EJERCICIO 1:

Cargamos el paquete ISLR para trabajar con Auto:

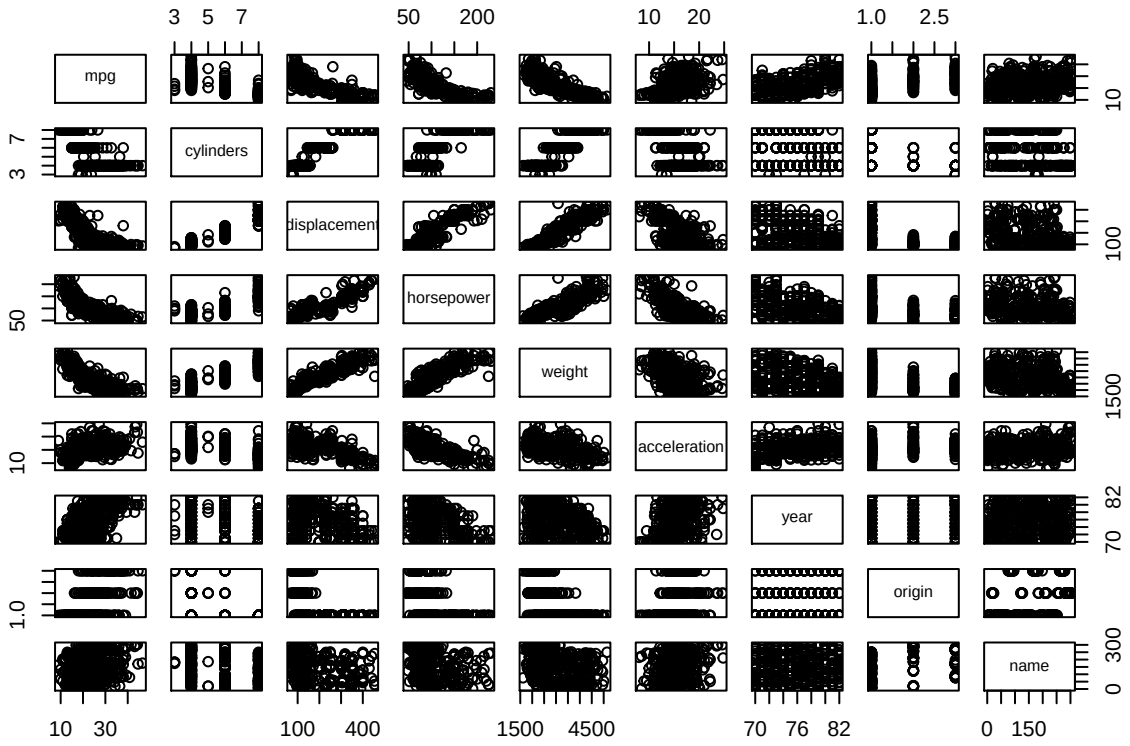
```
library(ISLR)
```

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

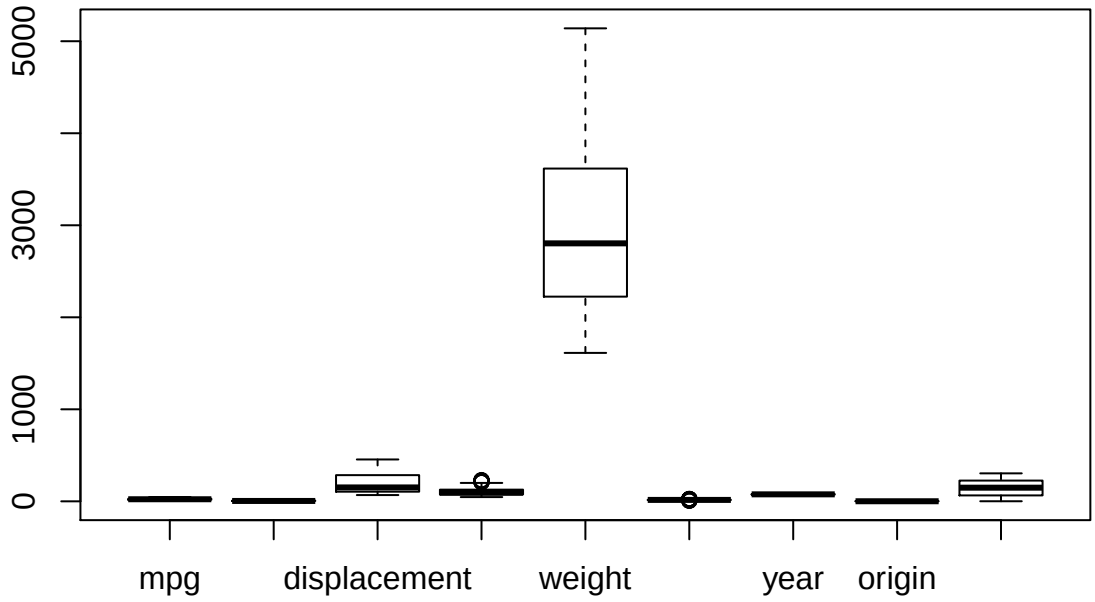
a) Usar las funciones de R pairs() y boxplot() para investigar la dependencia entre mpg y las otras características. ¿Cuáles de las otras características parece más útil para predecir mpg? Justificar la respuesta.

Vamos a visualizar la relación entre las distintas variables de la base Auto.

```
pairs(Auto)
```




```
boxplot(Auto)
```



Tras visualizar ambas gráficas, vemos que las variables origin y name no nos permiten predecir la variable mpg pues en name-mpg tenemos una nube de puntos dispersa en todo el intervalo y en origin-mpg no tenemos una distribución que nos permita establecer relaciones entre ambas. Las otras variables sí parecen útiles para predecir a mpg.

Analizando un poco más estas otras variables útiles, vemos que displacement, horsepowe y weight mantienen cierta dependencia lineal. De modo que podemos considerar sólamente una de ellas, en concreto, vamos a considerar la variable horsepowe ya que parece que nos va a permitir predecir mejor (mirando la tendencia de las gráficas).

Así pues, nos quedamos con las variables cylinders, horsepower, acceleration y year.

b) Seleccionar las variables predictoras que considere más relevantes.

Vamos a quedarnos con las variables cylinders, horsepower, acceleration y year, pues son las que hemos dicho anteriormente que nos interesan.

```
Predictoras = cbind(Auto$mpg, Auto$displacement, Auto$horsepower, Auto$weight, Auto$year)
colnames(Predictoras) = c("mpg", "displacement", "horsepower", "weight", "year")
```

Vemos las 5 primeras variables predictoras:

```
print(Predictoras[c(1,2,3,4,5),])
```

```
##      mpg displacement horsepower weight year
## [1,]  18          307        130   3504   70
## [2,]  15          350        165   3693   70
## [3,]  18          318        150   3436   70
## [4,]  16          304        150   3433   70
## [5,]  17          302        140   3449   70
```

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

Vamos a generar un 20% de los datos totales de forma aleatoria. Una vez tengamos los índices de los datos, vamos a quedarnos con los datos con dichos índices obteniendo las variables predictoras de conjunto test. Los datos con los índices que no hemos considerado para test, los consideramos para train y los almacenamos en Predictoras_train. Hemos tomado índices aleatorios pues no queremos que la muestra de test y de train se vean influenciadas por nuestra distinción de los conjuntos.

```
set.seed(2)
indices_test = sample(nrow(Predictoras), 2*nrow(Predictoras)%/%10, replace=FALSE)
Predictoras_test = Predictoras[indices_test,]
Predictoras_train = Predictoras[-indices_test,]
```

Veamos cómo han quedado los 5 primeros datos para el conjunto test:

```
print(Predictoras_test[c(1,2,3,4,5),])
```

```
##       mpg displacement horsepower weight year
## [1,] 13.0          307        130   4098   72
## [2,] 21.6          121        115   2795   78
## [3,] 17.5          250        110   3520   77
## [4,] 17.0          304        150   3672   72
## [5,] 29.0          135         84   2525   82
```

Veamos cómo han quedado los 5 primeros datos para el conjunto train:

```
print(Predictoras_train[c(1,2,3,4,5),])
```

```
##      mpg displacement horsepower weight year
## [1,]  18          307        130   3504   70
## [2,]  15          350        165   3693   70
## [3,]  17          302        140   3449   70
## [4,]  14          454        220   4354   70
## [5,]  14          440        215   4312   70
```

d) 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).

Obtenemos la media de los valores mpg de Auto. Añadimos una columna con mpg01 a las variables Predictorias train y test según el signo que obtenemos al hacer la diferencia de la mediana con el valor mpg de cada dato.

```
mediana_mpg = median(Auto$mpg)
Predictoras_train_mpg01 = data.frame(Predictoras_train,
                                     mpg01=sign(Predictoras_train[,"mpg"] - mediana_mpg))
Predictoras_test_mpg01 = data.frame(Predictoras_test,
                                    mpg01=sign(Predictoras_test[,"mpg"] - mediana_mpg))
```

Veamos la media para ver que la nueva variable se obtiene correctamente.

```
print(mediana_mpg)
```

```
## [1] 22.75
```

Veamos cómo han quedado los 5 primeros datos para el conjunto test:

```
print(Predictoras_test_mpg01[c(1,2,3,4,5),])
```

```
##    mpg displacement horsepower weight year mpg01
## 1 13.0          307        130   4098   72    -1
## 2 21.6          121        115   2795   78    -1
## 3 17.5          250        110   3520   77    -1
## 4 17.0          304        150   3672   72    -1
## 5 29.0          135         84   2525   82     1
```

Veamos cómo han quedado los 5 primeros datos para el conjunto train:

```
print(Predictoras_train_mpg01[c(1,2,3,4,5),])
```

```
##   mpg displacement horsepower weight year mpg01
## 1  18          307        130   3504   70    -1
## 2  15          350        165   3693   70    -1
## 3  17          302        140   3449   70    -1
## 4  14          454        220   4354   70    -1
## 5  14          440        215   4312   70    -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.

Necesito que los valores de mpg01 estén comprendido entre 0 y 1. De modo que vamos a reevaluar los datos con variable mpg01 = -1 como mpg01= 0.

```
for(i in 1:nrow(Predictoras_test_mpg01)){
 if((Predictoras_test_mpg01[i,"mpg01"]) == -1)
     Predictoras_test_mpg01[i,"mpg01"] = 0
}
for(i in 1:nrow(Predictoras_train_mpg01)){
 if((Predictoras_train_mpg01[i,"mpg01"]) == -1)
     Predictoras_train_mpg01[i,"mpg01"] = 0
}
```

Hacemos la regresión logística con el método glm usando las variables cylinders, horsepowe, acceleration y year.

```
ajuste_rlog <- glm(mpg01 ~ displacement + horsepower + weight + year,
                   data= Predictoras_train_mpg01, family=binomial)
```

Ahora usamos el método predict para obtenerlas probabilidades y poder obtener la variable mpg01 que predice el modelo.

```
probabilidades = predict(ajuste_rlog, Predictoras_test_mpg01, type="response")
probabilidades01 = rep(1,dim(Predictoras_test_mpg01)[1])
probabilidades01[probabilidades<0.5] = 0
```

Obtenemos la matriz de confusión y obtenemos el error que viene dado como suma de las predicciones erroneas de 0 y 1 dividido por el numero total de datos.

```
matriz_confusion = table(probabilidades01, Predictoras_test_mpg01$mpg01)
print(matriz_confusion)
```

```
##
## probabilidades01  0  1
##                0 30  1
##                1  5 42
```

```
error = (matriz_confusion[1,2] + matriz_confusion[2,1])/sum(matriz_confusion)
print(error)
```

```
## [1] 0.07692308
```

Al tener un error tan bajo (0.0769), concluimos que las variables que hemos seleccionado como predictoras son bastante buenas, así como el modelo.

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) (1 punto)

Cargamos los paquetes class y e1071 de R.

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

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

```
set.seed(2)
```

Para usar el algormitmo K-NN tenemos que normalizar los datos. Vamos a ello: Normalizamos los datos iniciales y nos quedamos en test con los los datos normalizados de los indices que teníamos y en train con los restantes.

```
columnas=  dim(Predictoras_train_mpg01)[2]
datos_normalizados = scale(rbind(Predictoras_test_mpg01[,-columnas],
                                 Predictoras_train_mpg01[,-columnas]))
test_normalizada = datos_normalizados[1:dim(Predictoras_test_mpg01)[1],]
train_normalizada = datos_normalizados[(dim(Predictoras_test_mpg01)[1] +1):
                                       dim(datos_normalizados)[1],]
```

Ahora nos quedamos con el vector mpg01 de los conjuntos train y test. Los combinamos en mpg01_ambos.

```
mpg01_train = Predictoras_train_mpg01[,columnas]
mpg01_test = Predictoras_test_mpg01[,columnas]
mpg01_ambos = c(mpg01_train, mpg01_test)
```

Veamos cuál es el mejor valor de k para ajustar los datos. Usaremos tune.knn:

```
posibles_k <- tune.knn(datos_normalizados, as.factor(mpg01_ambos), k=1:10,
                       tune_control = tune.control(sampling = "cross"))
summary(posibles_k)
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  4
##
## - best performance: 0.4132051
##
## - Detailed performance results:
##     k     error dispersion
## 1   1 0.5053205 0.08813063
## 2   2 0.4648077 0.09083708
## 3   3 0.4544231 0.12007880
## 4   4 0.4132051 0.11226521
## 5   5 0.4392308 0.10749448
## 6   6 0.4440385 0.12589008
## 7   7 0.4543590 0.09465103
## 8   8 0.4393590 0.10197131
## 9   9 0.4265385 0.09004031
## 10 10 0.4139744 0.10990212
```

Vemos que el valor de k que mejor ajusta los datos es k=4, de modo que usaremos este valor para aplicar KNN. Al igual que hacíamos antes, vemos la matriz de confunsión o obtenemos el error de test.

```
predicciones = knn(train_normalizada, test_normalizada, mpg01_train, k=4)
matriz_confusion_knn = table(predicciones, mpg01_test)
print(matriz_confusion_knn)
```

```
##             mpg01_test
## predicciones  0  1
##            0 33  0
##            1  2 43
```

```
errorknn = ((matriz_confusion_knn[1,2] + matriz_confusion_knn[2,1])/sum(matriz_confusion_knn))
print(errorknn)
```

```
## [1] 0.02564103
```

Obtenemos un error de 0.0256. Vemos que ha dismunuido con respecto al modelo que hacíamos en el apartado anterior.

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

Haremos uso de la función rocplot siguiente:

```
library(ROCR)
```

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

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

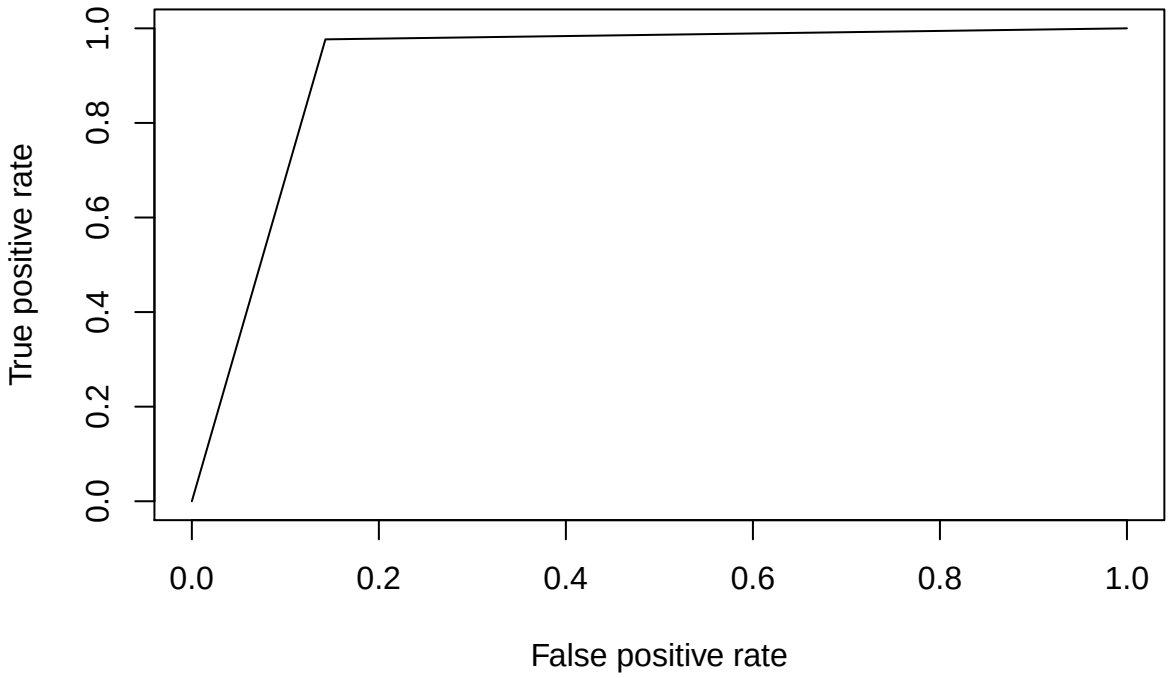
```
## Warning: package 'gplots' was built under R version 3.2.5
```

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

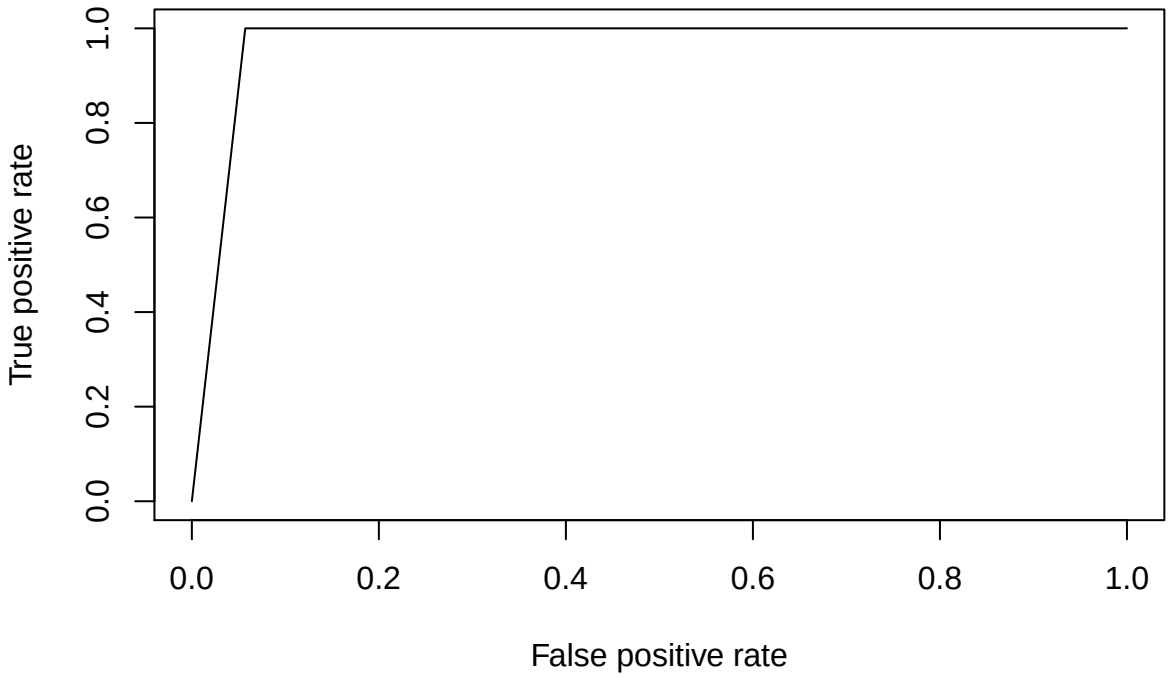
```
## The following object is masked from 'package:stats':
##
##     lowess
```

```
rocplot=function(pred, truth, ...){ #pag 365
  predob = prediction (pred, truth)
  perf = performance (predob, "tpr", "fpr")
  plot(perf, ...)
}

rocplot(probabilidades01, Predictoras_test_mpg01$mpg01)
```




```
rocplot(as.numeric(predicciones), mpg01_test)
```



Estas son las curvas ROC para ambos modelos. El primero para regresión logística y el segundo para KNN con k=4. //EXPLICAR GRAFICAS??????????????????????????????

//FALTAN BONUS.

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.

Al igual que hacíamos en el ejercicio anterior, creamos una nueva variable que nos dice que la tasa de criminalidad de cada dato está por encima (1) o por debajo de la mediana (-1). La añadimos a nuestro conjunto de datos y particionamos en conjunto test y train.

```
library(randomForest)
```

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

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

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

```
library(MASS)
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
```

```
set.seed(2)
mediana_crim = median(Boston$crim)
Boston_crim01 = data.frame(Boston, crim01=sign(Boston[,"crim"] - mediana_crim))

indices_train = sample(nrow(Boston_crim01), 8*nrow(Boston_crim01)%/%10, replace=FALSE)
Boston_train_crim01 = Boston_crim01[indices_train,]
Boston_test_crim01 = Boston_crim01[-indices_train,]
```

a) 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. (1 punto)

Nos quedamos con los datos de Boston sin las variables crim y crim01 y con unicamente variable crim, que es la que tratamos de predecir. alpha=1 indica que estamos realizando un modelo LASSO. Nos quedamos con aquellas variables que superan un umbral prefijado, en nuestro caso lo establecemos a 0.1.

```
Boston_sinCrims = model.matrix(crim~.-crim-crim01, data=Boston_train_crim01)[,-1]
Boston_solocrim = as.matrix(Boston_train_crim01[,1])

cv.lasso = cv.glmnet(x = Boston_sinCrims, y = Boston_solocrim, alpha=1)
lasso = glmnet(x = Boston_sinCrims, y = Boston_solocrim, alpha=1)
lasso.coef = predict(lasso, type = "coefficients", s = cv.lasso$lambda.min)[1:14,]
lasso.coef
```

```
##  (Intercept)           zn        indus         chas          nox
##  7.445732058  0.031911359 -0.070184303 -1.394068679 -3.243579158
##           rm          age          dis          rad          tax
##  0.086417685  0.000000000 -0.658473571  0.522833052  0.000000000
##      ptratio        black        lstat         medv
## -0.078252987 -0.001525618  0.090634608 -0.164479989
```

```
umbral = 0.1
predictoras_umbral <- lasso.coef[abs(lasso.coef) > umbral ]
predictoras_umbral
```

```
## (Intercept)        chas         nox         dis         rad        medv
##   7.4457321  -1.3940687  -3.2435792  -0.6584736   0.5228331  -0.1644800
```

```
seleccionadas = names(predictoras_umbral[2:length(predictoras_umbral)])
```

Hemos trabajado con el conjunto de entrenamiento. Podríamos haber considerado todo el conjunto de datos Boston simplemente cambiando BostonTrainCrim01 por BostonCrim01, pero nos ha parecido más adecuado hacer el particionamiento desde un principio.

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

Usamos el modelo ridge regression con las variables que teníamos seleccionadas anteriormente. A diferencia de antes, alpha=0. Finalmente calculamos el error:

```
rr = glmnet(x = as.matrix(Boston_train_crim01[,c(seleccionadas)]), y = Boston_solocrim, alpha=0)
ridge.pred <- predict(rr, newx = as.matrix(Boston_test_crim01[,c(seleccionadas)]),
                      s=cv.lasso$lambda.min, type = "response")
error <- mean((ridge.pred - Boston_test_crim01[,1])^2)
```

Obtenemos un eror de 46.7%. Es excesivamente grande, lo que nos lleva a pensar que tenemos underfitting.

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.

Estas variables ya las definimos al comienzo del ejercicios y las añadimos como una variable nueva en BostonTestCrim01 y BostonTrainCrim01.

```
set.seed(2)
library("e1071")
BostonSVM <- svm(crim01~., data=Boston_train_crim01, kernel="linear",
                 cost=1, scale = FALSE)
crime.pred <- predict(BostonSVM, Boston_train_crim01)
crime.pred.class <- 2*(crime.pred > 0)-1
matriz = table(predict=crime.pred.class, truth=Boston_train_crim01$crim01)
error = ((matriz[1,2] + matriz[2,1])/sum(matriz))
print(error)
```

```
## [1] 0.1825
```

Obtenemos un error de 0.1825.

Hemos probado con kernel polynomial pero tarda demasiado y obtenemos un error de 0.395. Probamos con un núcleo radial:

```
BostonSVM2 <- svm(crim01~., data=Boston_train_crim01, kernel="radial",
                  cost=1, scale = FALSE)
crime.pred2 <- predict(BostonSVM2, Boston_train_crim01)
crime.pred2.class <- 2*(crime.pred2 > 0)-1
matriz2 = table(predict=crime.pred2.class, truth=Boston_train_crim01$crim01)
error2 = ((matriz2[1,2] + matriz2[2,1])/sum(matriz2))
print(error2)
```

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

Obtenemos un error de 0.

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%).

Vamos a cargar la base de datos y las librerías que usaremos para los próximos apartados. Ahora, al igual que hacíamos en el ejercicio1, consideramos un 80% de los datos de Boston y los almacenamos en BostonTrain. Los datos restantes los guardamos en BostonTest

```
library(randomForest)
library(MASS)
set.seed(2)
indices_train = sample(1:nrow(Boston),8*nrow(Boston)%/%10)
Boston_train = Boston[indices_train,]
Boston_test = Boston[-indices_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 parámetros usados. Calcular el error del test. (1 punto)

Vemos que efectivamente la variable medv está en nuestro conjunto de datos y ajustamos el modelo de regresión usando bagging.

Bagging es un caso especial de Random Forest, donde se usa m=p. De modo que podremos usar randromForest para ambas técnicas.

Consideraremos los índices que tenemos en indices_train del conjunto Boston (podríamos haber cogido directamente data=BostonTrain y no especificar subset). En mtry tenemos que indicar la cantidad de variables que tenemos como predictoras. Nuestro conjunto tiene 14 variables, pero medv la estamos usando como salida. Calculamos el error del test:

```
set.seed(2)
names(Boston)
```

```
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "black"   "lstat"   "medv"
```

```
boston_bagging = randomForest(medv~.,data=Boston, subset=indices_train,mtry=13,ntree=30,importance =TRU
predict.bag = predict (boston_bagging , Boston_test)
mean((predict.bag -Boston_test$medv)^2)
```

```
## [1] 6.717474
```

Tenemos un error de 6.71

3. 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.

Usando Random Forest para modelos de regresión, en mtry tenemos m=p/3 como el valor que se establece por defecto. Nosotros lo especificamos para hacerlo visible. Calculamos el error del test del mismo modo:

```
set.seed(2)
boston_rf= randomForest(medv~.,data=Boston , subset=indices_train , mtry=13/3, ntree=30, importance =TRU
predict.rf = predict(boston_rf , Boston_test)
mean((predict.rf-Boston_test$medv)^2)
```

```
## [1] 6.324365
```

Tenemos un error de 6.32

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.

En este caso consideramos una gran cantidad de árboles, aunque no llega a tener el valor por defecto (5000) pues consume más tiempo y no llega a proporcionar grandes mejoras de error. Establecemos una profundidad para los árboles de 8 niveles.

```
library(gbm)
```

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

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

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

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

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

```
## Loaded gbm 2.1.1
```

```
set.seed(2)
boston_boosting = gbm(medv~.,data=Boston_train, distribution = "gaussian", n.trees = 4000, interaction.
predict.boost = predict(boston_boosting, Boston_test, n.trees = 4000)
mean((predict.boost-Boston_test$medv)^2)
```

```
## [1] 6.315222
```

El error es de 6.31. En comparación con los erroes obtenimos con bagging y Randon Forest, vemos que este método no merece mucho la pena, pues consigue errores similares sacrificando tiempo al producir una gran cantidad de árboles con una profundidad considerable.

EJERCICIO4: 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 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).

En primer lugar, para familiarizarnos con el conjunto de datos OJ, vamos a ver las variables de las que dispone. Hacemos uso de names.

```
library(ISLR)
library(tree)
```

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

```
names(OJ)
```

```
##  [1] "Purchase"       "WeekofPurchase" "StoreID"        "PriceCH"
##  [5] "PriceMM"        "DiscCH"         "DiscMM"         "SpecialCH"
##  [9] "SpecialMM"      "LoyalCH"        "SalePriceMM"    "SalePriceCH"
## [13] "PriceDiff"      "Store7"         "PctDiscMM"      "PctDiscCH"
## [17] "ListPriceDiff"  "STORE"
```

```
set.seed(1013)
```

Vamos a ajustar el arbol usando Purchase como variable respuesta. Tomo 800 indices de observaciones aleatorias y nos quedamos con estas observaciones para el conjunto train. El resto de observaciones lo introducimos en el conjunto test.

```
indices_train800 = sample(nrow(OJ),800)
trainOJ = OJ[indices_train800,]
testOJ = OJ[-indices_train800,]
treeOJ = tree(Purchase~. ,data = trainOJ)
```

En el siguiente apartado procedemos a analizar los resultados.

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. (0.5 puntos)

Veamos el resumen estadístico del árbol:

```
summary(treeOJ)
```

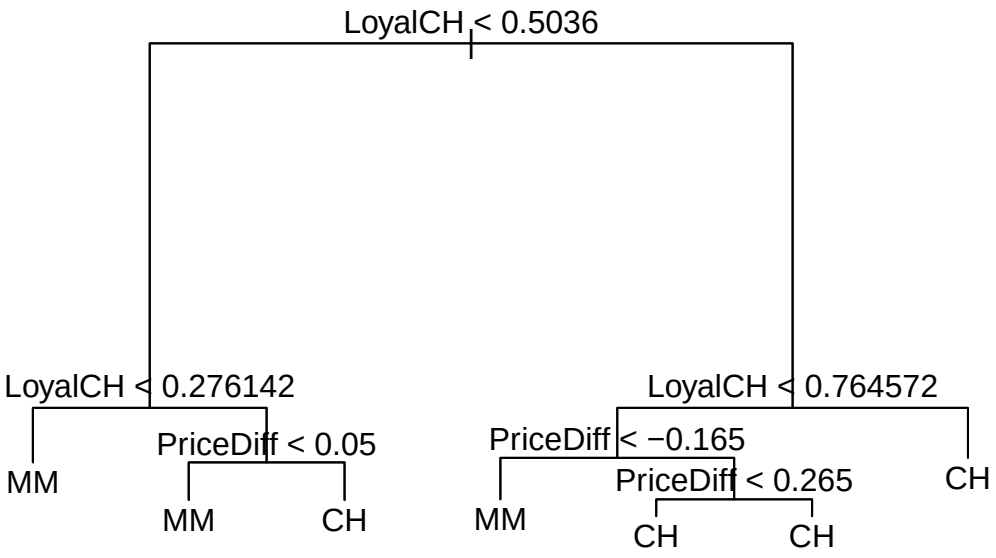
```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = trainOJ)
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff"
## Number of terminal nodes:  7
## Residual mean deviance:  0.7517 = 596.1 / 793
## Misclassification error rate: 0.155 = 124 / 800
```

Vemos que para la construcción del árbol se han usado las variables "LoyalCH", "PriceDiff", "SpecialCH" y "ListPriceDiff". Se han generado 8 nodos terminales. La media de la desviación residual es 0.7517 y la tasa de error de traning es de 0.155.

3. Crear un dibujo del árbol e interpretar los resultados (0.5 puntos)

Veamos el dibujo del árbol:

```
plot(treeOJ)
text(treeOJ, pretty = 0)
```



Vemos que el árbol que comienza dividiendo en función de si el valor de LoyalCH(la fidelidad del usuario con respecto a el zumo de la empresa Citrus Hill) es mayor o menor que 0.5036. A continuación se vuelve a subdividir segun "LoyalCH": si la fidelidad inicial era menor que 0.5036 vuelve a dividir en función de si su fidelidad es menor o mayor que 0.276142, mientras que si su fidelidad era mayor que 0.5036, divide en función de si LoyalCH es menor o mayor que 0.764572. Volvemos a ver subdivisiones en el árbol que se analizan del mismo modo a como ya hemos visto.

Ahora vamos a ver las reglas de clasificación más relevantes. Serán aquellas que marcan una subdivisión entre MM(Minute Maid) y CH(Citrus Hill): LoyalCH <0.5036, LoyalCH <0.276142, LoyalCH <0.764572, PriceDiff <0.165 y PriceDiff <0.05.

Mientras que las menos relevantes son: PriceDiff <0.265.

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 del test? ¿Cuál es la precisión del test? (1 punto)

```
probabilidades = predict(treeOJ, testOJ, type="class")
matriz_confusion = table(testOJ$Purchase, probabilidades)
print(matriz_confusion)
```

```
##     probabilidades
##       CH  MM
##   CH 152  19
```

```
##   MM  32  67
```

```
error = (matriz_confusion[1,2] + matriz_confusion[2,1])/sum(matriz_confusion)
print(error)
```

```
## [1] 0.1888889
```

Tenemos un error de test de 18.88% aprox., luego tendremos una precisión de test de 81.12% aprox..

5. Aplicar la función cv.tree() al conjunto de "training" y determinar el tamaño óptimo del árbol. ¿Qué hace cv.tree? (0.5 puntos)

```
cvtreeOJ = cv.tree(treeOJ, FUN= prune.tree)
cvtreeOJ
```

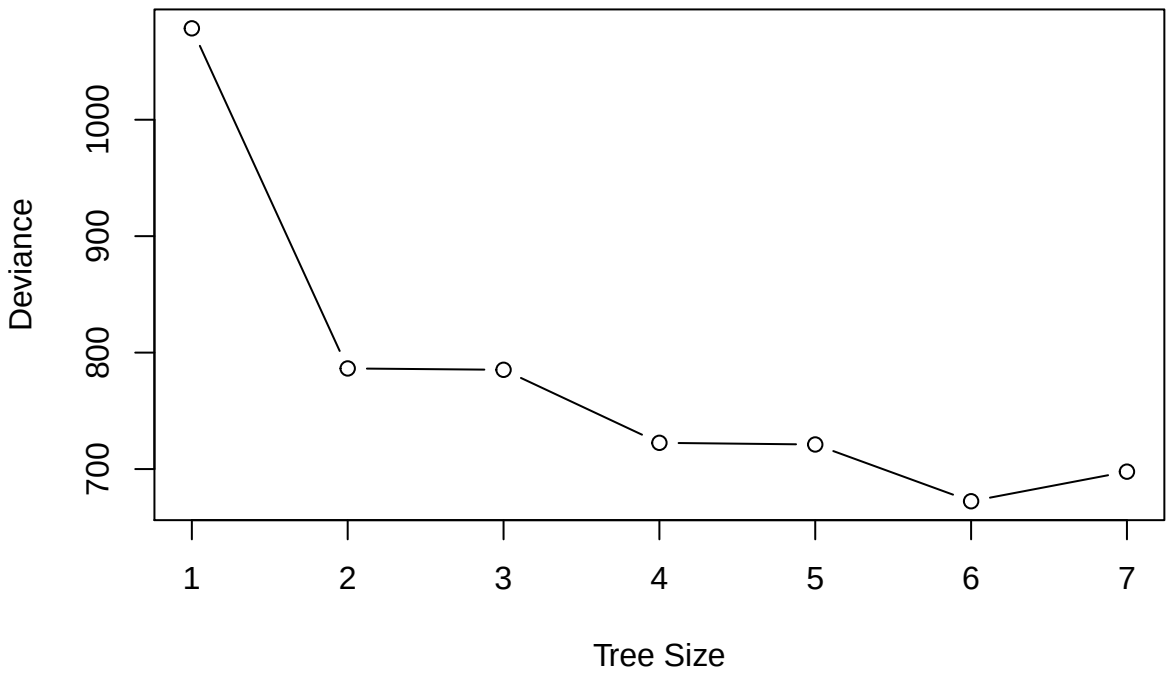
```
## $size
## [1] 7 6 5 4 3 2 1
##
## $dev
## [1]  697.7742  672.3676  721.1061  722.5296  785.2427  786.4040 1078.4624
##
## $k
## [1]      -Inf  14.33140  31.91234  35.17952  42.37864  46.23075 309.00727
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

La función cv.tree ejecuta un experimento de validación cruzada K veces para encontrar la desviación o el número de errores de clasificación.

El menor error de validación cruzada viene dado por 663, que se corresponde con el árbol que tiene 6 nodos terminales. Luego estos serán los tamaños óptimos del árbol.

Bonus-4 (1 punto). 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?

```
plot(cvtreeOJ$size, cvtreeOJ$dev, type = "b", xlab = "Tree Size", ylab = "Deviance")
```



Podemos ver que el tamaño 6 de árbol nos proporciona la tasa más pequeña de error de clasificación por validación cruzada.