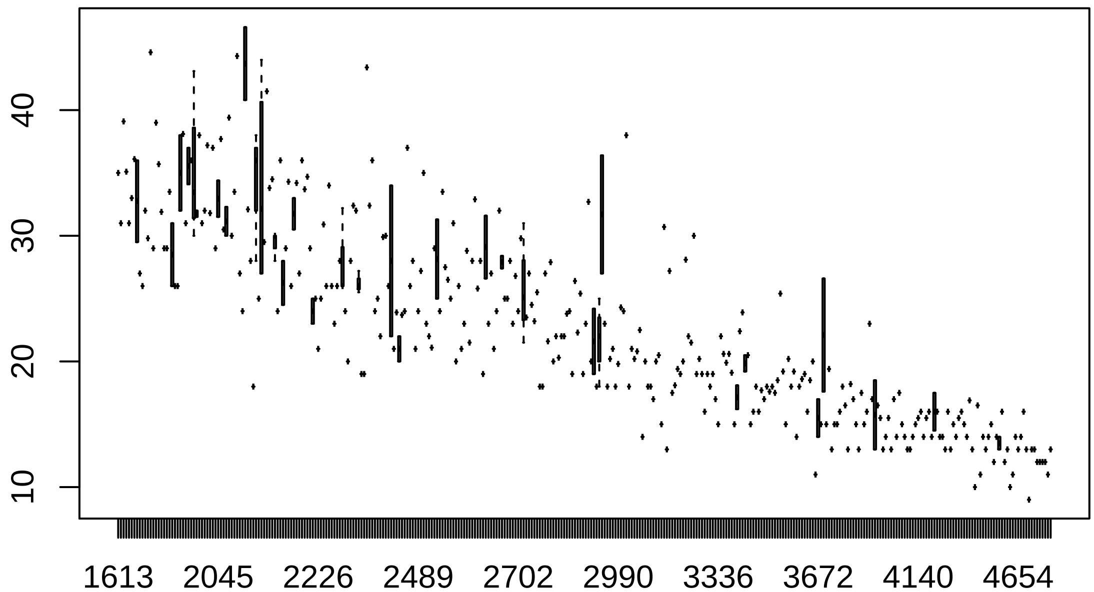

```r
boxplot(mpg ~ displacement)
```



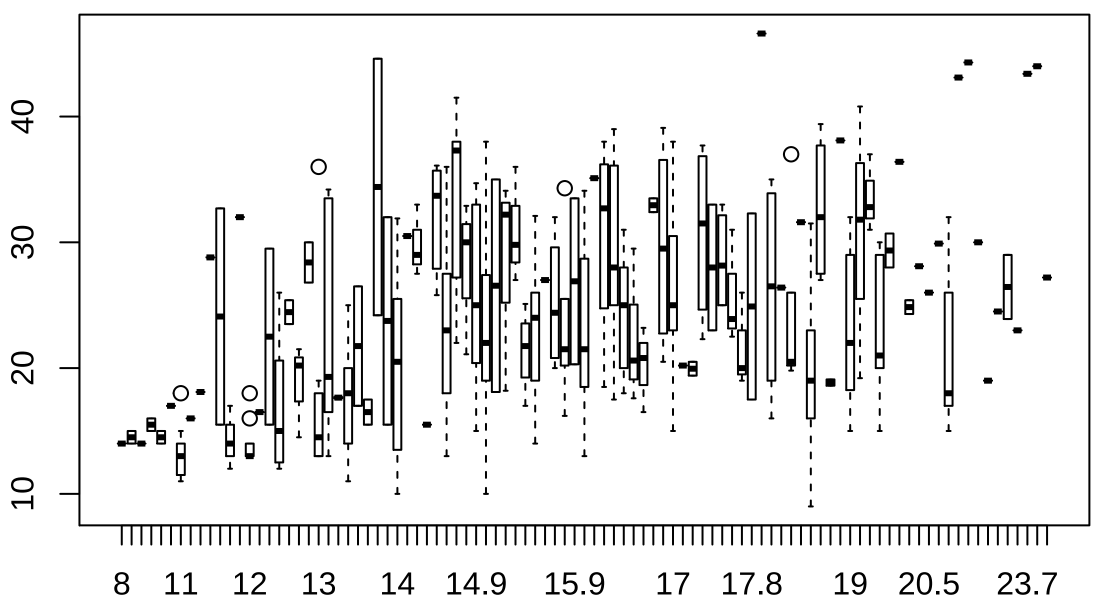

```r
boxplot(mpg ~ horsepower)
```



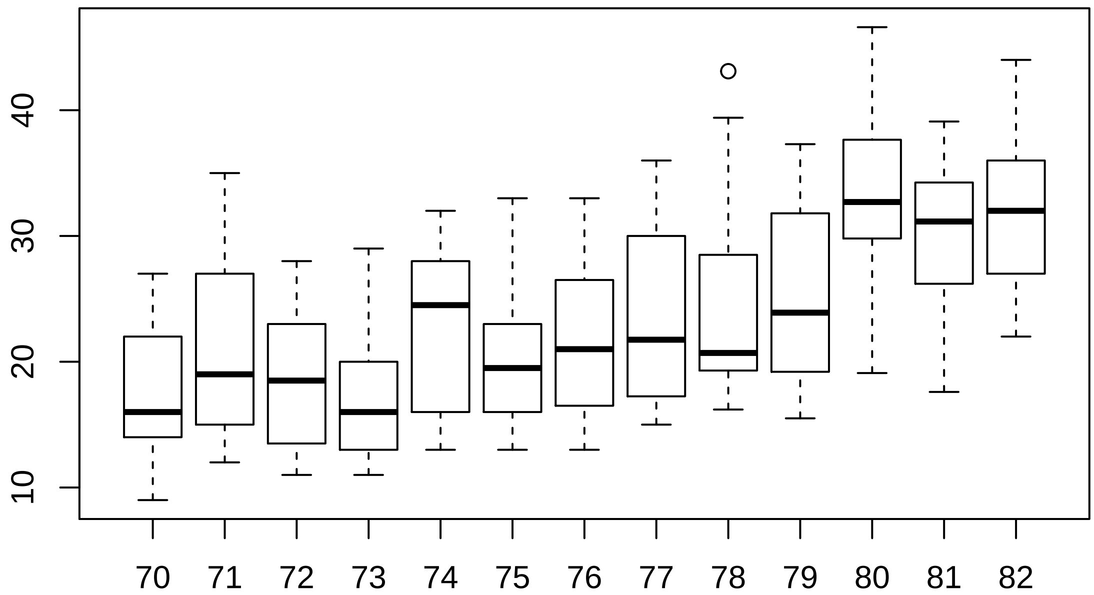

```r
boxplot(mpg ~ weight)
```



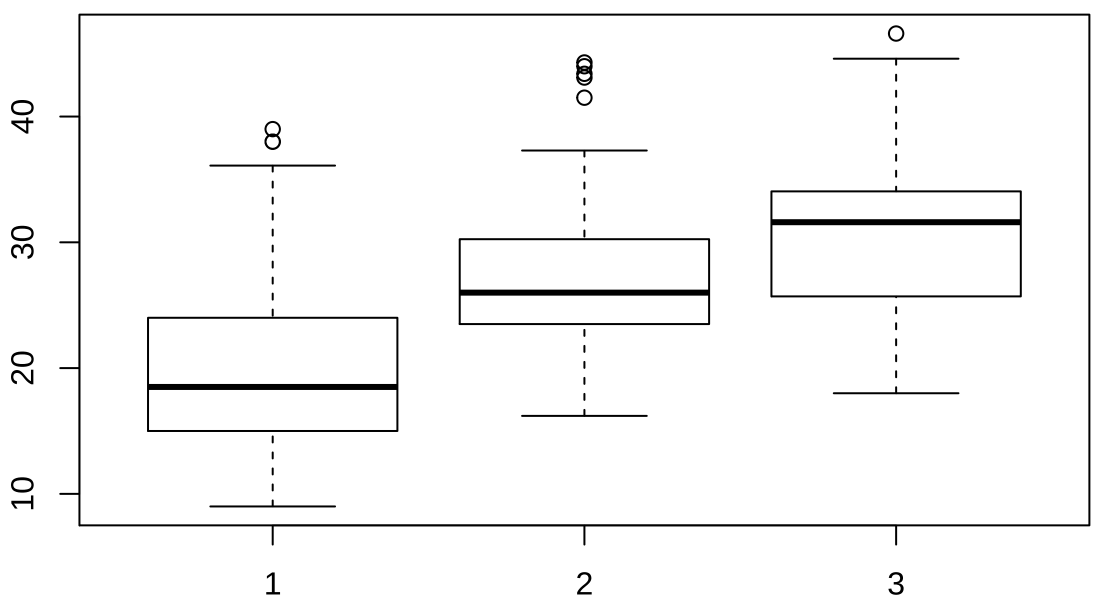

```r
boxplot(mpg ~ acceleration)
```



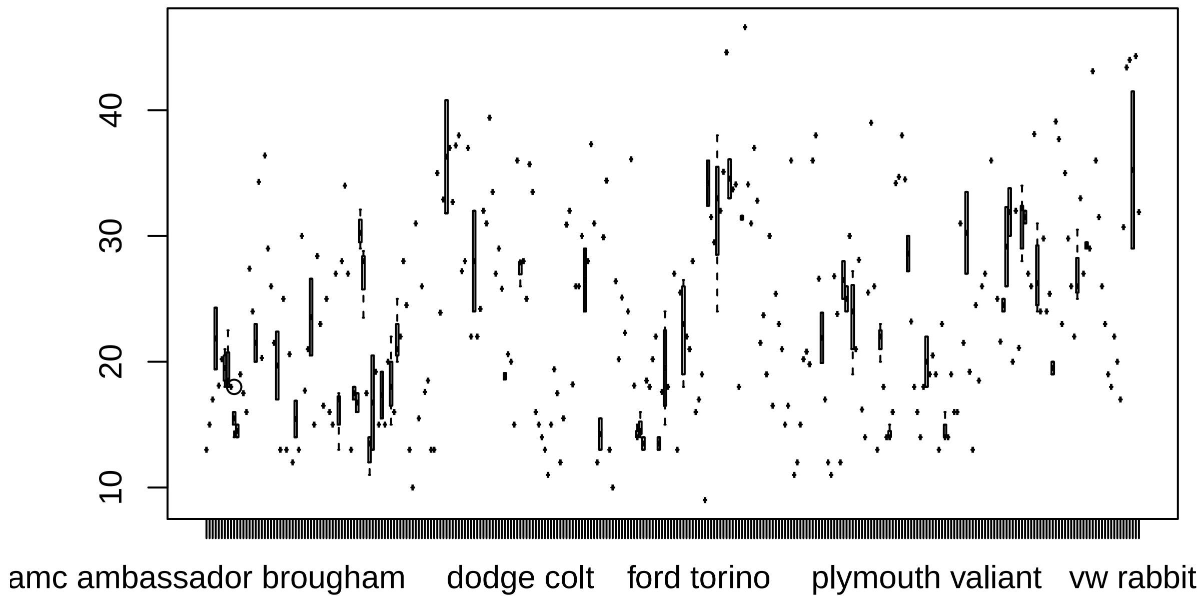

```r
boxplot(mpg ~ year)
```




```r
boxplot(mpg ~ origin)
```




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



Realmente con estas gráficas no observo ninguna información que me permita refinar o cambiar la decisión tomada anteriormente.

b) Seleccionar las variables predictoras que considere más relevantes.

Vamos a seleccionar aquellas que hemos visto en el apartado anterior que parecen seguir un patrón similar:

```
datos = Auto[,c("mpg","displacement", "horsepower", "weight")]
```

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

A priori había pensado en calcular en primer lugar la variable mpg1 del siguiente apartado para así poder realizar un particionamiento más homogéneo de las etiquetas positivas y negativas en los conjuntos de entrenamiento y test. El problema es que para hacer tal cosa tendría que calcular la mediana de todos los datos de Auto y lo que queremos es calcular la mediana, y por tanto el valor de mpg1, sólo en base a los datos de entrenamiento puesto que se dijo en teoría que para no contaminar el aprendizaje no podíamos usar los datos de test para calcular una mediana, sería como mirar los datos antes de aprender. Entonces he realizado simplemente un submuestreo de los datos aleatorio para dividirlos en entrenamiento y test.

```
n = nrow(datos)
idx_train = sample(seq(n), ceiling(0.8*n))

datos.train = datos[idx_train,]
datos.test = datos[-idx_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 Auto).

En lugar de los valores -1 y 1 que se dicen el enunciado lo que hacemos es darle los valores 0 y 1 ya quer esto es más natural para regresión logística (la sigmoide va de 0 a 1) y la profesora me sugirió tal cosa.

```
mediana = median(datos.train$mpg)
mpg1.train = sapply(datos.train$mpg, function(x) if (x < mediana) return(0) else return(1))
mpg1.test = sapply(datos.test$mpg, function(x) if (x < mediana) return(0) else return(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.

```
#elaboramos los dataframes añadiendo la variable a predecir
trainRL = data.frame(mpg01 = mpg1.train, datos.train)
testRL = data.frame(mpg01 = mpg1.test, datos.test)

#ajustamos un modelo de regresión logística a los datos de train
RL = glm(mpg01 ~ displacement+horsepower+weight, data = trainRL, family = binomial)

#predecimos las respuestas para test y según la probabilidad obtenida damos un valor u otro
prediccion = predict(RL, newdata = testRL, type = "response")
RL.pred = rep(0, length(testRL$mpg01))
RL.pred[prediccion > .5] = 1

#calculamos el porcentaje de error
cat("El % de errores en test con RL es: ", sum(testRL$mpg01 != RL.pred)/length(testRL$mpg01)*100)
```

```
## El % de errores en test con RL es:  15.38462
```

Obtenemos un error del 15% que si bien no es demasiado alto puede deberse a la incertidumbre implícita en la clasificación que da la regresión logística, ya que esta nos permite trabajar con eso, expresar una cierta incertidumbre en las decisiones.

- 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 del modelo? ¿Cuál es el valor de K que mejor ajusta los datos?

Lo que vamos a hacer en primer lugar es normalizar los datos de modo que a la hora de calcular las distancias, que es como funciona KNN, una característica no tenga a priori más peso que otras.

Observemos que para normalizar los datos de test empleamos aquellos escalados que se hacen sobre los de train, de modo que para calcular el escalado de train no usemos los datos de test y así no contaminar el proceso.

Hemos elaborado una función para que sea más cómodo calcular este error de clasificación en futuros apartados:

```r
getErrorKNN <- function(datos.train, datos.test, et.train, et.test){
  #normalizamos los datos
  train.norm = scale(datos.train[,c("weight","displacement","horsepower")])
  medias = attr(train.norm, "scaled:center")
  escalados = attr(train.norm, "scaled:scale")
  test.norm = scale(datos.test[,c("weight","displacement","horsepower")], medias, escalados)

  #agrupamos ambos dataframes en uno para el tune
  datos.full = rbind(train.norm, test.norm)
  et.full = as.factor(c(et.train,et.test))

  #obtenemos el mejor k para nuestros datos
  set.seed(75570417)
  mknns = tune.knn(datos.full, et.full, k=1:20, tunecontrol = tune.control(sampling = "cross"), cross =

  mejor_k = mknns$best.model$k

  cat("Se ha elegido como mejor k el: ", mejor_k, "\n")

  #predecimos las etiquetas para test
  KNN = knn(train.norm, test.norm, et.train, k = mejor_k, prob = TRUE)

  #calcuamos el error
  err.test = sum(et.test != KNN)/length(et.test)*100
  cat("El % de errores que obtenemos en el test es: ", err.test, "\n")
  list(KNN, err.test)
}

KNN =  getErrorKNN(trainRL, testRL, mpg1.train, mpg1.test)[[1]]
```

```
## Se ha elegido como mejor k el:  7
## El % de errores que obtenemos en el test es:  16.66667
```

Como podemos ver el error es algo más alto, un 1%, que el que obtenemos con regresión logística, aquí no hablamos tanto de incertidumbre sino de cómo se distribuyen "espacialmente" las muestras, lo que puede dar lugar a que los vecinos más cercanos a una muestra determinada no sean, la mayoría, de la clase real a la que pertenece la muestra a la que queremos etiquetar. El KNN es un algoritmo que depende fuertemente de la distribución de las muestras de entrenamiento.

- Pintar las curvas ROC (instalar paquete ROCR en R) y comparar y valorar los resultados obtenidos por ambos modelos.

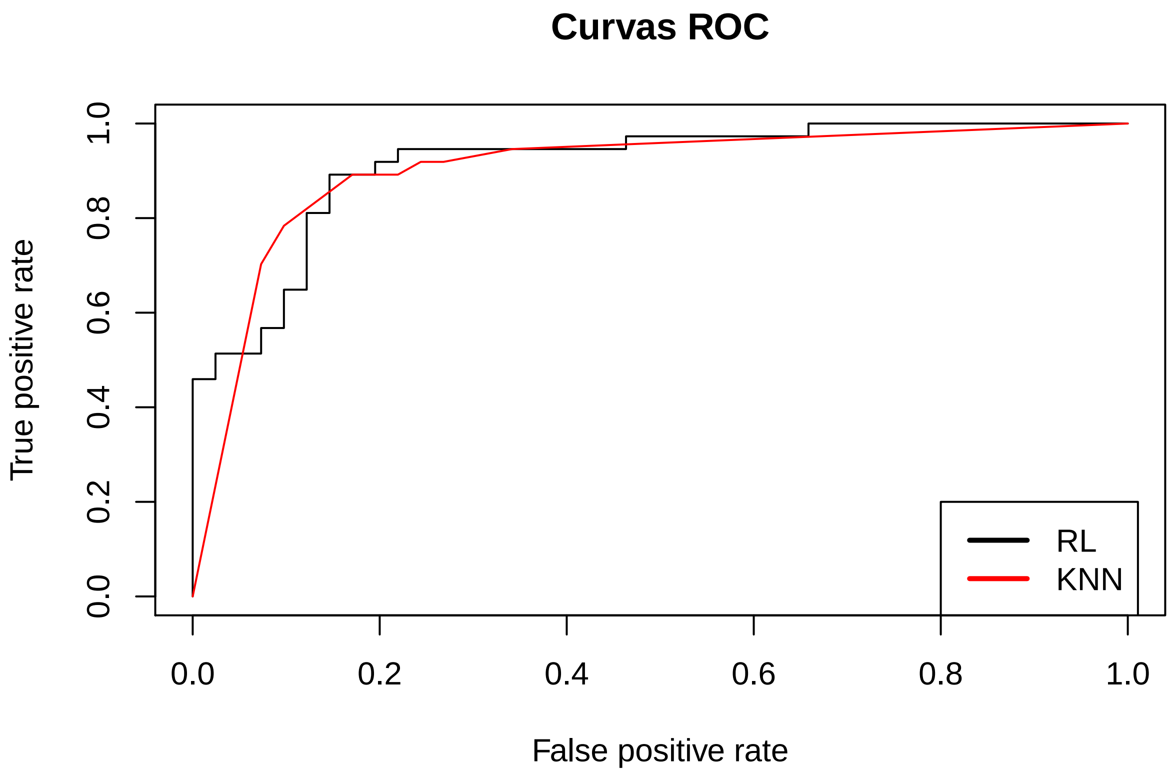
Vamos a emplear las probabilidades que nos devuelven ambos métodos (por esto en knn tenemos *prob* a true y en la predicción de regresión logística ponemos *type* como responses) y observemos que para aquellos que el KNN ha etiquetado como 0 entonces le damos la vuelta a la probabilidad para obtener la curva ROC correcta.

```r
pRL = prediction(prediccion, testRL$mpg01)
perf = performance(pRL, "tpr", "fpr")
plot(perf, main = "Curvas ROC")

prob = attr(KNN, "prob")
prob = ifelse(KNN == "0", 1-prob, prob)
```

```r
pKNN = prediction(prob, testRL$mpg01)
perf = performance(pKNN, "tpr", "fpr")


plot(perf,  add = TRUE, col = "red")
legend(0.8,0.2,c("RL", "KNN"), lty=c(1,1),lwd=c(2.5,2.5),col=c("black","red"))
```



Al inicio RL parece mas estable puesto que pasa más tiempo teniendo un tasa menor de falso positivos, ahora bien cuando esta tasa aumenta vemos como antes la misma pérdida o el mismo error de clasificación es el KNN el que acierta un mayor número de veces, es decir, que aunque cometemos el mismo error al menos clasificamos bien más muestras.

Cuando la tasa de falsos positivos aumenta es nuevamente la RL la que mejor se comporta puesto que tiene una tasa de verdaderos positivos más elevada aunque no difiere en gran medidad de la del KNN. Con lo cual en mi opinión es el KNN el que tiene un mejor comportamiento puesto que es el que da mejores resultados ante un error igual.

e) (Bonus 1) Estimar el error de test de ambos modelos pero usando Validación Cruzada de 5-particiones. Comparar con los resulados obtenidos en el punto anterior.

Lo primero que hacemos es calcular la variable a predecir mpg01, en esta ocasión y dado que vamos a emplear validación cruzada lo que haremos es calcular la mediana con todos los datos en lugar de sólo con los de training, que cambiarán de un fold a otro.

```r
mediana = median(datos$mpg)
mpg01 = sapply(datos$mpg, function(x) if (x < mediana) return(0) else return(1))
fullData = data.frame(mpg01 = mpg01, datos)
```

```
RL = glm(mpg01 ~ displacement+horsepower+weight, data = fullData, family = binomial)
pp = cv.glm(fullData, RL, K = 5)

cat("El error de test obtenido haciendo VC en RL ha sido: ", pp$delta[1]*100, "\n")
```

```
## El error de test obtenido haciendo VC en RL ha sido:  8.223104
```

```
#hacemos las particiones a mano
folds = kfold(fullData, k = 5)
err = 0

#obtendremos el error de KNN en cada fold y haremos el promedio
for (i in seq(5)) {
  err = err + getErrorKNN(fullData[folds != i,], fullData[folds == i,], fullData[folds != i,]$mpg01, ful
}
```

```
## Se ha elegido como mejor k el:  8
## El % de errores que obtenemos en el test es:  7.692308
## Se ha elegido como mejor k el:  10
## El % de errores que obtenemos en el test es:  6.329114
## Se ha elegido como mejor k el:  7
## El % de errores que obtenemos en el test es:  8.974359
## Se ha elegido como mejor k el:  8
## El % de errores que obtenemos en el test es:  8.860759
## Se ha elegido como mejor k el:  7
## El % de errores que obtenemos en el test es:  7.692308
```

```
cat("El error de test obtenido haciendo VC en KNN ha sido: ", err/5)
```

```
## El error de test obtenido haciendo VC en KNN ha sido:  7.90977
```

Como podemos ver el error obtenido es menor que en los puntos anteriores, el tamaño de las particiones de training y test es el mismo con lo que esto no influye en la diferencia. Si bien es cierto que ahora nuestra variable mpg01 se hace con todos los datos disponibles en Auto por lo tanto se tiene más información con lo que es lógico que el ajuste sea mejor.

f) (Bonus 2) Ajustar el mejor modelo de regresión posible considerando la variables mpg como salida y el resto como predictoras. Justificar el modelo ajustando en base al patrón de los residuos. Estimar su error de entrenamiento y de test.

En primer lugar vamos a ver si hay algunar correlación entre las distintas variables que podemos emplear, para ello vamos a ajustar un modelo donde usamos las tres variables de forma independiente y otros tres donde las usamos correladas dos a dos, viendo cuál nos da mejor tasa de error, el error que vamos a medir, dado que estamos hablando de regresión será el error cuadrático medio, MSE:

```
R1 = lm(mpg ~ displacement + horsepower + weight, data = datos.train)
prediccion = predict(R1, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  17.47243
```

```r
R2 = lm(mpg ~ displacement*weight, data = datos.train)
prediccion = predict(R2, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  15.67
```

```r
R3 = lm(mpg ~ horsepower*displacement, data = datos.train)
prediccion = predict(R3, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  14.48405
```

```r
R4 = lm(mpg ~ horsepower*weight, data = datos.train)
prediccion = predict(R4, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  13.85424
```

Como podemos ver obtenemos mejor error cuando usamos las variables *horsepower* y *weight* correladas con lo que es lógico pensar que estas variables están estrechamente relacionadas y en consecuencia es mejor emplear un modelo donde esto se vea reflejado en la fórmula.

Ahora bien tenemos una variable que no estamos usando, veamos qué ocurre si la incorporamos al modelo, la vamos a incorporar de dos modos, en primer lugar tal y como está y en segundo lugar por medio de un polinomio de grado 2:

```r
R5 = lm(mpg ~ horsepower*weight + displacement, data = datos.train)
prediccion = predict(R5, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  14.02451
```

```r
R6 = lm(mpg ~ horsepower*weight + poly(displacement,2), data = datos.train)
prediccion = predict(R6, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  13.6401
```

Como podemos ver el error es mejor cuando la incorporamos mediante un polinomio cuadrático, realmente ya habíamos hecho experimentos anteriormente y habíamos visto por medio del plot que facilita lm que los residuos parecían seguir una polinomial con lo que habíamos probado con usar todas las variables como polinomios como veremos más adelante. Con lo cual es un buen intento el usar un polinomio cuadrático como hemos visto.

```r
R7 = lm(mpg ~ horsepower*weight + poly(displacement,5), data = datos.train)
prediccion = predict(R7, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  14.89589
```

```r
R8 = lm(mpg ~ poly(horsepower,2)+ poly(weight,2) + poly(displacement,2), data = datos.train)
prediccion = predict(R8, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  13.8842
```

```r
R9 = lm(mpg ~ poly(horsepower,8)+ poly(weight,8) + poly(displacement,8), data = datos.train)
prediccion = predict(R9, newdata = datos.test, type = "response")
cat("El MSE en test con regresión es: ", mean((prediccion - datos.test$mpg)^2) )
```

```
## El MSE en test con regresión es:  15.06453
```

Aquí vemos otros experimentos donde observamos claramente el sobreajuste pues al aumentar el orden del polinomio en *displacement* a 5 así como usando todas las variables por medio de polinomio de grado 8 vemos como el error aumenta (en este ultimo caso con respecto a usar todas las variables con respecto a polinomios de grado 2). Finalmente si comparamos el modelo con correlación con el modelo con todas las variables de forma cuadrática e independiente vemos que el error es muy similar, siendo ligéramente mejor el que tiene las variables correladas con lo que nos quedaremos con este último.

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.

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 a un umbral prefijado.

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 conportamiento de los residuos muestran algún indicio de "underfitting".

Lo que vamos a hacer en primer lugar es emplear validación cruzada para obtener el mejor *s* para nuestros datos, este s lo que hace es regular el impacto de la restricción de "contracción", la cual lleva a coeficientes más pequeños cuanto mayor sea su impacto, es decir, el valor de s.

Con el objetivo de tener un buen s lo que vamos a hacer es obtenerlo mediante validación cruzada en la cual emplearemos todos los datos, ya vimos en teoria que con validación cruzada si empleamos todos los datos podemos tener un buen estimador sin riesgo de que usar los datos de test suponga un problema de sobreajuste.

```r
attach(Boston)

set.seed(41192)
cv.out = cv.glmnet(as.matrix(Boston[,-1]), Boston[,1],alpha=1)
bestlam = cv.out$lambda.min #el mejor ese obtenido
```

Ahora aplicamos un modelo lasso a nuestros datos empleando el s que hemos obtenido, esto nos dará una serie de coeficientes para las distintas características de la muestra, aquellos que, en valor absoluto, estén por encima de un cierto umbral (que iremos modificando hasta obtener un buen resultado) serán los de las variables que seleccionemos para el siguiente apartado.

```
#generamos un submuestreo aleatorio 80-20
set.seed(41192)
train = sample(1:nrow(Boston), nrow(Boston)*0.8)
test = (-train)

out = glmnet(as.matrix(Boston[train,-1]),Boston[train,1], alpha = 1)
lasso.coef = predict(out, type="coefficients",s=bestlam)

umbral = 0.1
var_seleccionadas = which(abs(lasso.coef) > umbral)[-1]

#ajustamos el weight decay con las variables que han sido seleccionadas
ridge.mod = glmnet(as.matrix(Boston[train, var_seleccionadas]), Boston[train,1], alpha = 0)
ridge.pred = predict(ridge.mod, s=bestlam,newx = as.matrix(Boston[test, var_seleccionadas]))

error = sqrt(mean((Boston[test,1] - ridge.pred)^2)/(nrow(Boston[test,])-2))
print(error)
```

```
## [1] 0.3129668
```

El error que calculamos es el RSE que es la raíz cuadrada del error cuadrático medio dividido por el número de muestras menos dos, a continuación mostramos una tabla con los distintos parámetros que hemos probado. Los parámetros que hemos ido modificando han sido el umbral que marca la selección de las características y el s para el weight decay, ya que aunque tenemos uno óptimo para el Lasso no tiene por qué ser el mismo para esta nueva regresión, aunque veremos que sí. En la siguiente tabla recogemos los parámetros junto con el RSE obtenido con ellos (para la partición 80-20 que hacemos de los datos con la semilla seleccionada):

umbral	s	RSE
0.1	bestlam	0.3129668
0.1	100	0.4107367
0.1	200000bestlam	0.4997857
0	bestlam	0.3206541
0.45	bestlam	0.3429374

Como vemos el bestlam es el que mejores resultados da con el mismo umbral, también hemos obsevado que si dividimos el bestlam por prácticamente cualquier número el error no varía, no obstante y dado que veo que lo que marca el s es el peso de una condición que podríamos llamar de regularización opto por dejarlo lo "más grande posible", para evitar en la medida de lo posible sobreajuste. Cuando amplio demasiado el s, dándole mucho peso a la regularización entonces el algoritmo se olvida de ajustar los datos y da peores errores.

Por otro lado si para el umbral hacemos que se consideren todas las variables el error empeora aunque no demasiado, indicando que no hay tanto ruido que afecte al algoritmo como podríamos pensar. En cambio lo que sí que empeora más el resultado es ser demasiado elitistas con la selección de variables ya que según vemos estamos perdiendo información para un buen ajuste.

Por tanto en mi opinión observamos underfitting cuando le damos demasiado peso a la condición de regularización, cuando el s es demasiado grande, para los otros parámetros el error fuera de la muestra es relativamente bueno.

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 el resultado del uso de distintos núcleos.

A continuación vamos a aplicar modelos de SVM a los datos de entrenamiento. El principal problema que tenemos es que nuestros datos no tienen sólo dos propiedades con lo cual el hecho de hacer un plot de la clasificación dada (como se hace en el libro) no nos dirá qué tipo de núcleo usar. Vamos entonces a hacer 1000 submuestreos distintos de los datos y a obtener la media del error promedio de clasificación para ver qué kernel es el que mejor funciona:

```r
set.seed(41192)

errL = 0
errS = 0
errP = 0
errR = 0

Nexp = 1000
for(i in seq(Nexp)) {
  train = sample(1:nrow(Boston), nrow(Boston)*0.8)
  test = (-train)

  mediana = median(Boston[train,1])
  crim1.train = sapply(Boston[train,1], function(x) if (x < mediana) return(-1) else return(1))
  crim1.test = sapply(Boston[test,1], function(x) if (x < mediana) return(-1) else return(1))


  data.train = data.frame(Boston[train,-1], crim1 = as.factor(crim1.train))
  data.test = data.frame(Boston[test, -1], crim1 = as.factor(crim1.test))

  svmfitL = svm(crim1~., data=data.train, kernel = "linear")
  svm.pred = predict(svmfitL, newdata = data.test)
  errL = errL + mean(svm.pred != data.test$crim1)

  svmfitS = svm(crim1~., data=data.train, kernel = "sigmoid")
  svm.pred = predict(svmfitS, newdata = data.test)
  errS = errS + mean(svm.pred != data.test$crim1)

  svmfitR = svm(crim1~., data=data.train, kernel = "radial")
  svm.pred = predict(svmfitR, newdata = data.test)
  errR = errR + mean(svm.pred != data.test$crim1)

  svmfitP = svm(crim1~., data=data.train, kernel = "polynomial")
  svm.pred = predict(svmfitP, newdata = data.test)
  errP = errP + mean(svm.pred != data.test$crim1)
}

cat("Obtenemos en promedio un error medio con kernel lineal de:", errL/Nexp, "\n")
```

```
## Obtenemos en promedio un error medio con kernel lineal de: 0.1122353
```

```r
cat("Obtenemos en promedio un error medio con kernel sigmoidal de:", errS/Nexp, "\n")
```

```
## Obtenemos en promedio un error medio con kernel sigmoidal de: 0.1964216
```

```r
cat("Obtenemos en promedio un error medio con kernel radial de:", errR/Nexp, "\n")
```

```
## Obtenemos en promedio un error medio con kernel radial de: 0.1156078
```

```r
cat("Obtenemos en promedio un error medio con kernel polinomial de:", errP/Nexp, "\n")
```

```
## Obtenemos en promedio un error medio con kernel polinomial de: 0.141598
```

Por lo tanto el que mejor resultados da es el clasificador con un núcleo lineal, seguido muy de cerca por el radial, pero como un clasificador lineal es lo más simple que podemos utilizar pues nos decantaremos por este. Al ser un modelo simple es menos sensible al sobreajuste por lo tanto si los datos de entrenamiento se han podido separar bien mediante él es lógico que su error de test (generalización) sea bueno.

Como vemos consideramos todas las variables excepto la variable crim ya que esta es la que da la que queremos predecir, con lo que estaríamos contaminando el aprendizaje. Indicar que como comentó la profesora no tenemos que usar solamente aquellas variables seleccionadas en apartados anteriores del ejercicio, sino que consideramos todas. Esto se debe a que según qué modelo de aprendizaje estemos empleando unas variables tendrán más importancia que otras; cada modelo tiene sus criterios y "preferencias", entonces a priori, sin un análisis más exhaustivo de los datos, no podemos descartar ninguna de las características.

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

Vamos a estimar el error de entrenamiento y test tanto para regresión con weight decay como para el SVM con núcleo lineal.

```r
set.seed(41192)
mediana = median(Boston$crim)
crim1 = sapply(Boston$crim, function(x) if (x < mediana) return(-1) else return(1))
data.full = data.frame(Boston[,-1], crim1 = as.factor(crim1))

folds = kfold(Boston, k = 5)


#Para regresión con weight decay
err.in = 0
err.out = 0

for (i in seq(5)) {
  out = glmnet(as.matrix(Boston[folds != i,-1]),Boston[folds != i,1], alpha = 1)
  lasso.coef = predict(out, type="coefficients",s=bestlam)

  umbral = 0.1
  var_seleccionadas = which(abs(lasso.coef) > umbral)[-1]

  ridge.mod = glmnet(as.matrix(Boston[folds != i, var_seleccionadas]), Boston[folds != i,1], alpha = 0)
```

```r
  ridge.pred.train = predict(ridge.mod, s=bestlam,newx = as.matrix(Boston[folds != i, var_seleccionadas]
  ridge.pred.test = predict(ridge.mod, s=bestlam,newx = as.matrix(Boston[folds == i, var_seleccionadas]

  err.in = err.in + mean((Boston[folds != i,1] - ridge.pred.train)^2)
  err.out = err.out + mean((Boston[folds == i,1] - ridge.pred.test)^2)
}

cat("El MSE de train estimado por CV para regresión es: ", err.in/5, ".\n")
```

```
## El MSE de train estimado por CV para regresión es:  41.50698 .
```

```r
cat("El MSE de test estimado por CV para regresión es: ", err.out/5, ".\n")
```

```
## El MSE de test estimado por CV para regresión es:  44.15175 .
```

```r
#Para el de SVM

err.in = 0
err.out = 0

for (i in seq(5)) {
  svmfit = svm(crim1 ~ ., data=data.full[folds != i,], kernel = "linear")
  err.in = err.in + mean(svmfit$fitted != data.full[folds != i,]$crim1)
  pred = predict(svmfit, newdata = data.full[folds == i,])
  err.out = err.out + mean(pred != data.full[folds == i,]$crim1)
}

cat("El error de train estimado por CV para SVM es: ", err.in/5, ".\n")
```

```
## El error de train estimado por CV para SVM es:  0.08893778 .
```

```r
cat("El error de test estimado por CV para SVM es: ", err.out/5, ".\n")
```

```
## El error de test estimado por CV para SVM es:  0.1147156 .
```

Como podemos ver para calcular el error de train para la regresión lo que hemos hecho es predecir las etiquetas de los datos de entrenamiento con el modelo estimado a partir de los mismos datos de entrenamiento. Esto lo hemos hecho así ya que ha diferencia del SVM el modelo que obtenemos con *glmnet* no nos devuelve los datos ajustados.

Ejercicio 3

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

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

```r
set.seed(41192)
attach(Boston)
```

```
## The following objects are masked from Boston (position 3):
##
##     age, black, chas, crim, dis, indus, lstat, medv, nox, ptratio,
##     rad, rm, tax, zn
```

```r
train = sample(nrow(Boston), nrow(Boston)*0.8)
test = -train
```

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

```r
set.seed(41192)
bag = randomForest(medv ~., data = Boston, subset = train, mtray = ncol(Boston)-1, importance = TRUE)
pred = predict(bag, newdata = Boston[test,])
cat("El MSE con bagging es: ", mean((pred - Boston[test,]$medv)^2), "\n")
```

```
## El MSE con bagging es:  24.11276
```

Como vemos estamos usando la función *randomForest* ya que el baggin es un caso particual de Random Forest donde el número de variables empleado en cada separación son todas las disponibles.

importance a true lo que hace es que la importancia de los predictores se tenga en cuenta, de hecho he observado que poniendo este parámetro a false, que es su valor por defecto, el error obtenido es algo mayor, aunque no demasiado.

Como vemos el error que estamos calculando es el MSE.

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

Para obtener una aproximación del número de árboles necesario lo que vamos a hacer es usar la validación cruzada con 5 particiones probando valores de 100 a 500 dando saltos de 20 en 20.

```r
set.seed(41192)
#empleamos CV para estimar el mtray óptimo
randomfcv = rfcv(Boston[,1:13], Boston$medv)
print(randomfcv$n.var)
```

```
## [1] 13  6  3  1
```

```r
print(randomfcv$error.cv)
```

```
##       13        6        3        1
## 14.96407 15.07807 17.18453 42.65902
```

```r
#entonces voy a usar mtray = 6 orque es el que menos MSE da quitando el 13 que sería hacer el bagging

#vamos a obtener una estimación del número de árboles necesarios
CVerrs = sapply(seq(100, 500, 20), function(x) {
  K = 5
  folds = kfold(Boston, k = K)
  err = 0

  for (i in seq(K)) {
    rf = randomForest(medv ~ ., data = Boston, subset = which(folds != i), mtry = 6, importance = TRUE)
    pred = predict(rf, newdata = Boston[folds == i,])
    err = err + mean((pred - Boston[folds == i,]$medv)^2)
  }

  err/K
})

opt_ntree = seq(100,500,20)[which.min(CVerrs)]
cat("El numero optimo de arboles es: ", opt_ntree)
```

```
## El numero optimo de arboles es:  300
```

```r
rf = randomForest(medv ~., data = Boston, subset = train, mtray = 6, importance = TRUE)

pred = predict(rf, newdata = Boston[test,])
cat("El MSE con rf con 500 árboles es: ", mean((pred - Boston[test,]$medv)^2), "\n")
```

```
## El MSE con rf con 500 árboles es:  24.10268
```

```r
rf = randomForest(medv ~., data = Boston, subset = train, mtray = 6, ntree = opt_ntree, importance = TRU

pred = predict(rf, newdata = Boston[test,])
cat("El MSE con rf con opt árboles es: ", mean((pred - Boston[test,]$medv)^2), "\n")
```

```
## El MSE con rf con opt árboles es:  25.31449
```

Por lo tanto como podemos comprobar el error con random forest es algo menor que con bagging pero no demasiado, con lo cual podemos pensar en dos cosas: lo primero es que quizás todas las variables no sean relevante para la clasficación e incluso que introduzcan ruido en ésta, en segundo lugar vemos como el random forest con un procedimiento que será más eficiente, al emplear menos variables, obtiene un mejor resultado. Por lo tanto se pone de relieve cómo una buena seleccion de variables es muy importante. Además ha habido una ocasión en la cual el error con random forest ha sido incluso menor, pero ahora con esta semilla los resultados son estos. Esto es usando el mismo número de árboles que el empleado en bagging.

Ahora bien, si usamos el número de árboles que parece ser el óptimo, 300, el error aumenta. Pero dado que este óptimo ha sido obtenido mediante un procedimiento de CV entonces nos quedaríamos en la práctica con él ya que teóricamente tendrá una mejor generalización pese a que para esta muestra concreta el error aumente.

4. 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.

```
set.seed(41192)
BOSton = gbm(medv ~ ., data = Boston[train,], distribution = "gaussian", n.trees = 50000)
pred = predict(BOSton, newdata = Boston[test,], n.trees = 50000)
cat("El MSE con boosting para 50000 árboles es: ", mean((pred - Boston[test,]$medv)^2), "\n")
```

```
## El MSE con boosting para 50000 árboles es:  24.89646
```

```
set.seed(41192)
BOSton = gbm(medv ~ ., data = Boston[train,], distribution = "gaussian", n.trees = 500)
pred = predict(BOSton, newdata = Boston[test,], n.trees = 500)
cat("El MSE con boosting para 500 árboles es: ", mean((pred - Boston[test,]$medv)^2), "\n")
```

```
## El MSE con boosting para 500 árboles es:  58.7642
```

Cuando empleamos 50000 árboles sí que obtenemos un error parecido al obtenido con Random Forest y Boosting, ahora bien, con 500 árboles, ya que tenemos el error para tal cantidad de árboles en los dos métodos anteriores, el MSE es de 58.7642, con lo cuál aquí se hace patente cómo Boosting necesita de muchos clasificadores "débiles" para poder compararse a Random Forest o Bagging.

Ejercicio 4

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

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

```
set.seed(41192)
attach(OJ)
train = sample(nrow(OJ), 800)
test = -train

tree.OJ = tree(Purchase ~ ., OJ[train, ])
```

2. 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.

```
summary(tree.OJ)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ[train, ])
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.6977 = 552.6 / 792
## Misclassification error rate: 0.145 = 116 / 800
```

Lo primero que observamos es que de las 17 variables con las que podría trabajar el árbol para realizar la clasificacion realmente solo usa 3: *LoyalCH*, *PriceDiff* y *ListPriceDiff*. Por lo tanto o bien el resto de características no aportan ninguna información relevante o la información que aportan estas tres es suficiente.

Tenemos que hay 8 nodos terminales por lo tanto en base a las variables anteriormente mencionadas se hacen 8 particiones de los datos a partir de las cuales podemos saber cuál de los dos valores posibles de *Purchase* toma cada objeto de los datos de training en teoría.

Tenemos que se clasifican mal 116 elementos de los 800 por lo tanto aunque el algoritmo ha considerado que no necesita dividir más los datos, quizás porque sus medidas le han dicho que ya no va a obtener nueva información, todos los datos no han sido bien clasificados; hay datos que "han caído" en particiones donde la clase mayoritaria de *Purchase* no era la suya.

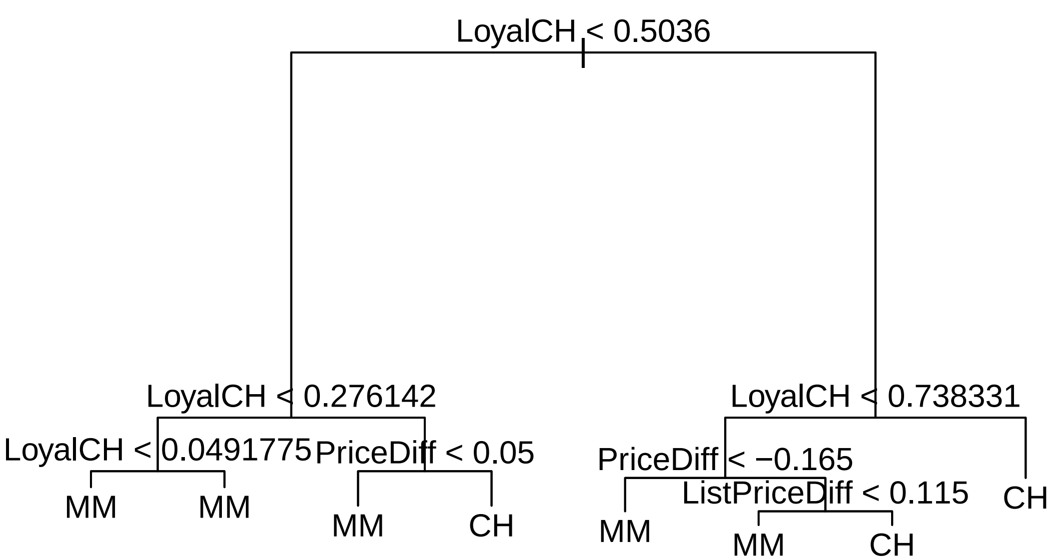
La desviación media se obtiene dividiendo por el número de total de muestras menos el número de nodos final la siguiente cantidadad:

$-2\sum_m \sum_k n_{mk} log\hat{p}_{mk}$

donde n_{mk} es el número de muestras de la clase k que hay en el nodo terminal m-ésimo y $\hat{p}_{mk}$ es la proporción de muestras de la clase k que hay en el nodo m-ésimo, por lo tanto cuanot más nodos de esa clase haya, mayor será esta proporción y en consecuencia el logaritmo será mayor aportando menos a la sumatoria del error, que es lo que estamos midiendo en definitiva.

3. Crear un dibujo del árbol e interpretar los resultados.

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



Entonces lo que vemos que primero se clasifican los datos en base a si el valor de *LoyalCH* está por debajo o por encima de 0.5036, luego si está pode debajo esas muestras se dividen nuevamente en base a si el valor de LoyalHC está por debajo de 0.276142 o no y en caso afirmativo volvemos a dividir por un umbral, aunque como vemos esta división resulta inútil pues ambas particiones son etiquetadas por **MM**. En caso de no estar por debajo de 0.276142 entonces entra en juego la variable PriceDiff que sí que establece dos particiones con etiquetas distintas.

En la otra rama del árbol cuando el valor de *LoyalHC* está por encima de 0.5036 entonces si está por encima de 0.738331 se etiquetan los datos como **CH** y en caso contrario se parten los datos en base a un umbral con *PriceDiff* y en caso de estar por encima de dicho umbral es la variable *ListPriceDiff*, la que quedaba por usarse de las que hemos dicho antes que se para los datos en base a un umbral etiquetando cada una de las dos particiones con una clase distinta.

4. 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 de test? ¿Cuál es la precisión del test?

```r
predOJ = predict(tree.OJ, OJ[test, ], type="class")
table(predOJ, OJ[test,]$Purchase)
```

```
##
## predOJ  CH  MM
##     CH 137  40
##     MM  20  73
```

Como vemos de los que se clasifican como CH hay 137 correctos y 40 incorrectos. Del mismo modo de los que se clasifican como MM hay 20 incorrectos y 73 que sí que pertenecen a esta clase.

La tasa de error de test es $\dfrac{40+20}{270} = 0.222222$.

La precisión es $\dfrac{137}{137+40} = 0.774011$

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

La función cv.tree lo que haces es una validación cruzada particionando el conjunto de datos que le demos en tantas particiones como indiquemos, por defecto es 10. Entonces a partir de esta validación cruzada podemos estimar el valor óptimo de algunos parámetros del algoritmo para luego aplicarlos al conjunto de datos.

```r
cvTree = cv.tree(tree.OJ, FUN = prune.misclass)
print(cvTree)
```

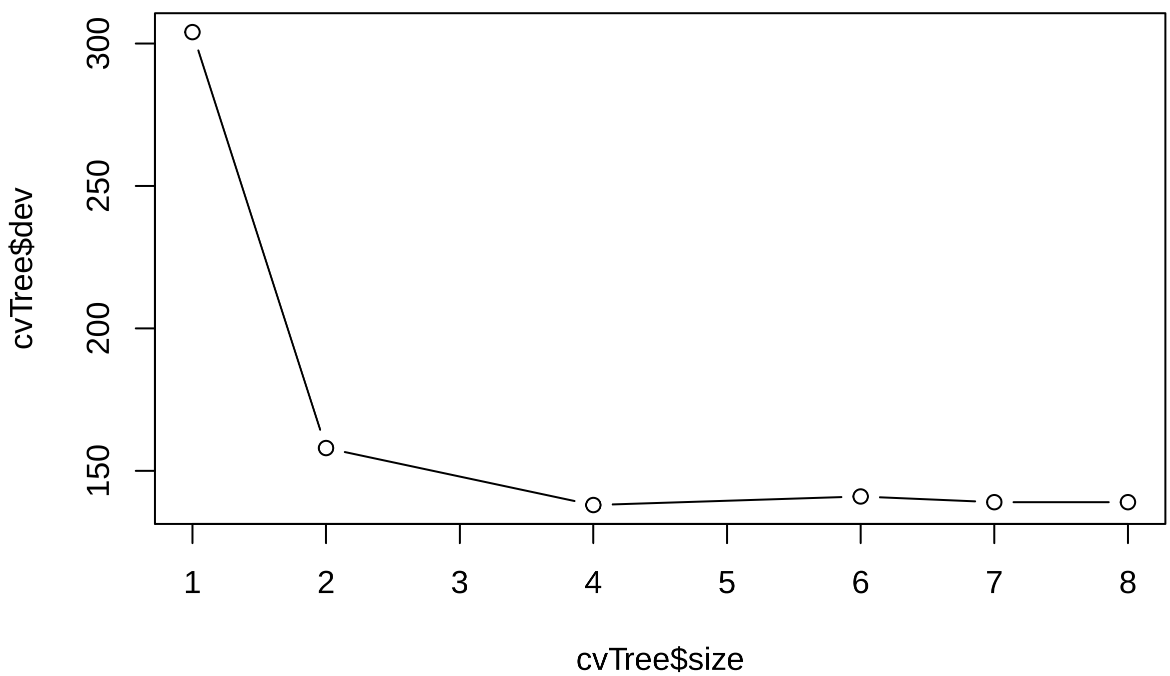
```
## $size
## [1] 8 7 6 4 2 1
##
## $dev
## [1] 139 139 141 138 158 304
##
## $k
```

```
## [1]  -Inf   0.0   1.0   5.5  11.5 153.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Con lo cual en base a la tasa de error de clasificacion el tamaño optimo es el 4, nodos terminales, recordemos que nosotros obteníamos 8 nodos terminales, con lo cuál aquí se reduce a la mitad.

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?

```r
plot(cvTree$size, cvTree$dev, type = 'b')
```



El tamaño del árbol óptimo como ya hemos dicho antes es 4, por lo tanto vemos con disminuye un poco el error, solamente baja en 1, pero el modelo de clasificación sería mucho más simple, no como en el árbol obtenido anteriormente que tenía ramas que sobraban.