# Practica 3

*Rafael Nogales*

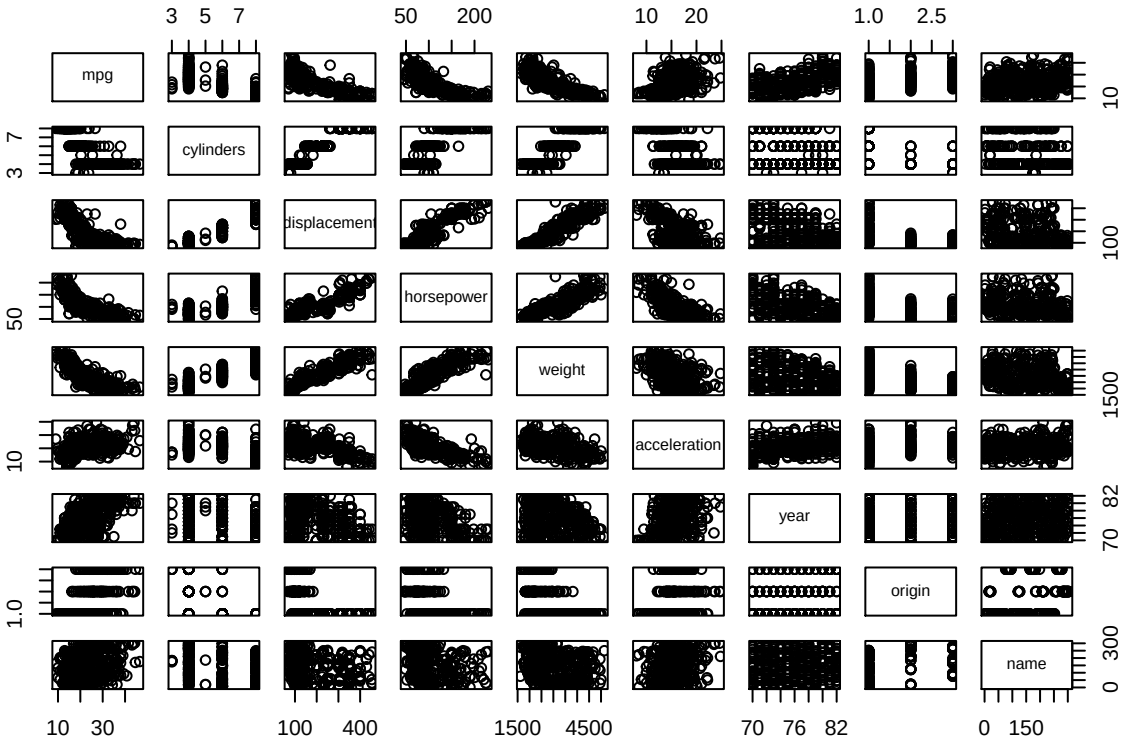
*29 de mayo de 2016*

## Ejercicio1

**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 mediana de la variable mpg y bajo en caso contrario.**

**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.**

```
library("ISLR")
pairs(Auto)
```




Las que tienen una relación más directa con el consumo del coche (mpg) son: horsepower (potencia en CV), weight (peso), displacement (cilindrada del motor), cylinder (nº de cilindros) y year (año)

Se puede apreciar la correlación en las graficas.

**Seleccionar las variables predictoras que considere más relevantes.**

En mi caso voy a utilizar displacement, horsepower, weight y año ya que son las que he visto que minimizan el error.

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

```
#Creamos los indices de TRAIN, los de TEST son los restantes
library("caret")
```

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

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

```
## Warning: package 'ggplot2' was built under R version 3.2.4
```

```
set.seed(15)
train <- createDataPartition(Auto$mpg, times = 1, p = 0.8, list = F)
```

**Crear una variable binaria, mpg01, que será igual 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).**

```
umbral <- median(Auto$mpg)
mpg01 <- 2*(Auto$mpg > umbral)-1
XAuto <- cbind(Auto, mpg01)
```

**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.**

```
Auto.train <- XAuto[train,]
Auto.test <- XAuto[-train,]
Auto.model <- glm(mpg01~ displacement + horsepower + weight + year, data = Auto.train)
summary(Auto.model)
```

```
##
## Call:
## glm(formula = mpg01 ~ displacement + horsepower + weight + year,
##     data = Auto.train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6172  -0.3442   0.1247   0.4397   1.7357
```

```
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -3.2000549  0.8577860  -3.731 0.000227 ***
## displacement -0.0031041  0.0010678  -2.907 0.003912 **
## horsepower    0.0036924  0.0021221   1.740 0.082858 .
## weight       -0.0005728  0.0001180  -4.853 1.93e-06 ***
## year          0.0673809  0.0107549   6.265 1.24e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.3945955)
##
##     Null deviance: 315.00  on 314  degrees of freedom
## Residual deviance: 122.32  on 310  degrees of freedom
## AIC: 607.97
##
## Number of Fisher Scoring iterations: 2
```

```
mpg.pred <- predict(Auto.model, Auto.test)
mpg.pred.class <- 2*(mpg.pred > 0)-1
t<- table(predict=mpg.pred.class, truth=Auto.test$mpg01)
t
```

```
##        truth
## predict -1  1
##      -1 33  0
##      1   5 39
```

```
error <- 1 - sum(diag(t))/sum(t)
error
```

```
## [1] 0.06493506
```

**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)**

```
#Creamos particion de training
training <- XAuto[train,]
trainX  <- training[,( names(training) =="displacement" | names(training) == "horsepower" |
                           names(training) == "weight" | names(training) == "year") ]

#Creamos particion de testing
testing <- XAuto[-train,]
testX  <- testing[,( names(testing) =="displacement" | names(testing) == "horsepower" |
                           names(testing) == "weight" | names(testing) == "year") ]

labels.train <- XAuto[train, "mpg01"]
labels.test <- XAuto[-train, "mpg01"]

#Normalizamos los datos de train y test, es importante hacerlo por separado ya que
```

```r
# el train no conoce nada del test (tampoco su escala)
trainNorm <- scale(trainX)
testNorm <- scale(testX)
```

```r
library("e1071")
set.seed(1)
tune_k <- tune.knn(trainNorm, as.factor(XAuto[train,names(XAuto)=="mpg01"]), k=1:10, tune_control = tun
summary(tune_k)
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  3
##
## - best performance: 0.0733871
##
## - Detailed performance results:
##     k      error dispersion
## 1   1 0.08266129 0.05521397
## 2   2 0.09526210 0.05229481
## 3   3 0.07338710 0.07893545
## 4   4 0.07973790 0.05960656
## 5   5 0.08911290 0.06381791
## 6   6 0.08911290 0.06713235
## 7   7 0.07973790 0.06116782
## 8   8 0.07973790 0.06691715
## 9   9 0.08286290 0.06675218
## 10 10 0.08286290 0.06122013
```

Como vemos el mejor parametro es $k = 3$, notemos que hemos utilizado únicamente los datos del train para elegir el parámetro, en este caso no sería grave haber utilizado todos los datos ya que se está haciendo validación cruzada y lo único que vamos a elegir es el mejor valor de $k$ ya que obtenemos un error de solo 8% en train
Pero aun asi nos matenemos fieles a la filosofía de que los datos de test se van a usar unicamente para el test, con el objetivo de no falsear los resultados.

```r
library("class")
knn.predict <- knn(trainNorm, testNorm, labels.train, k=3)
knn.conf.matrix <- table(knn.predict, labels.test)
print(knn.conf.matrix)
```

```
##            labels.test
## knn.predict -1  1
##          -1 38  0
##          1   0 39
```

Como sabemos el acierto es la proporción entre la suma de la diagonal principal y la sumna de la matriz completa, por tanto el error es:

```
error <-  1 - sum(diag(knn.conf.matrix)) / sum(knn.conf.matrix)
print(error)
```

```
## [1] 0
```

En este caso vemos que hemos tenido mucha suerte ya que hemos acertado todos los casos de test, eso significa que nuestro clasificador es muy bueno, ya que a pesar de equivocarse en train (8% error) es capaz de acertar en test que es lo más importante.

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

Para pintar las curvas ROC vamos a escribir una pequeña función que toma como parámetros un vector con nuestras predicciones y otro con los datos verdaderos.

```
library("ROCR")
```

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

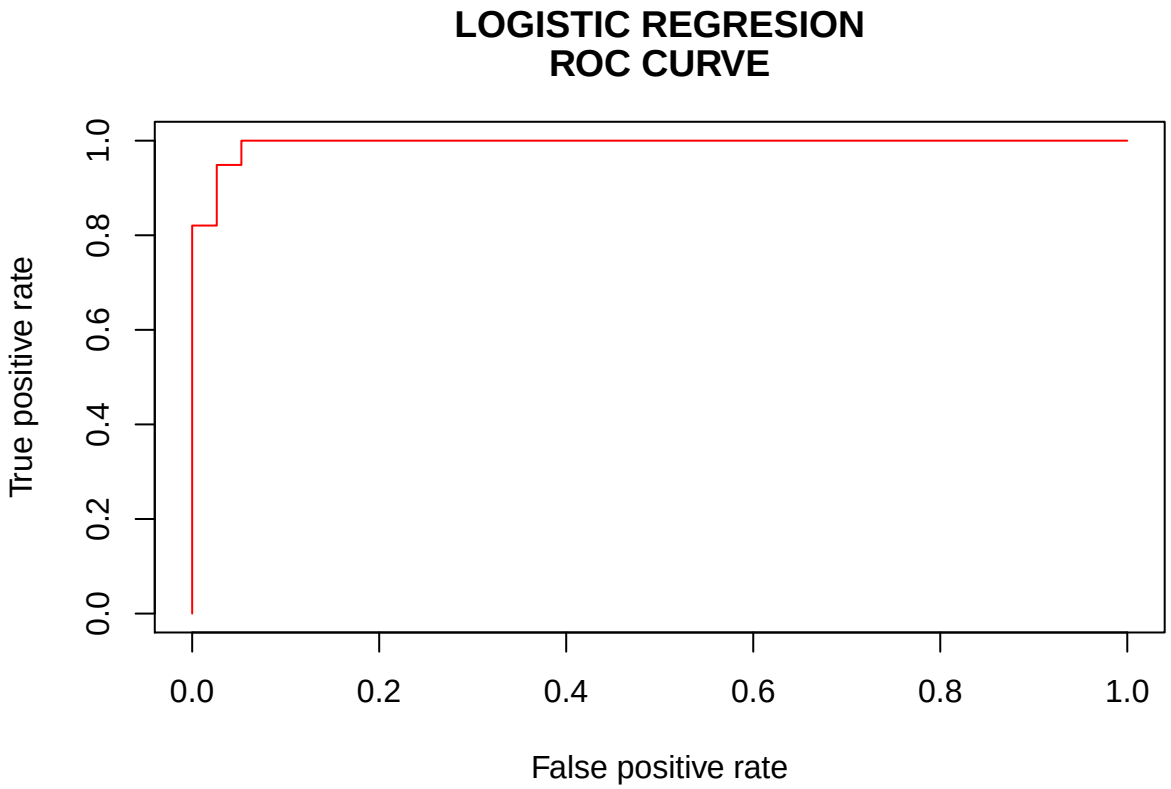
```
## Warning: package 'gplots' was built under R version 3.2.4
```

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

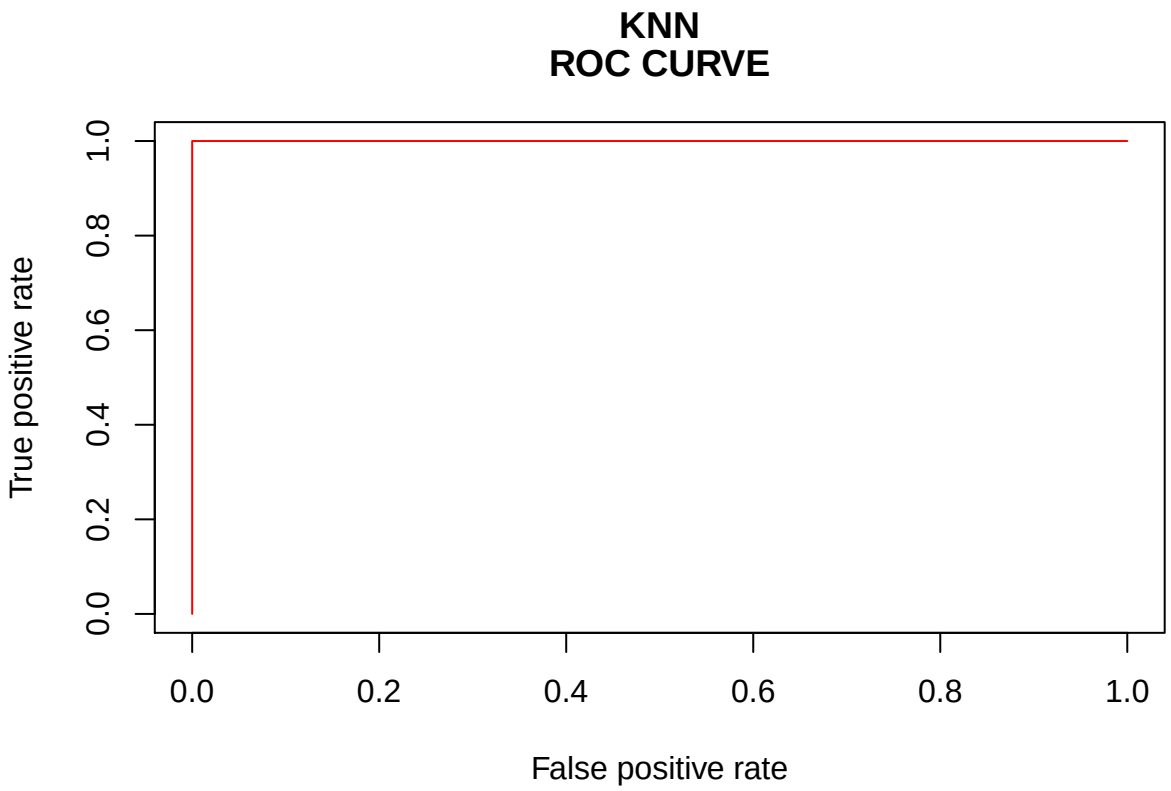
```
## The following object is masked from 'package:stats':
##
##     lowess
```

```
rocplot <- function(pred, truth, ...){
    if(class(pred) == "factor"){
        pred <- as.numeric(pred)
        #Ahora tenemos un vector de 1's y 2's
        #Lo pasamos a -1 y 1
        pred <- 2*pred-3
    }
    predob <- prediction(pred, truth)
    perf <- performance(predob, "tpr", "fpr")

    plot(perf, ...)
}
```

```
#Logistic rocplot
log.predict <- mpg.pred             #Solo es renombrar
rocplot(log.predict, labels.test, col="red", main=c("LOGISTIC REGRESION", "ROC CURVE"))
```




```
#KNN rocplot
rocplot(knn.predict, labels.test, col="red", main=c("KNN", "ROC CURVE"))
```



## BONUS - 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.

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.

```
cv.error.rl <- function(data, k=5){
    errores <- vector(length = k)
    for(i in 1:k){
        labels <- XAuto[,"mpg01"]
        train <- createDataPartition(labels, times = 1, p = 0.8, list = F)

        Auto.train <- XAuto[train,]
        Auto.test <- XAuto[-train,]

        labels.train <- Auto.train[, "mpg01"]
        labels.test  <- Auto.test[, "mpg01"]

        Auto.model <- glm(mpg01~ displacement + horsepower + weight + year, data = Auto.train)
        summary(Auto.model)
        mpg.pred <- predict(Auto.model, Auto.test)

        #Pasamos de una variable real a una discreta asignando -1 o 1 en funcion de su cercania
        mpg.pred.class <- 2*(mpg.pred > 0)-1
        rl.conf.matrix <- table(predict=mpg.pred.class, truth=labels.test)

        error <- 1 - sum(diag(rl.conf.matrix))/sum(rl.conf.matrix)
        errores[i] <- error
    }

    return(mean(errores))
}
```

```
set.seed(10)
cv.error.rl(XAuto)
```

```
## [1] 0.1076923
```

```
cv.error.knn <- function(data, k=5){
    errores <- vector(length = k)
    for(i in 1:k){
        labels <- XAuto[,"mpg01"]
        train <- createDataPartition(labels, times = 1, p = 0.8, list = F)

        labels.train <- XAuto[train, "mpg01"]
        labels.test <- XAuto[-train, "mpg01"]

        #Creamos particion de training
        training <- XAuto[train,]
        trainX  <- training[,( names(training) =="displacement" | names(training) == "horsepower" |
                           names(training) == "weight" | names(training) == "year") ]
```

```
        #Creamos particion de testing
        testing <- XAuto[-train,]
        testX  <- testing[,( names(testing) =="displacement" | names(testing) == "horsepower" |
                          names(testing) == "weight" | names(testing) == "year") ]


        #Normalizamos los datos de train y test, es importante hacerlo por separado ya que
        # el train no conoce nada del test (tampoco su escala)
        trainNorm <- scale(trainX)
        testNorm <- scale(testX)

        #Optimizamos el parametro k
        tune_k <- tune.knn(trainNorm, as.factor(XAuto[train,names(XAuto)=="mpg01"]), k=1:10,
                          tune_control = tune.control(sampling = "cross"))

        #Nos quedamos en cada caso con el mejor k
        k <- as.numeric(tune_k$best.parameters)

        #Hacemos la prediccion
        knn.predict <- knn(trainNorm, testNorm, labels.train, k)

        #Medimos el error
        knn.conf.matrix <- table(knn.predict, labels.test)
        error <-  1 - sum(diag(knn.conf.matrix)) / sum(knn.conf.matrix)
        errores[i] <- error
    }
    return(mean(errores))
}
```

```
set.seed(10)
cv.error.knn(XAuto)
```

```
## [1] 0.06923077
```

## Ejercicio2

**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.**

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

```
library("MASS")
library("glmnet")
```

```
## Warning: package 'glmnet' was built under R version 3.2.4
```

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

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

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

```
set.seed(13)
lambdas <- cv.glmnet(x = as.matrix(Boston[,-1]), y = as.matrix(Boston[,1]), alpha=1)
bestLambda <- lambdas$lambda.min
lasso.model <- glmnet(x = as.matrix(Boston[,-1]), y = as.matrix(Boston[,1]), alpha = 1, standardize = F
lasso.coef <- predict(lasso.model, type="coefficients", s=bestLambda)[1:14,]
lasso.coef
```

```
##  (Intercept)           zn        indus         chas          nox
## 10.070220392  0.044793719 -0.109319719  0.000000000  0.000000000
##           rm          age          dis          rad          tax
##  0.048302129 -0.003843237 -0.735147927  0.550892493 -0.003560762
##      ptratio        black        lstat         medv
## -0.102202500 -0.007973239  0.117022106 -0.157816171
```

```
selected <- lasso.coef[abs(lasso.coef) > 0.1 ]
selected
```

```
## (Intercept)       indus         dis         rad     ptratio       lstat
##  10.0702204  -0.1093197  -0.7351479   0.5508925  -0.1022025   0.1170221
##        medv
##  -0.1578162
```

**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".**

```
set.seed(13)

nameSelected <- names(selected[2:length(selected)])
ridge.model <- glmnet(x = as.matrix(Boston[,nameSelected]), y = matrix(Boston[,1]), alpha = 0, standardi
ridge.pred <- predict(ridge.model, newx = as.matrix(Boston[,nameSelected]), s=bestLambda, type = "respo
error <- mean((ridge.pred - Boston[,1])^2)
error
```

```
## [1] 41.72708
```

El error en train es muy grande, esto es debido a que el modelo no es capaz de explicar la complejidad de la realidad. Es decir hay "underfitting"

**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.**

```
set.seed(13)
library("e1071")
umbral <- median(Boston$crim)
#Pasamos de 0, 1 a -1, 1
XCrim <- 2*(Boston$crim > umbral)-1

#Sustituimos crim por la nueva columna y creamos un nuevo dataset
XBoston <- cbind(XCrim, Boston[,-1])
#Creamos un clasificador SVM linear y otro radial
XBostonSVM <- svm(XCrim~., data=XBoston, kernel="linear", cost=0.1, scale = FALSE)
XBostonSVM2 <- svm(XCrim~., data=XBoston, kernel="radial", cost=10, scale = FALSE)
```

```
#Utilizamos el clasificador para predecir las etiquetas
crime.pred <- predict(XBostonSVM, XBoston)
#Pasamos de variables reales a variables discretas
crime.pred.class <- 2*(crime.pred > 0)-1
#Vemos su matriz de confusion
table(predict=crime.pred.class, truth=XBoston$XCrim)
```

```
##        truth
## predict  -1   1
##      -1 237  76
##      1   16 177
```

```
#Analogo con el caso radial
crime.pred2 <- predict(XBostonSVM2, XBoston)
crime.pred.class2 <- 2*(crime.pred2 > 0)-1
table(predict=crime.pred.class2, truth=XBoston$XCrim)
```

```
##        truth
## predict  -1   1
##      -1 253   0
##      1    0 253
```

Como vemos en el caso de utilizar un kernel radial el error de entrenamiento se reduce a cero, ya que hay un sobreaprendizaje muy notorio. En este problema en concreto es mejor utilizar el kernel lineal.

**(Bonus) Estimar el error de entrenamiento y test por validación cruzada de 5 particiones.**

```
cv.error.svm <- function(k=5, test=TRUE){
    errores <- vector(length = k)
    for(i in 1:k){
        if(test){
```

```
            labels <- XBoston$XCrim
            train <- createDataPartition(labels, 1, 0.8, FALSE)
            XBoston.train <- XBoston[train,]
            XBoston.test <- XBoston[-train,]
            XBostonSVM <- svm(XCrim~., data=XBoston.train, kernel="linear", cost=0.1, scale = FALSE)
            crime.pred <- predict(XBostonSVM, XBoston.test)
            crime.pred.class <- 2*(crime.pred > 0)-1
            svm.conf.matrix <- table(predict=crime.pred.class, truth=XBoston.test$XCrim)

            error <- 1 - sum(diag(svm.conf.matrix))/sum(svm.conf.matrix)
            errores[i] <- error
        }
        else{
            XBostonSVM <- svm(XCrim~., data=XBoston, kernel="linear", cost=0.1, scale = FALSE)
            crime.pred <- predict(XBostonSVM, XBoston)
            crime.pred.class <- 2*(crime.pred > 0)-1
            svm.conf.matrix <- table(predict=crime.pred.class, truth=XBoston$XCrim)

            error <- 1 - sum(diag(svm.conf.matrix))/sum(svm.conf.matrix)
            errores[i] <- error
        }

    }

    return(mean(errores))
}
```

```
set.seed(10)
cv.error.svm(test = TRUE)
```

```
## [1] 0.182
```

```
cv.error.svm(test = FALSE)
```

```
## [1] 0.1818182
```

Como vemos el error en test y train es muy parecido, esto indica que nuestro clasificador de kernel lineal es bueno y que "no ha sobreajustado".

# 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%).

```
set.seed(13)
library("MASS")
library("caret")
library("randomForest")
```

```
## randomForest 4.6-12

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

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
train <- createDataPartition(Boston$medv, times = 1, p = 0.8, list = F)
Boston.train <- Boston[train,]
Boston.test <- Boston[-train,]
```

**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.**

```
#Creamos un modelo
Boston.model <- randomForest(medv~., data = Boston, subset = train, mtry=13,ntree=25, importance = TRUE)
#Utilizamos el modelo para estimar las etiquetas de test
Boston.predict <- predict(Boston.model, newdata = Boston[-train,])
#Calculamos el ECM
mean((Boston$medv[-train] - Boston.predict)^2)
```

```
## [1] 7.461966
```

- data es el conjunto de datos
- subset es el subconjunto que vamos a utlizar para crear el modelo
- mtry es el numero de variables predictoras que vamos a utilizar en cada arbol
- ntree es el numero de arboles del "bosque"
- importance = TRUE indica que quiero considerar la importancia relativa de cada variable a la hora de hacer el modelo

**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.**

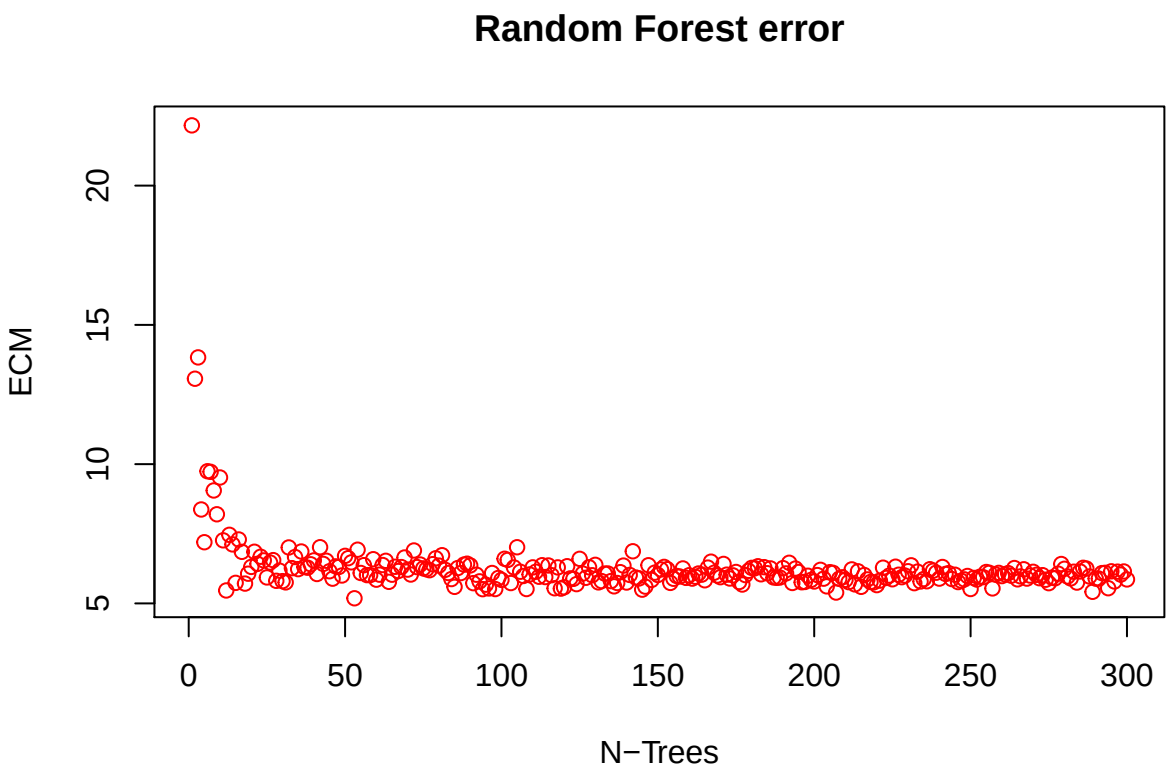
```
set.seed(13)
train <- createDataPartition(Boston$medv, times = 1, p = 0.8, list = F)
Boston.train <- Boston[train,]
Boston.test <- Boston[-train,]
Boston.model <- randomForest(medv~., data = Boston, subset = train, ntree=50, importance = TRUE)
Boston.predict <- predict(Boston.model, newdata = Boston[-train,])
mean((Boston$medv[-train] - Boston.predict)^2)
```

```
## [1] 6.394562
```

Notamos que ahora mtry no aparece, se toma la configuración por defecto (random forest) en lugar de usar todas las variables.
Vemos que a pesar de usar menos variables hemos conseguido mejorar el error, ya que ahora hay más variedad en los clasificadores elementales que vamos a combinar y esto da riqueza al modelo.

Vamos a estimar el número más adecuado de árboles con la siguiente función:

```
#Estimacion del numero correcto de arboles:
errorNtree <- function(ntree_arg){
    Boston.model <- randomForest(medv~., data = Boston, subset = train, ntree=ntree_arg, importance = T
    Boston.predict <- predict(Boston.model, newdata = Boston[-train,])
    return(mean((Boston$medv[-train] - Boston.predict)^2))
}

N <- 300
error <- sapply(1:N, errorNtree)

plot(1:N, error, col="red", main = "Random Forest error", xlab="N-Trees", ylab="ECM")
```




```
bestntree <- which.min(error)
cat(c("El menor error se produce con ", bestntree, " arboles: ",error[bestntree]))
```

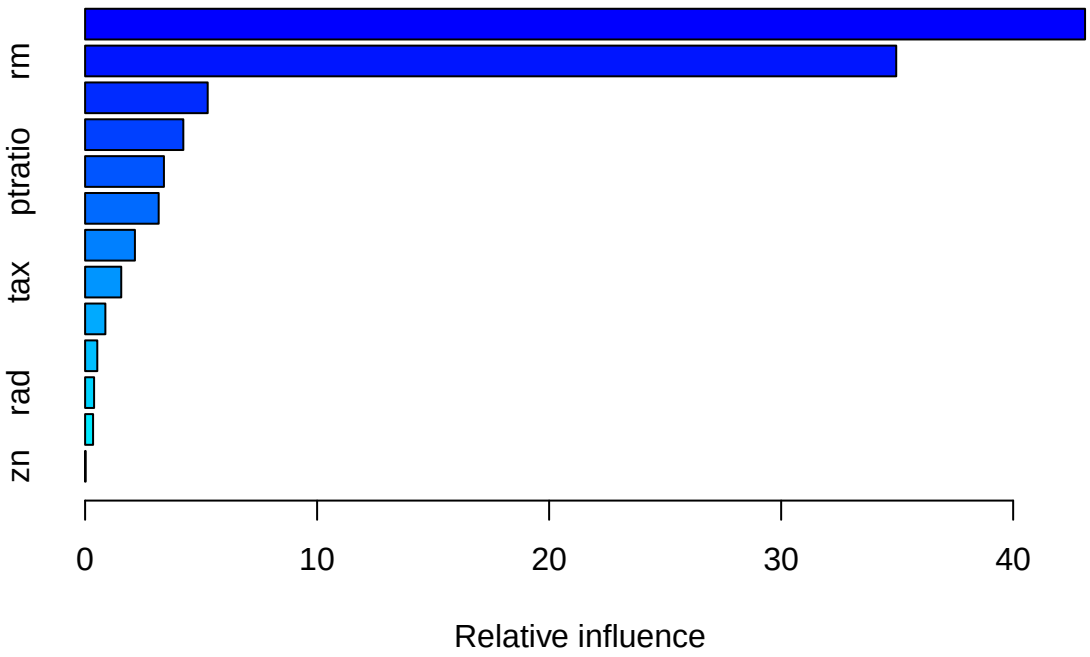
```
## El menor error se produce con  53  arboles:  5.18446634084197
```

**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.**

```
library("gbm", lib.loc="/Library/Frameworks/R.framework/Versions/3.2/Resources/library")
```

```
## Loading required package: survival

##
## Attaching package: 'survival'

## The following object is masked from 'package:caret':
##
##     cluster

## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.1
```

```
set.seed(13)
Boston.boost <- gbm(medv~., data= Boston[train,], distribution = "gaussian", n.trees = 5000, interactio
summary(Boston.boost)
```




```
##             var     rel.inf
## lstat     lstat 43.09877286
## rm           rm 34.95972683
## dis         dis  5.28393505
## nox         nox  4.23344679
```

```
## ptratio ptratio  3.39754974
## crim       crim  3.17061071
## age         age  2.14714290
## tax         tax  1.55705033
## black     black  0.87341835
## indus     indus  0.52581389
## rad         rad  0.38790283
## chas       chas  0.34331890
## zn           zn  0.02131081
```

```
Boston.predict <- predict(Boston.boost, newdata = Boston[-train,], n.trees = 5000)
mean((Boston.predict - Boston$medv[-train])^2)
```

```
## [1] 6.551035
```

El error de boosting es similar al de random forest, pero es más peligroso porque puede sobreajustar con más facilidad si no se regulariza adecuadamente, yo me quedaría con el random forest en este caso.

# 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 predictores (paquete tree de R).**

```
set.seed(1)
library("ISLR", lib.loc="/Library/Frameworks/R.framework/Versions/3.2/Resources/library")
train <- sample(1:1070, 800, replace = F)
train.data <- OJ[train, ]

library("tree", lib.loc="/Library/Frameworks/R.framework/Versions/3.2/Resources/library")
modeloOJ <- tree(formula = Purchase ~., data = OJ, subset = train)
```

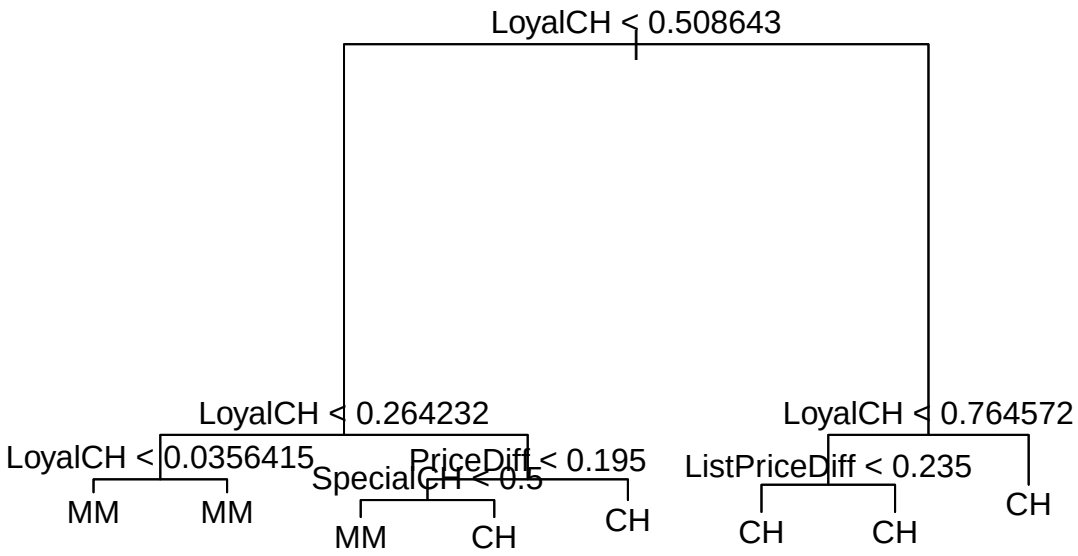
**Usar la función summary() para generar un resumen estadístico acerca del árbol y describir los resultados obtenidos: tasa de error de "training", número de nodos del árbol, etc.**

```
summary(modeloOJ)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ, subset = train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "SpecialCH"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7305 = 578.6 / 792
## Misclassification error rate: 0.165 = 132 / 800
```

**Crear un dibujo del árbol e interpretar los resultados**

```
plot(modeloOJ)
text(modeloOJ)
```




**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?**

```
OJ.pred <- predict(modeloOJ, OJ[-train,], type = "class")
tree.conf.matrix <- table(OJ.pred, OJ[-train,1])
print(tree.conf.matrix)
```

```
##
## OJ.pred  CH  MM
##      CH 147  49
##      MM  12  62
```

```
tasa.error <- sum(diag(tree.conf.matrix))/sum(tree.conf.matrix)
cat(tasa.error)
```

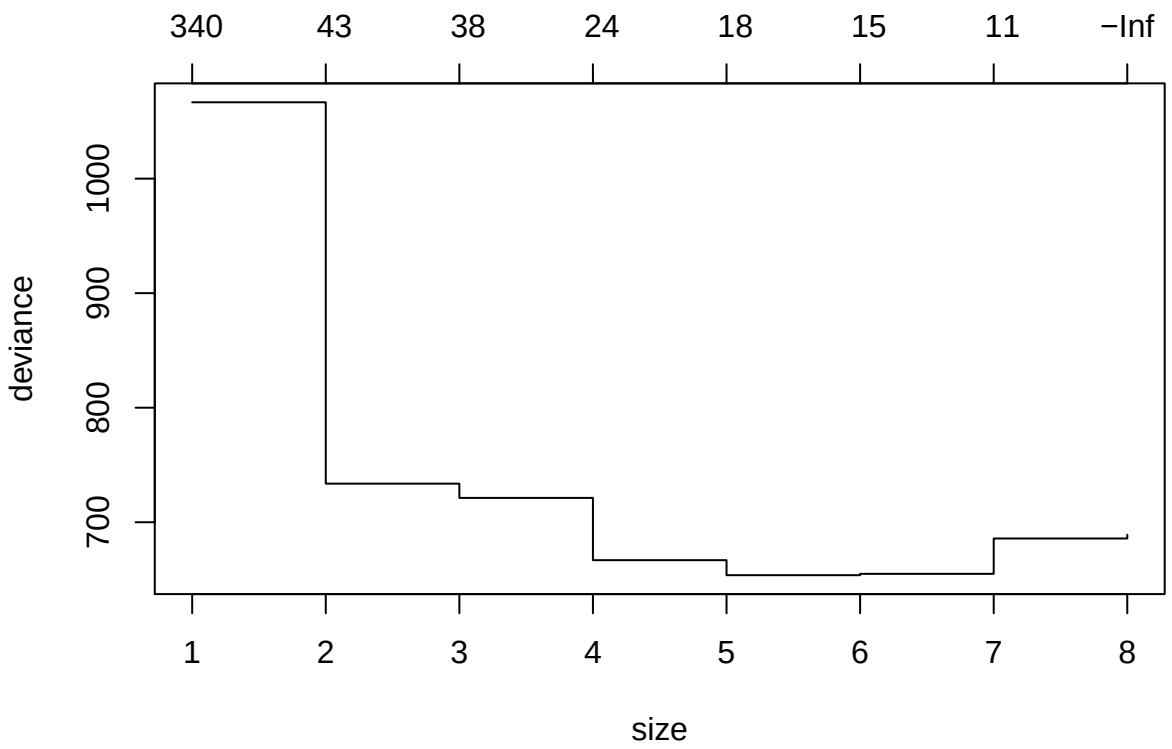
```
## 0.7740741
```

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

```
cv.OJ <- cv.tree(modeloOJ)
print(cv.OJ)
```

```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  689.1001  685.8030  654.9314  653.7774  666.8890  721.2494  733.6936
## [8] 1066.6499
##
## $k
## [1]      -Inf  11.20965  14.72877  17.88334  23.55203  38.37537  43.02529
## [8] 337.08200
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
plot(cv.OJ)
```




La función cv.tree analiza como afecta la poda de nodos terminales a la precisión del arbol, es util ya que la forma de regularizar un arbol es mediante la poda. ####(Bonus) 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?

```
set.seed(11)
cv.tree.error <- function(){
    errores_m <- NULL
    for(i in 1:5){
        train <- sample(1:1070, 800, replace = F)
        train.data <- OJ[train, ]
```

```
        modeloOJ <- tree(formula = Purchase ~., data = OJ, subset = train)

        sizes <- cv.tree(modeloOJ)$size
        sizes <- sort(sizes)
        errores <- vector(length = length(sizes))
        for(j in sizes){
            if(j==1){
                next
            }
            print(j)
            OJ.j <- prune.tree(modeloOJ, best=j)
            OJ.pred <- predict(OJ.j , OJ[-train,], type = "class")
            tree.conf.matrix <- table(OJ.pred, OJ[-train,1])
            tasa.error <- sum(diag(tree.conf.matrix))/sum(tree.conf.matrix)
            errores[j] <- tasa.error
            print(tasa.error)
        }
        print(errores)
        errores_m <- rbind(errores_m, errores)
    }
    error_v <- vector(length = length(errores_m))
    return(errores_m)

}

errores_m <- cv.tree.error()
```

```
## [1] 2
## [1] 0.7777778
## [1] 3
## [1] 0.7777778
## [1] 4
## [1] 0.7777778
## [1] 5
## [1] 0.7777778
## [1] 6
## [1] 0.7925926
## [1] 7
## [1] 0.7925926
## [1] 8
## [1] 0.7888889
## [1] 0.0000000 0.7777778 0.7777778 0.7777778 0.7777778 0.7925926 0.7925926
## [8] 0.7888889
## [1] 2
## [1] 0.8148148
## [1] 3
## [1] 0.8148148
## [1] 4
## [1] 0.8148148
## [1] 5
## [1] 0.8296296
## [1] 7
## [1] 0.8148148
```

```
## [1] 8
## [1] 0.8333333
## [1] 9
## [1] 0.8333333
## [1] 0.0000000 0.8148148 0.8148148 0.8148148 0.8296296 0.0000000 0.8148148
## [8] 0.8333333 0.8333333
```

```
## Warning in rbind(errores_m, errores): number of columns of result is not a
## multiple of vector length (arg 2)
```

```
## [1] 2
## [1] 0.7777778
## [1] 3
## [1] 0.7777778
## [1] 4
## [1] 0.7777778
## [1] 5
## [1] 0.8074074
## [1] 6
## [1] 0.8
## [1] 7
## [1] 0.8
## [1] 0.0000000 0.7777778 0.7777778 0.7777778 0.8074074 0.8000000 0.8000000
```

```
## Warning in rbind(errores_m, errores): number of columns of result is not a
## multiple of vector length (arg 2)
```

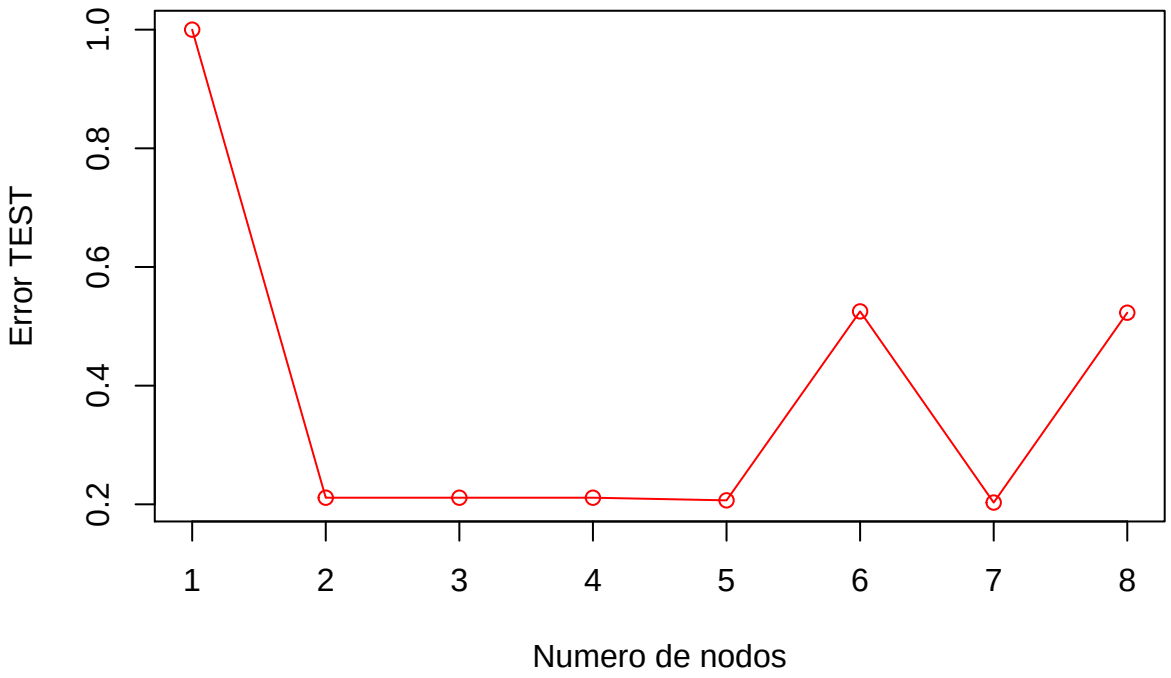
```
## [1] 2
## [1] 0.8111111
## [1] 3
## [1] 0.8111111
## [1] 4
## [1] 0.7777778
## [1] 5
## [1] 0.7777778
## [1] 6
## [1] 0.7814815
## [1] 7
## [1] 0.8148148
## [1] 0.0000000 0.8111111 0.8111111 0.7777778 0.7777778 0.7814815 0.8148148
```

```
## Warning in rbind(errores_m, errores): number of columns of result is not a
## multiple of vector length (arg 2)
```

```
## [1] 2
## [1] 0.762963
## [1] 3
## [1] 0.762963
## [1] 4
## [1] 0.7962963
## [1] 5
## [1] 0.7740741
## [1] 0.0000000 0.7629630 0.7629630 0.7962963 0.7740741
```

```
## Warning in rbind(errores_m, errores): number of columns of result is not a
## multiple of vector length (arg 2)
```

```
errores_v <- apply(errores_m,MARGIN = 2, FUN = mean)
plot(1:length(errores_v), 1 - errores_v, type = "o", col = "red", ylab = "Error TEST", xlab="Numero de n
```




En este caso lo más adecuado es utilizar un arbol de tamaño 5 ya que tiene el menor error de validación cruzada y tambien minimiza el gráfico de la "deviance".