

Práctica 3

Anabel Gómez Ríos

```
library(ISLR)
```

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

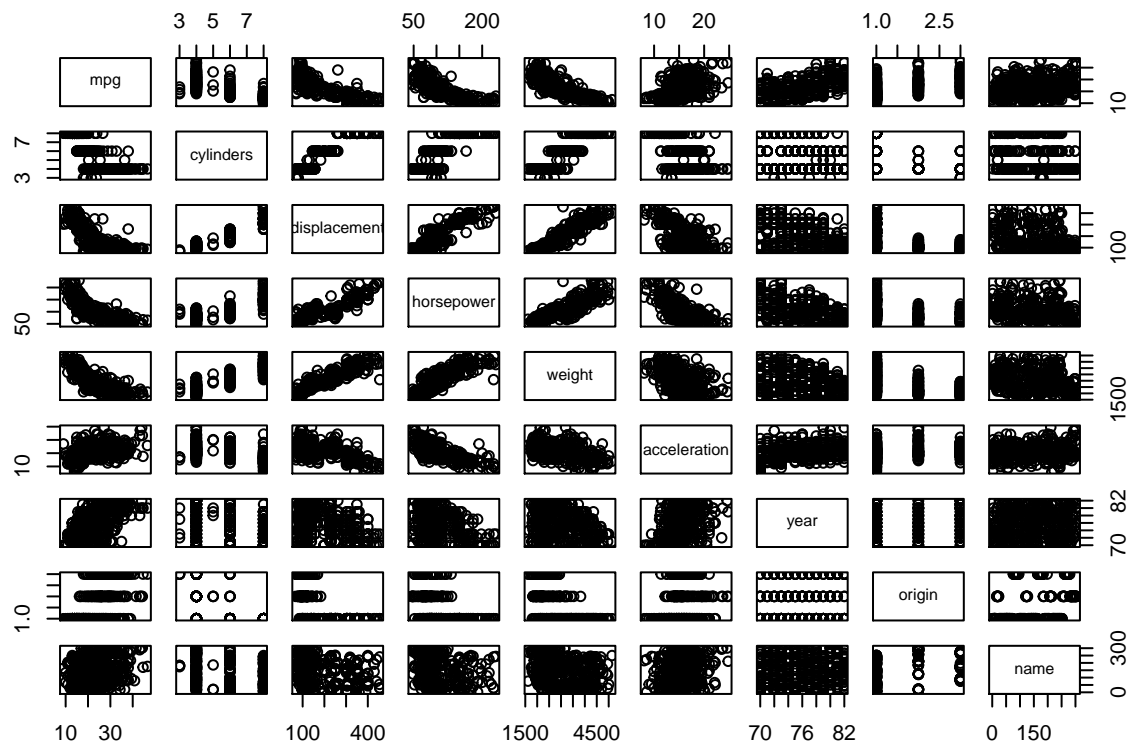
```
#library(MASS)  
#library(class) # Para el KNN  
# Para ahorrarnos el prefijo en Auto$mpg (cada vez que queramos acceder a algo de  
# Auto:  
attach(Auto)
```

Ejercicio 1

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

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.

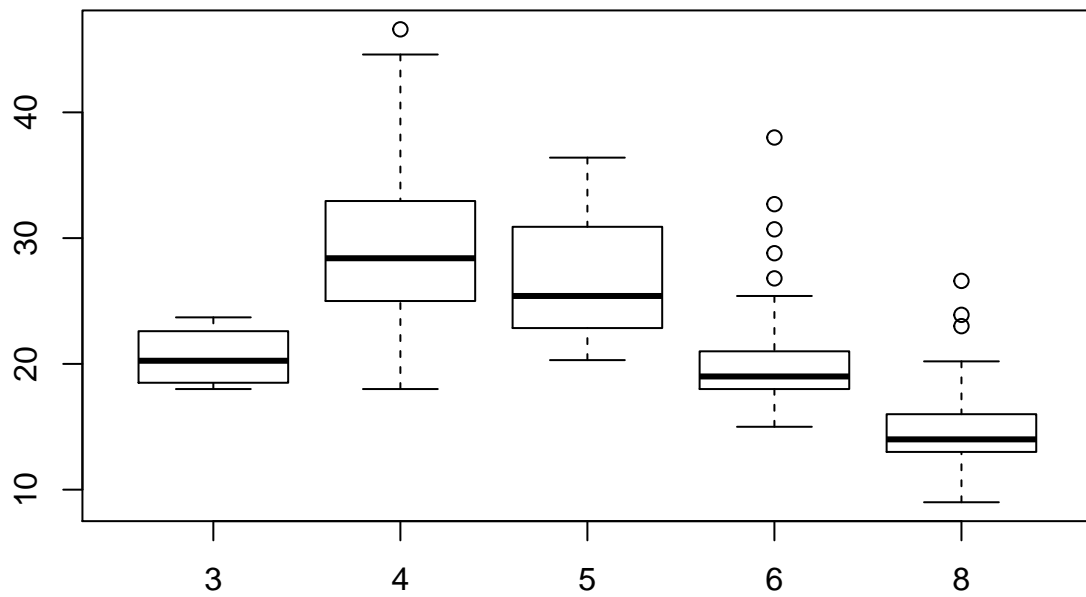
```
pairs(Auto)
```



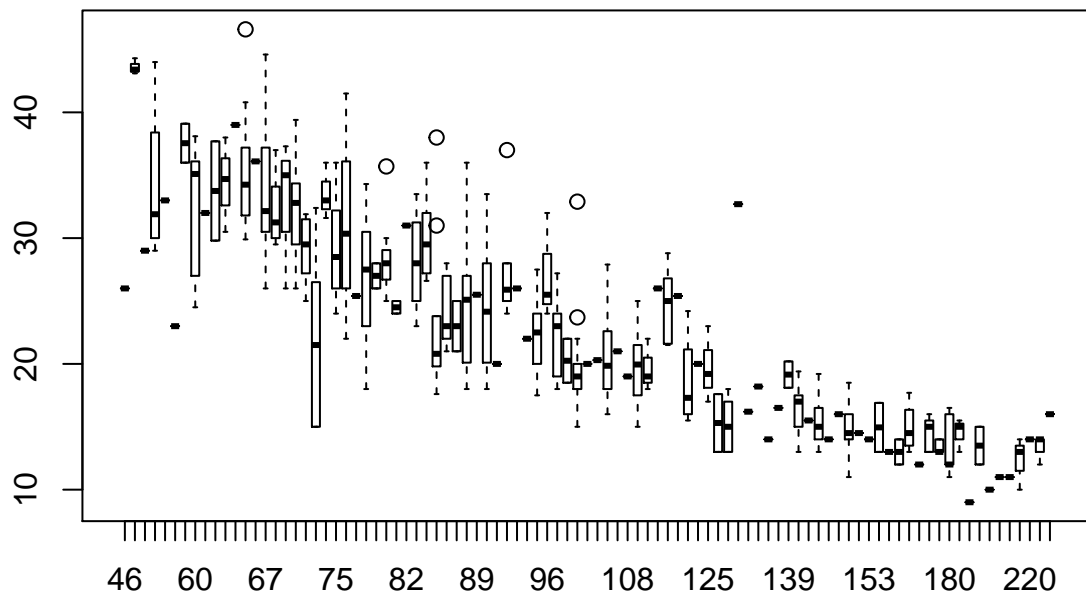
Como vemos con la función `pairs`, las características que parecen más útiles para predecir `mpg` (que son aquellas que tienen un patrón más o menos claro con respecto a `mpg`) son `displacement`, `horsepower` y `weight`.

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

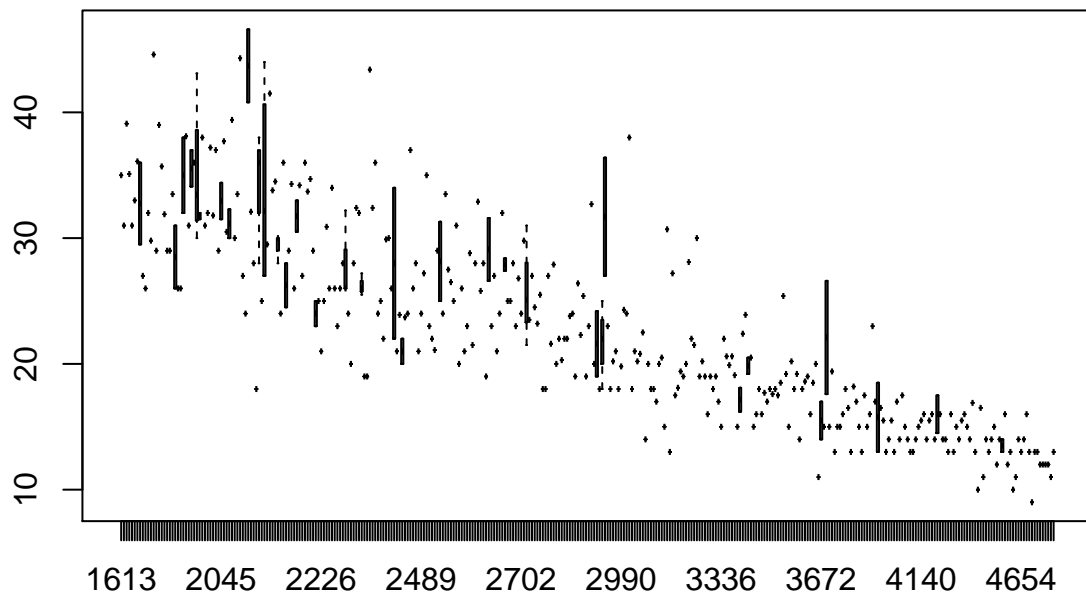
```
boxplot(Auto$mpg~Auto$cylinders)
```



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

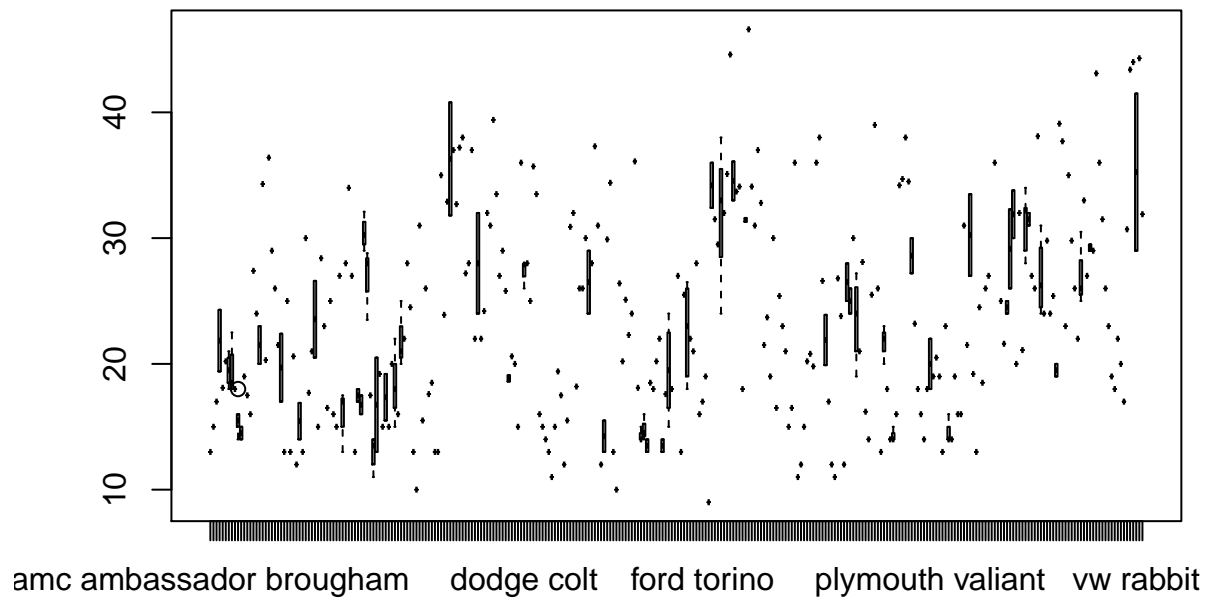


```
boxplot(Auto$mpg~Auto$weight)
```

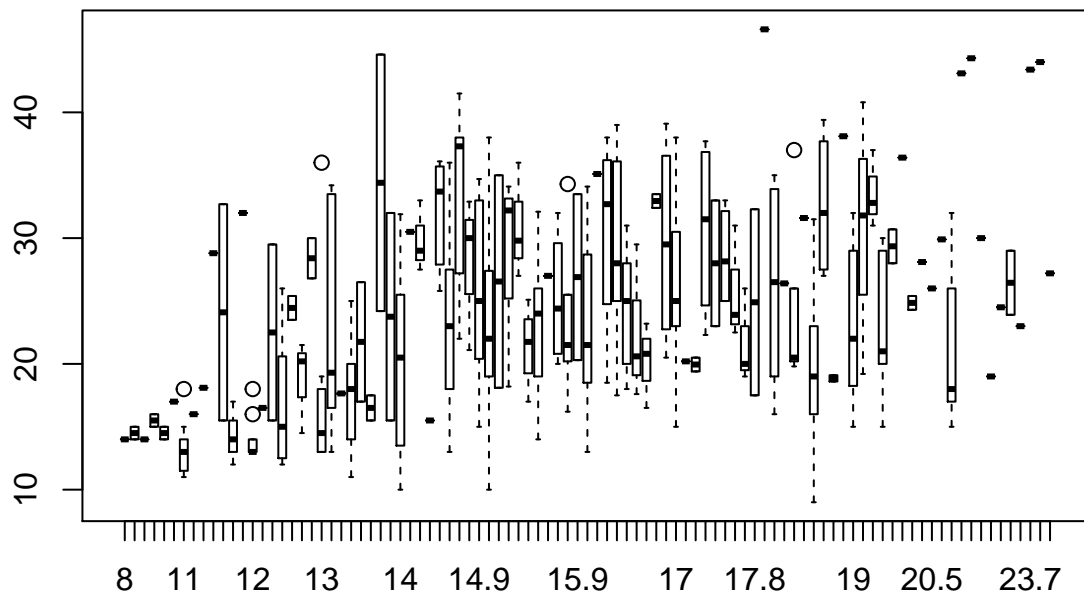


Como vemos las tres siguen más o menos una estructura clara. Veamos otra que no sea ninguna de estas tres:

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



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



Estas por ejemplo, vemos que no tienen mucho que ver con mpg.

b) Seleccionar las variables predictorias que considere más relevantes.

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

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

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

```
train = sample(1:nrow(AutoMod), round(nrow(AutoMod)*0.8))
test = AutoMod[-train, ]
train = AutoMod[train, ]
```

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

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

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

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.

```
modlog1 <- glm(train$mpg~., data = train)
predicciones <- predict(modlog1, test)
comp <- sign(predicciones) == sign(test$mpg)
errores <- comp[comp == FALSE]
error.regresion <- 100*(length(errores)/nrow(test))
cat("El error con regresión ha sido:", error.regresion)
```

```
## El error con regresión ha sido: 14.10256
```

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

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

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

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



```

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

##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
## k
## 5
##
## - best performance: 0.0922379

# Utilizamos knn con el mejor k que nos ha dicho tune.knn
pred.knn <- knn(train[,-1], test[,-1], train[,1], k=5)

```

Vamos a ver ahora el error de test

```

confM <- table(pred.knn, test[,1])
error <- (confM[1,2] + confM[2,1])/(confM[1,1]+confM[1,2]+confM[2,1]+confM[2,2])

```

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

```

library(ROCR)

```

```

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

## Loading required package: gplots

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

##
## Attaching package: 'gplots'

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

```

```

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

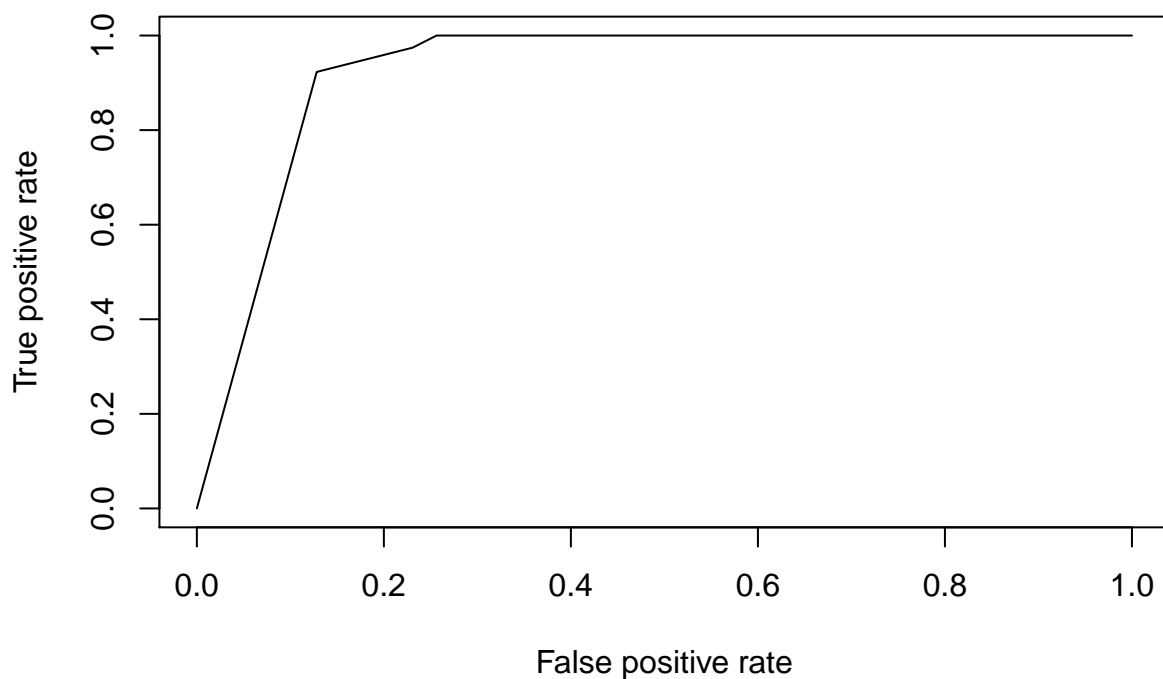
```

```

    }
    else {
      prob[i]
    }
  })

pred <- prediction(prob, test$mpg)
perf <- performance(pred, "tpr", "fpr")
plot(perf)

```



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

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

```

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

```

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.

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

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

Utilizando glmnet, le tenemos que dar un 1 al parámetro alpha para que sea un modelo LASSO.

```
library(glmnet)

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

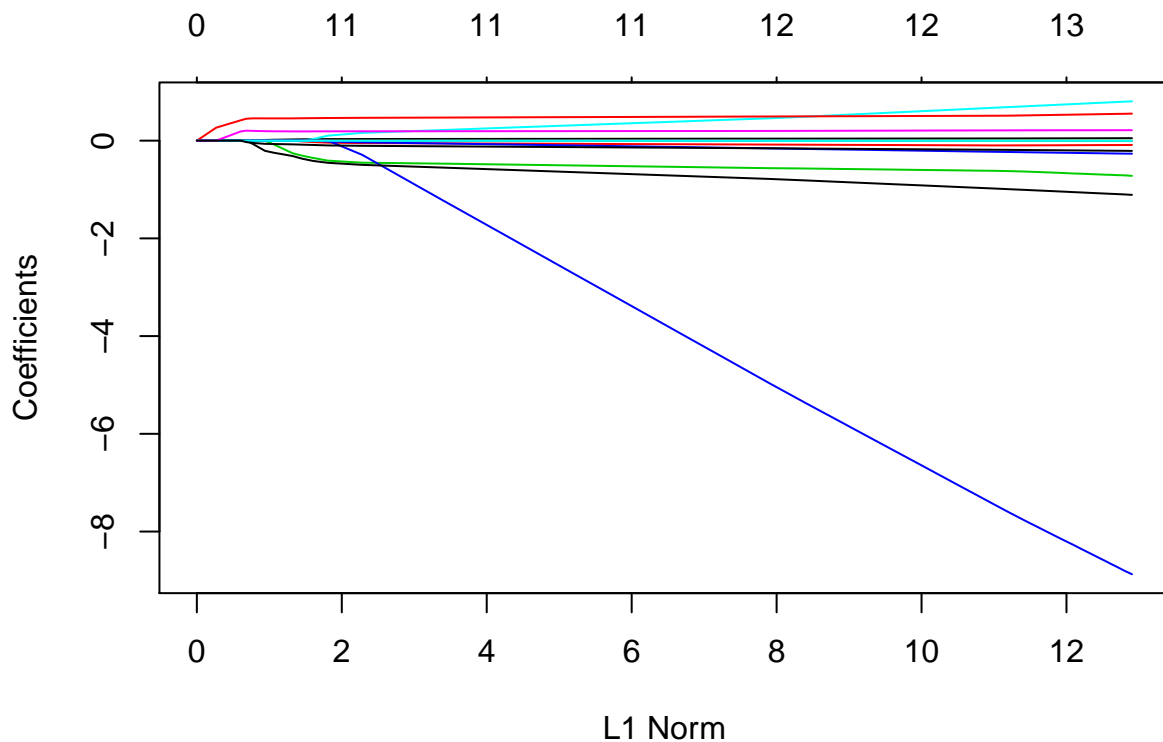
## Loading required package: Matrix

## Loading required package: foreach

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

## Loaded glmnet 2.0-5

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



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

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

```
## (Intercept)      zn      indus      chas      nox
## 6.663693369 0.040417814 -0.076929392 -0.551013275 -4.465790375
##      rm      age      dis      rad      tax
## 0.426488379 0.000000000 -0.752173795 0.491769131 0.000000000
## ptratio    black    lstat    medv
## -0.144639653 -0.003021976 0.198034616 -0.150936268
```

Como vemos hay muchos de estos coeficientes muy cercanos a 0. Nos vamos a quedar con aquellos que estén (en valor absoluto) por encima de un umbral 0.5, que significará que tienen cierta correlación con la variable `crim`, aunque podríamos elegir uno un poco más bajo si quisiéramos considerar más variables (aunque serían menos relevantes)

```
coeficientes <- coeficientes[abs(coeficientes)>0.5]
print(coeficientes)
```

```
## (Intercept)      chas      nox      dis
## 6.6636934 -0.5510133 -4.4657904 -0.7521738
```

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

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

Para *weight-decay* tenemos que usar también `glmnet` pero con el parámetro $\alpha = 0$.

Nos quedamos primero con las variables seleccionadas por el método anterior, y vamos a utilizar el mejor λ que nos ha salido del apartado anterior por cross-validation:

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

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

```
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
```

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

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

```
## El error para lambda = 0.08 ha sido 4.610146
```

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

```
## El error para lambda = 1 ha sido 4.649119
```

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

El error para lambda = 0.02 ha sido 4.608881

```
lambda <- 0.002
modelo.ridge <- glmnet(as.matrix(BostonMod.train[,-1]),
                      as.matrix(BostonMod.train[,1]), alpha=0, lambda=lambda)
predicciones <- predict(modelo.ridge, s=lambda,
                      newx=as.matrix(BostonMod.test[,-1]))
error.res <- sum((BostonMod.test[,1] - predicciones)^2)
error.res <- sqrt(error.res/length(predicciones))
cat("El error para lambda =", lambda, "ha sido", error.res)
```

El error para lambda = 0.002 ha sido 4.608556

Como vemos, cuanto mayor es el λ , menor es el error (mayor la cantidad de regularización), mientras que si lo bajamos, el error aumenta, por lo que sí que estamos ajustando poco el modelo aún, se podría mejorar aumentando λ .

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

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

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

Vamos a hacer ahora un SVM con núcleo lineal

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

```
## El error con SVM lineal ha sido: 22.77228
```

Vamos a probar con los otros tres tipos de núcleos a ver los resultados en errores que obtenemos

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

```
## El error con SVM polinomial ha sido: 18.81188
```

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

```
## El error con SVM radial ha sido: 12.87129
```

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

```
## El error con SVM sigmoidal ha sido: 46.53465
```

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

```
# Borramos lo que no necesitamos
rm(BostonMod.test, BostonMod.train, test, train)
rm(coeficientes, comp, crosssv, error, error.res, errores)
rm(lambda, modelo.lasso, modelo.ridge, predicciones, modelo.svm)
```

Ejercicio 3.

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

- Dividir la base de datos en dos conjuntos de entrenamiento (80%) y test (20%).
- Usando la variable medv como salida y el resto como predictoras, ajustar un modelo de regresión usando bagging. Explicar cada uno de los parámetros usados. Calcular el error del test.
- Ajustar un modelo de regresión usando Random Forest. Obtener una estimación del número de árboles necesario. Justificar el resto de parámetros usados en el ajuste. Calcular el error de test y compararlo con el obtenido con bagging.
- 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.

Ejercicio 4.

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

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

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

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

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

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

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


```
summary(OJ)
```

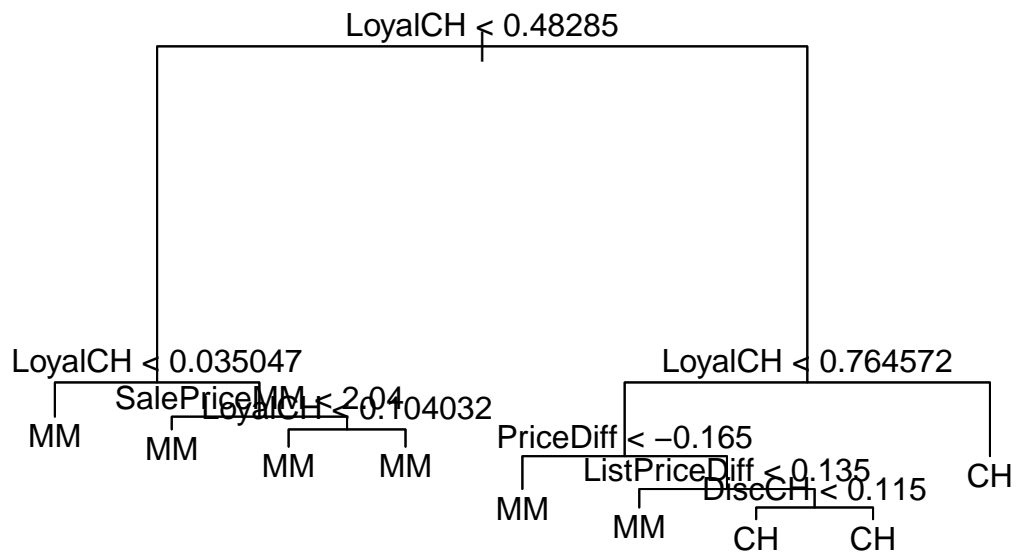
```
## Purchase WeekofPurchase StoreID PriceCH PriceMM
## CH:653 Min. :227.0 Min. :1.00 Min. :1.690 Min. :1.690
## MM:417 1st Qu.:240.0 1st Qu.:2.00 1st Qu.:1.790 1st Qu.:1.990
## Median :257.0 Median :3.00 Median :1.860 Median :2.090
## Mean :254.4 Mean :3.96 Mean :1.867 Mean :2.085
## 3rd Qu.:268.0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

Como vemos, Purchase toma sólo dos valores: CH y MM, con lo que vamos a utilizar clasificación:

```
tree.oj <- tree(train$Purchase~., train)
```

Vamos a ver el resultado de este árbol:

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



Como este no queda muy bien, vamos a hacerlo también con el software **RWeka**, que también permite dibujar árboles, y se ven mejor:

```
library(RWeka)
```

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

```
library(partykit)
```

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

```
## Loading required package: grid
```

```
oj.rweka = J48(OJ$Purchase~., data = OJ, subset = train.idx)
plot(oj.rweka)
```


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

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

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

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

El error en test es $(11+39)/(125+11+39+95) = 0.1851852$

La precisión es $(125+95)/(125+11+39+95) = 0.8148148$

Es decir, hay un error en test del 18.5% y una precisión del 81.4%.

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

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

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

```
## $size
## [1] 9 5 4 2 1
##
## $dev
## [1] 144 144 145 151 311
##
## $k
## [1] -Inf  0.0  2.0  7.5 163.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"          "tree.sequence"
```

Como podemos ver, los árboles con 9 y 5 nodos terminales son los que tienen el menor error, 144. El árbol que habíamos ajustado en el apartado c) tenía justo 9 nodos terminales, por lo que ya habíamos obtenido aquel con menor error y mejor precisión.

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?