Trabajo de prácticas 3

Miguel López Campos

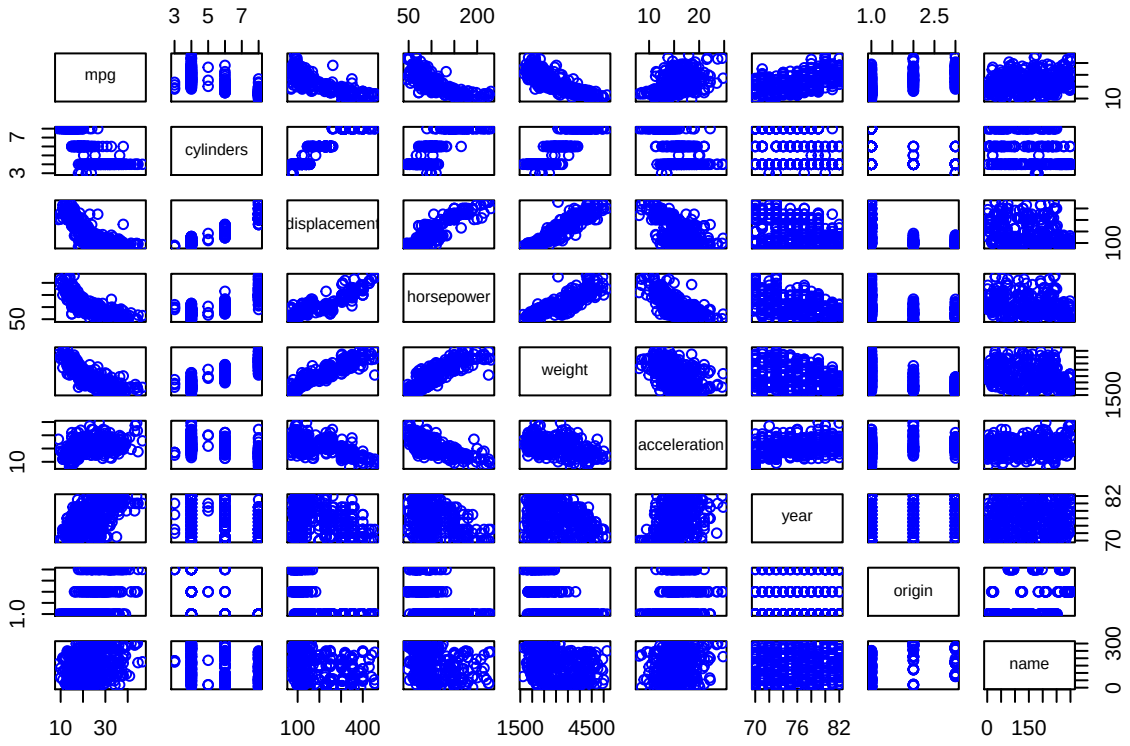
4 de mayo de 2016

Ejercicio 1

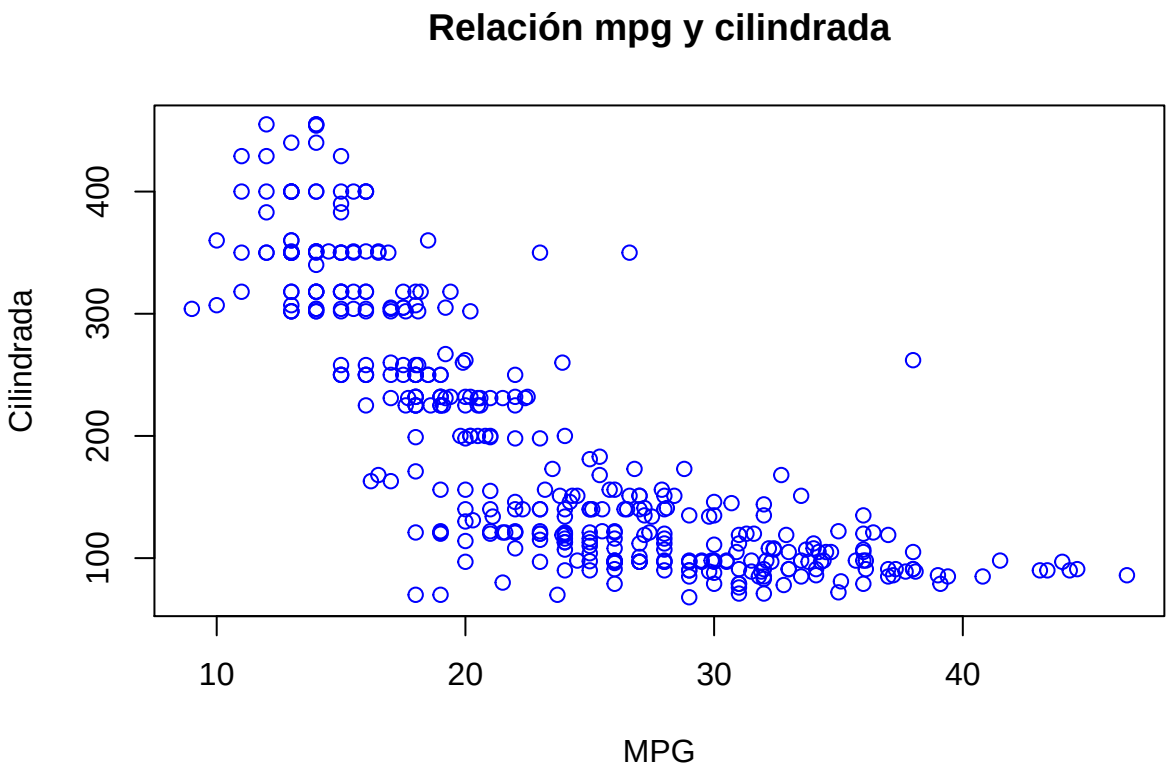
Apartados a y b

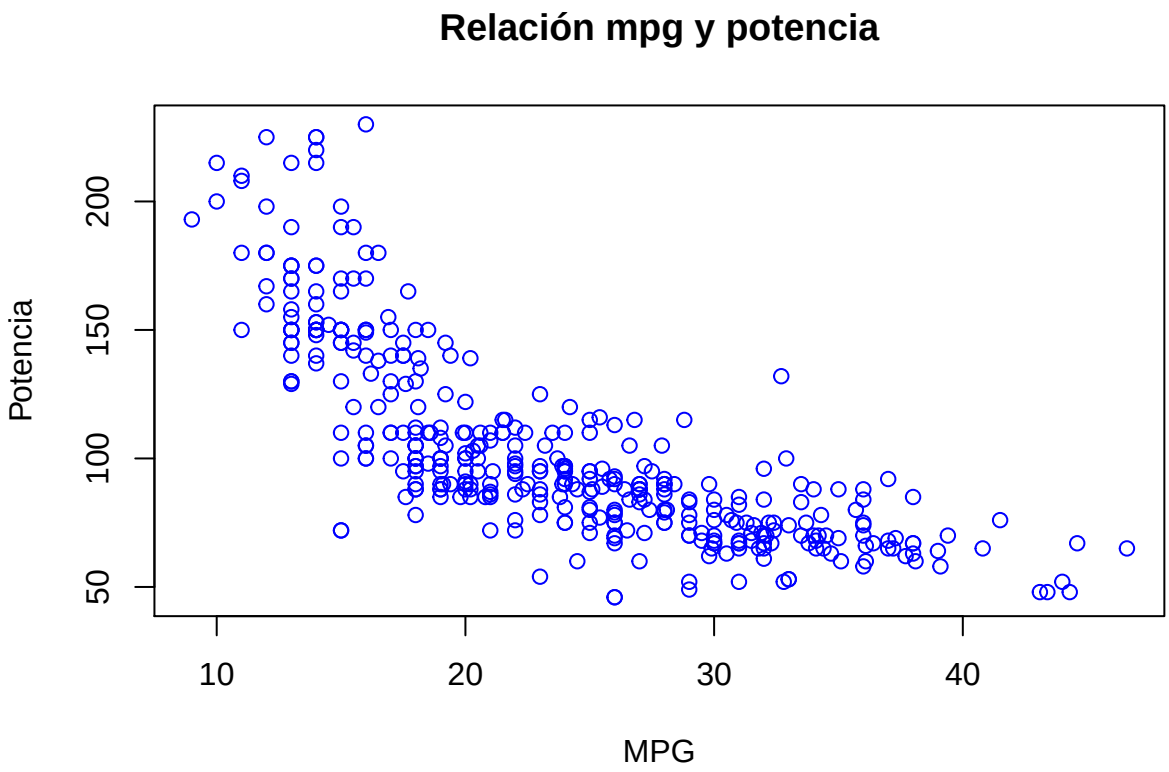
Para trabajar con la base de datos Auto solo tenemos que importar la librería ISLR y llamar al comando Auto que directamente nos devuelve el data frame correspondiente a la base de datos. Yo lo introduciré en una variable llamada 'auto'.

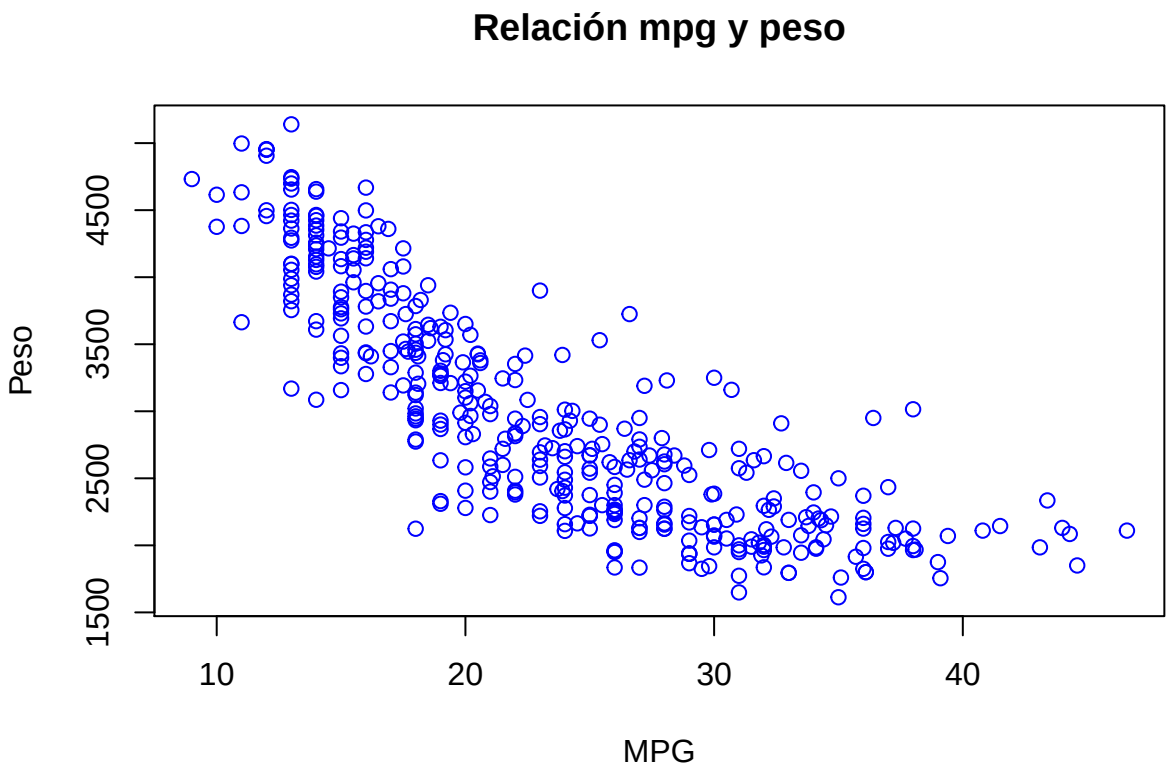
```
##Ejercicio 1
##Apartados a y b
auto <- Auto
pairs(auto, col="blue")
```



Como podemos ver, la característica mpg parece tener cierta relación lineal con el 'displacement' (cilindrada), 'horsepower'(potencia en caballos de vapor) y 'weight' (peso), ya que en las tres gráficas parece apreciarse que cuando el valor de una característica aumenta el de la otra disminuye. En las siguientes gráficas se puede observar mejor.


Relación mpg y cilindrada
Cilindrada
100
200
300
400
10
20
30
40
MPG


Relación mpg y potencia
Potencia
50
100
150
200
10
20
30
40
MPG



Pensándolo de otra manera, dejando aparte las gráficas, tiene sentido decir que el consumo depende de estos tres factores, a si que podemos "aceptar" estas relaciones.

Apartados c y d

Hago un nuevo data frame con las características elegidas y posteriormente realizo el etiquetado según el enunciado: para Knn, si el MPG está por encima de la mediana le asigno etiqueta +1 y si está por debajo, etiqueta -1. Para regresión logística etiquetaré con 0 y 1 como a continuación hago. Para extraer una submuestra empleo la función sample, cogiendo el 80% de índices del conjunto de datos. Los restantes serán el test set.

```
###Apartados c y d
##Hago el etiquetado (0,1 para LR y -1 y +1 para knn)
set.seed(1)
mediana <- median(auto$mpg)
datos <- data.frame(auto$displacement, auto$weight, auto$horsepower)
training_ind <- sample(x=(1:nrow(datos)), size=0.8*nrow(datos))
training <- datos[training_ind,]
test <- datos[-training_ind,]
label_training <- auto$mpg[training_ind]
label_test <- auto$mpg[-training_ind]
positivos <- label_training>mediana
label_training[positivos] <- 1
label_training[!positivos] <- 0
positivos <- label_test>mediana
```

```
label_test[positivos] <- 1
label_test[!positivos] <- 0
```

A continuación aplico regresión logística. Para aplicarla uso la función glm. Esta función lo que hace es ajustar modelos lineales generalizados. Para indicarle a R que vamos a aplicar regresión logística lo que hacemos es introducirle el argumento 'family=binomial'.

Lo que hago después es usar la función predict para hacer la 'predicción' del conjunto de datos de test. A predict le meto como argumento el modelo ajustado con la regresión logística, el data set de test y el parámetro type tenemos que ajustarlo a "response" para que nos devuelva probabilidades. Probs, por lo tanto, será la probabilidad de cada uno de los datos de test. Si la probabilidad es mayor que 0.5, consideraremos que su etiqueta es 1, es decir, consumo bajo. Si es menor que 0.5, entonces consideraremos que será consumo alto.

```
##Aplico glm (regresión logística)
set.seed(1)

model <- glm(formula=label_training~.,family=binomial,data=training)
summary(model)
```

```
##
## Call:
## glm(formula = label_training ~ ., family = binomial, data = training)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.3586  -0.3148  -0.0034   0.3702   3.3607
##
## Coefficients:
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)       11.7934871  1.8081615   6.522 6.92e-11 ***
## auto.displacement -0.0124994  0.0057158  -2.187  0.02876 *
## auto.weight       -0.0017599  0.0007688  -2.289  0.02206 *
## auto.horsepower   -0.0495878  0.0151868  -3.265  0.00109 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 433.65  on 312  degrees of freedom
## Residual deviance: 173.54  on 309  degrees of freedom
## AIC: 181.54
##
## Number of Fisher Scoring iterations: 7
```

```
##Con predict calculo las probabilidades
probs <- predict(model,test,type="response")
##Hago el error de clasificación
label_prediction <- probs
label_prediction[label_prediction>=0.5] <- 1
label_prediction[label_prediction<0.5] <- 0
error <- label_test!=label_prediction
error <- error[error==TRUE]
error <- length(error)/length(label_test)
error
```

```
## [1] 0.08860759
```

Como vemos el error que es de un 8%.

Ahora aplicaremos k-nn. Este algoritmo es un algoritmo de aprendizaje por semejanza. Lo que hace es calcular la distancia euclídea entre los datos de training y el que queremos predecir, y los k vecinos más cercanos dirán por voto mayoritario a qué clase pertenece el vector de características que queremos predecir. Para aplicar k-nn usaremos la función knn de la librería 'class'. Antes de emplearlo tenemos que normalizar los datos. Para normalizar los datos emplearé la fución normalizar, implementada por mí:

```
#Función que normaliza un conjunto de datos
normalizar <- function(datos)
{
  for(i in 1:ncol(datos))
  {
    minimo <- min(datos[,i], na.rm=TRUE)
    maximo <- max(datos[,i], na.rm=TRUE)

    if(minimo == 0 && maximo==0){datos[,i]<-0}else
    {
      datos[,i] <- (datos[,i]-minimo)/(maximo-minimo)
    }
  }

  return(datos)
}
```

A continuación normalizo los datos. Después particiono los datos, dejando los mismos conjuntos de training y test que cuando he aplicado regresión logística. Para ello uso los mismos índices que he usado anteriormente.

Para aplicar knn usamos la función knn de la librería "class". A esta función le introducimos el training dataset y el test dataset como argumentos, además del valor de k y el vector de etiquetas de training. La función nos devolverá un vector que será la clasificación que el algoritmo ha predicho.

En este caso, como ya no estamos prediciendo probabilidades, las etiquetas serán -1 (en el caso de que el consumo sea alto) y +1 (en el caso de que el consumo sea bajo).

```
##A continuación aplicamos knn
set.seed(1)

datos_normalizados <- normalizar(datos)
##
##
training_knn <- datos_normalizados[training_ind,]
test_knn <- datos_normalizados[-training_ind,]
label_training_knn <- label_training
label_training_knn[label_training_knn==0] <- -1
prediction <- knn(training_knn, test_knn, cl=label_training_knn, k=3)
##########Etiquetas reales del conjunto de test
label_test_knn <- label_test
label_test_knn[label_test_knn==0] <- -1
##########Calculo el error
error <- label_test_knn[label_test_knn!=prediction]
error <- length(error)/length(label_test_knn)
##############################
error
```

```
## [1] 0.1139241
```

En este caso tenemos un error de un 11%.

Para ver cual es el valor de k que mejor ajusta los datos del knn, usaremos la función tune.knn de la biblioteca e1071 . Para usarla primero la variable dependiente tenemos que cambiarla por tipos de datos lógicos, para que funcione la función. Después aplico la función indicándole que queremos una cross validation (validación cruzada) y que la haga 10 veces. El parámetro k variará entre 1 y 20.

```
##Aplicamos tune.knn para ver que k sería el óptimo para nuestro caso
##las etiquetas deben ser booleanos
label_training_knn[label_training_knn==-1] <- 0
label_training_knn[label_training_knn==1] <- 1

knn.cross <- tune.knn(x = training_knn, y = as.logical(label_training_knn), k = 1:20,
                      tunecontrol=tune.control(sampling = "cross"), cross=10)
```

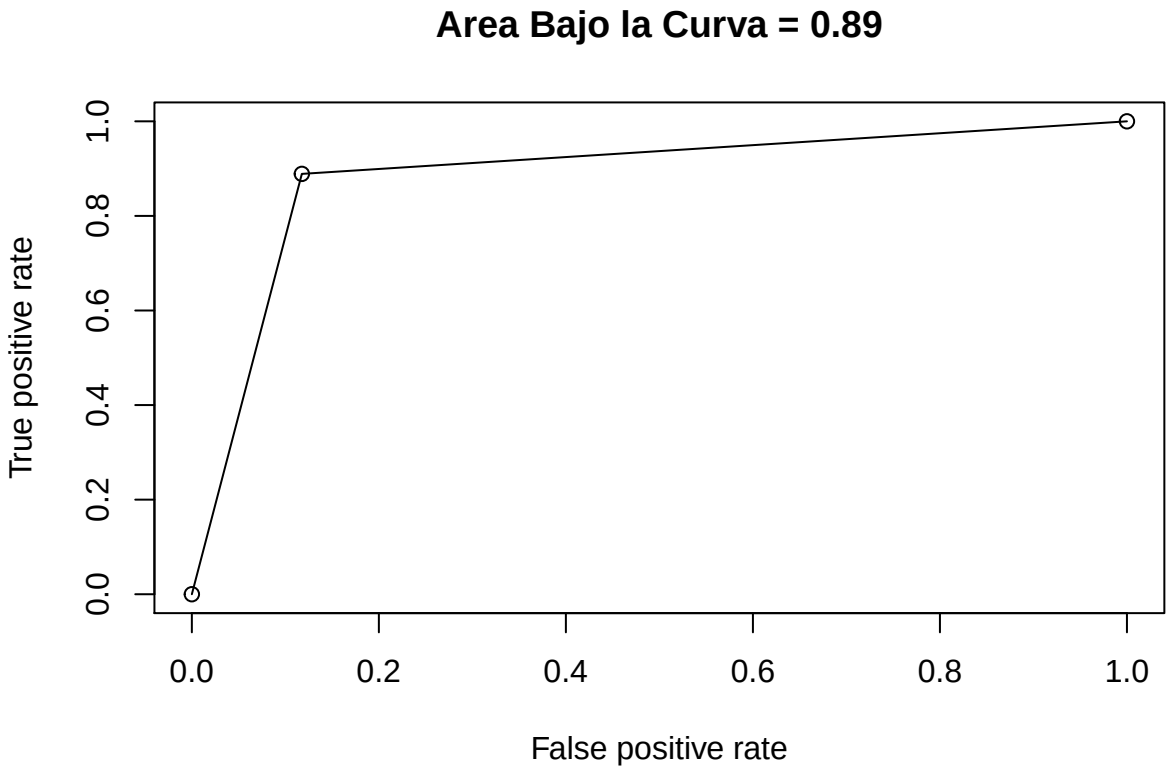
Resumen de tune.knn:

```
summary(knn.cross)
```

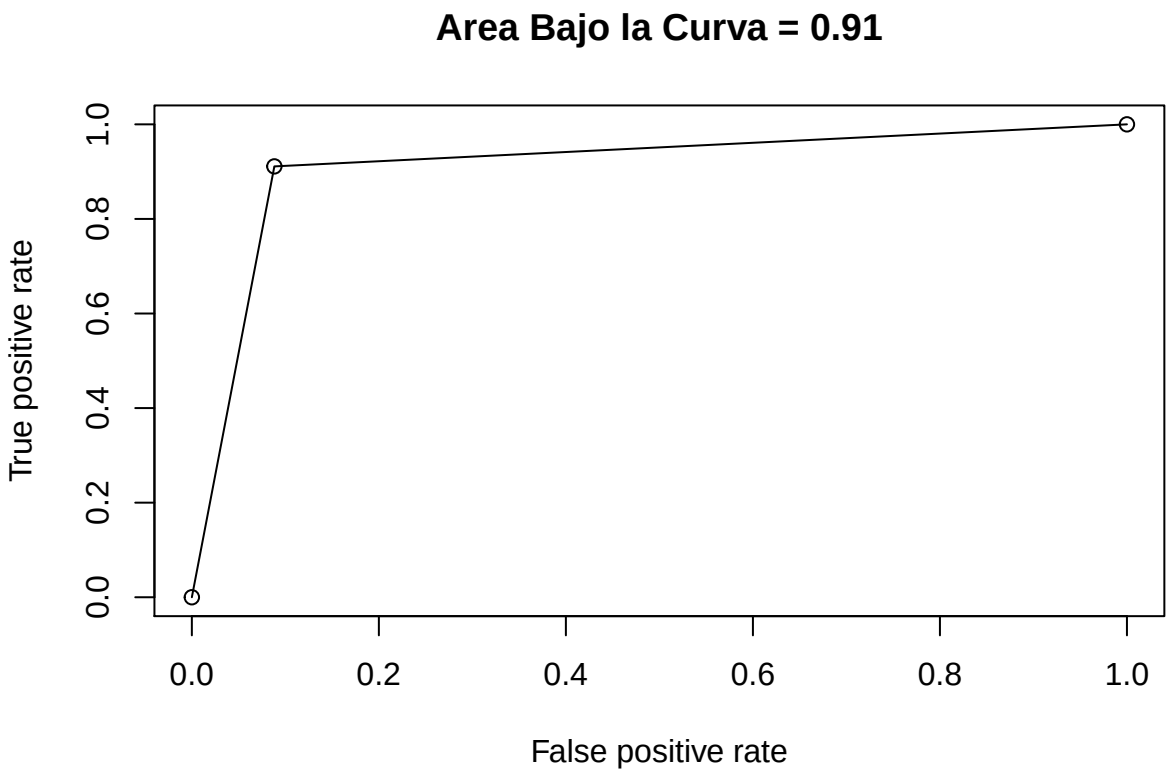
```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  5
##
## - best performance: 0.09919355
##
## - Detailed performance results:
##     k      error dispersion
## 1   1 0.12752016 0.09311525
## 2   2 0.13750000 0.06659536
## 3   3 0.11844758 0.07782834
## 4   4 0.10241935 0.09727483
## 5   5 0.09919355 0.08265610
## 6   6 0.10564516 0.09274169
## 7   7 0.10241935 0.08317900
## 8   8 0.11532258 0.07816268
## 9   9 0.11219758 0.08116718
## 10 10 0.11219758 0.07211565
## 11 11 0.10897177 0.07358863
## 12 12 0.11219758 0.07525389
## 13 13 0.11219758 0.07525389
## 14 14 0.11219758 0.07525389
## 15 15 0.11219758 0.08667798
## 16 16 0.11209677 0.08390559
## 17 17 0.12157258 0.08577070
## 18 18 0.12157258 0.08577070
## 19 19 0.11844758 0.08493205
## 20 20 0.11844758 0.08493205
```

Ahora dibujaremos las curvas ROCR

Para k-nn:



Para regresión logística:



El principal dato estadístico de las curvas es el área bajo la circunferencia (AUC). Este área representa la probabilidad de que un clasificador puntuará una instancia positiva elegida aleatoriamente más alta que una negativa. Por lo tanto, como podemos ver, en el modelo de regresión logística este área es más grande y por lo tanto la probabilidad es mayor.

Apartado e (Bonus)

En este apartado calcularemos por validación cruzada el error. Cogeremos 5 particiones, por lo que en cada partición se cogerá como training el 80% de los datos y como test el 20% restante.

Primero haremos la regresión logística.

```
##Apartado e (Bonus)

#Primero hacemos validación cruzada para regresión logística
set.seed(1)

probs <- rep(1,nrow(Auto))
errores <- vector()

for(i in 1:5)
{
    ##Saco la submuestra de test (20%)
    test <- sample(x=1:nrow(datos), size=(0.2*nrow(datos)), prob=probs)

    ##Pongo probs a 0 del subconjunto cogido para que no se vuelva a repetir
```

```
    probs[test] <- 0

    mediana <- median(Auto$mpg)
    label <- Auto$mpg
    positivos <- label>=mediana
    label[positivos] <- 1
    label[!positivos] <- 0

    label_test <- label[test]
    label_training <- label[-test]


    training_set <- datos[-test,]
    test_set <- datos[test,]

    ##Aplicamos ahora la función glm sobre el training data set
    model <- glm(formula=label_training ~ ., family=binomial, data=training_set)
    model_probs <- predict(model,test_set,type="response")
    label_prediction <- model_probs
    label_prediction[label_prediction>=0.5] <- 1
    label_prediction[label_prediction<0.5] <- 0
    error <- label_prediction!=label_test
    error <- error[error==TRUE]
    errores[i] <- length(error)/length(label_test)

}

##El error promedio es el siguiente:

mean(errores)
```

```
## [1] 0.1153846
```

A continuación sigo el mismo procedimiento pero con el knn (con k=5, que es el mejor k para nuestro modelo, anteriormente calculado con tune.knn)

```
##Aplicamos validación cruzada al modelo k-nn
set.seed(1)

probs <- rep(1,nrow(Auto))
errores <- vector()

for(i in 1:5)
{
   ##Saco la submuestra de test (20%)
    test <- sample(x=1:nrow(datos), size=(0.2*nrow(datos)), prob=probs)

    ##Pongo probs a 0 del subconjunto cogido para que no se vuelva a repetir
    probs[test] <- 0

    ##Realizo el etiquetado
    mediana <- median(Auto$mpg)
    label <- Auto$mpg
```

```
    positivos <- label>=mediana
    label[positivos] <- 1
    label[!positivos] <- -1

    label_test <- label[test]
    label_training <- label[-test]

    training_set <- datos_normalizados[-test,]
    test_set <- datos_normalizados[test,]
    ##Aplicamos la funcion k-nn
    prediction <- knn(training_set, test_set, cl=label_training, k=5)

    ##Vemos el error
    error <- prediction[prediction!=label_test]
    errores[i] <- length(error)/length(label_test)
}

mean(errores)
```

```
## [1] 0.09230769
```

Ejercicio 2

Para trabajar con la base de datos Boston, tengo que cargar el paquete 'MASS'. Guardaré la base de datos en una variable llamada dataBoston. Para realizar este ejercicio he tomado como fuente el libro ISLR que hay en decsai.

```
############EJERCICIO 2
dataBoston <- Boston
```

Apartado a

A continuación, para hacer una selección de características, emplearé un modelo de regresión Lasso. Para ello usaré el paquete 'glmnet'. Mediante validación cruzada averiguamos el mejor lambda, ya que glmnet aplica el modelo para varios lambdas, dados como argumento o por defecto. Mediante predict hacemos una 'predicción' de los coeficientes y le daremos como argumento el mejor lambda calculado por cross validation. Para ver cuales son relevantes, cogeremos aquellos cuyo valor absoluto sea mayor al umbral 10^{-2}. Usaremos crim como variable continua.

```
##Apartado a
set.seed(1)
##Creo un array que será nuestro Y (variable que predeciremos)

label_boston <- dataBoston$crim
##Elimino del dataset la columna que consideraremos etiqueta

dataBoston <- dataBoston[,-1]


##Aplico el modelo lasso sobre todo el conjunto de datos.
##Para aplicar regresión Lasso, tenemos que elegir el parámetro
```

```
##alpha de la función glmnet como '1'. Primero averiguo mediante
##Validación cuzada el mejor lambda posible

modelo_lasso <- cv.glmnet(as.matrix(dataBoston), label_boston, alpha=1)
mejor_lambda <- modelo_lasso$lambda.min


##A continuación aplico regresión lasso y hago una predicción de los
##coeficientes para el mejor lambda anteriormente calculado
##A glmnet hay que pasar el dataSet como una matriz

out <- glmnet(as.matrix(dataBoston), label_boston, alpha=1)
coefs <- predict(out, type="coefficients", s=mejor_lambda)


##Valor de los coeficientes:
coefs
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                          1
## (Intercept) 14.428038932
## zn           0.039364727
## indus       -0.071259165
## chas        -0.641610611
## nox         -8.404034511
## rm           0.321491735
## age          .
## dis         -0.875434881
## rad          0.538512550
## tax         -0.001104582
## ptratio     -0.223867381
## black       -0.007540242
## lstat        0.127002383
## medv        -0.174760563
```

```
##Cogeremos los coeficientes cuyo valor absoluto sea mayor que
##el umbral 10^-2
coeficientes_validos <- coefs[2:nrow(coefs),]
coeficientes_validos <- abs(coeficientes_validos)>=10^-2
```

Ya tenemos los coeficientes que son más grandes que el umbral propuesto. Ahora tenemos que eliminar las características que no lo superan:

```
dataBoston <- dataBoston[,coeficientes_validos]

##Nos quedamos con 10 características
```

Finalmente nos quedamos con 10 características (de las 13 que había inicialmente).

Apartado b

Ahora ajustaremos un modelo de regresión regularizada con "weight-decay" (ridge-regression). Usaremos las variables seleccionadas por el modelo Lasso. Para aplicar este modelo de regresión regularizada, emplearemos

la misma función que para el modelo Lasso (glmnet), pero como parámetro elegiremos alpha=0 (que indica que queremos realizar una regresión ridge).

Primero, al igual que al aplicar Lasso, buscaremos el mejor lambda posible mediante Cross-validation.

```
##Apartado b

set.seed(1)
##Averiguo mejor lambda

ridge.cv <- cv.glmnet(as.matrix(dataBoston), label_boston, alpha=0)
mejor_lambda <- ridge.cv$lambda.min

##Aplicamos ahora regresión ridge
##Predecimos los coeficientes de la misma manera que en el modelo lasso
##con la función predict

out <- glmnet(as.matrix(dataBoston), label_boston, alpha=0)
coefs <- predict(out, type="coefficients", s=mejor_lambda)
coefs <- coefs[1:nrow(coefs),]

##Los coeficientes (pesos) son los siguientes
coefs
```

```
## (Intercept)          zn       indus        chas         nox          rm
##  5.20805729  0.03632933 -0.05140127 -0.91377317 -4.09499387  0.50968577
##         dis         rad     ptratio       lstat        medv
## -0.71150058  0.48490279 -0.14050335  0.15576203 -0.16028160
```

```
##A continución calculamos la predicción con predict
##y posteriormente el error residual cuadrático
prediccion <- predict(out, s=mejor_lambda, newx=as.matrix(dataBoston))

error <- mean((prediccion-label_boston)^2)


error
```

```
## [1] 41.04487
```

El error es 41.044. Como vemos el error es algo grande y esto podría significar que hay 'under-fitting'. Esto quiere decir que el error dentro de la muestra es grande debido a que nuestro ajuste se ha quedado "escueto" frente a la complejidad de la función f.

Apartado c

Para este apartado definiremos un etiquetado (-1 y +1) dependiendo de que crim esté por encima o por debajo de la mediana:

```
##Apartado c

##Creamos el etiquetado
```

```
mediana <- median(label_boston)
label_crim <- label_boston
positivos <- label_crim >= mediana
label_crim[positivos] <- +1
label_crim[!positivos] <- -1
label_crim <- as.factor(label_crim)


##Para que tune svm funcione debo tener la etiqueta en
##el dataset

dataBoston <- cbind(dataBoston, label_crim)
```

A continuación aplicaremos svm (Support Vector Machine). Support vector machine busca un hiperplano que separe de forma óptima los puntos de una clase y otra, manteniéndose informalmente dicho 'en medio' de las dos clases.

Primero aplicaremos tune al modelo para ver que 'cost' de los dados como lista es el mejor. El argumento 'cost' representa el tamaño del margen del SVM. Cuando el argumento 'cost' es pequeño, entonces el margen es amplio. Si cost es mayor, el margen será menor. Los valores de cost los daré de forma arbitraria.

```
set.seed(1)
out.tune <- tune(svm ,label_crim ~ ., data = dataBoston, kernel="linear",
                 ranges = list(cost=c(0.001, 0.01, 0.1, 1, 5, 10, 100)))
```

Este es el resumen de la prueba:

```
summary(out.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   100
##
## - best performance: 0.1224314
##
## - Detailed performance results:
##    cost     error dispersion
## 1 1e-03 0.1898039 0.05281499
## 2 1e-02 0.1620392 0.04889791
## 3 1e-01 0.1560784 0.05003105
## 4 1e+00 0.1361961 0.05254481
## 5 5e+00 0.1342353 0.04862255
## 6 1e+01 0.1283529 0.04548834
## 7 1e+02 0.1224314 0.04867304
```

En el anterior resumen hemos visto los distintos errores y varianzas para cada uno de los costes probados. Cogeremos pues el modelo que menor error nos haya dado directamente desde el objeto devuelto por la función tune.

```
##Con $best.parameters saco el mejor valor del coste
mejor_modelo <- out.tune$best.parameters

boston.svm <- svm(label_crim~., data=dataBoston, kernel="linear", cost=mejor_modelo)
prediccion <- predict(boston.svm, dataBoston)



##Tabla de confusión
table(predict=prediccion, truth=label_crim)
```

```
##        truth
## predict  -1   1
##      -1 227  26
##      1   26 227
```

El error dentro de la muestra es el siguiente

```
error <- prediccion[prediccion!=dataBoston$label_crim]
error <- length(error)/nrow(dataBoston)
error
```

```
## [1] 0.1027668
```

Bonus-3

En este apartado averiguaremos por validación cruzada el error de Test y Training del Support Vector Machine.

Al igual que en el bonus del ejercicio 1, crearemos las particiones usando sample y un vector de probabilidades que será puesto a 0 en las posiciones que vayamos cogiendo para que no se repitan en la siguiente iteración. También seguiremos los mismos pasos que en el apartado c, cogiendo primero por validación cruzada el mejor coste y aplicándolo en nuestra validación cruzada.

```
##Bonus-3

##Validación cruzada para svm
set.seed(1)

probs <- rep(1,nrow(dataBoston))
errores_training <- vector()
errores_test <- vector()

##Encuentro el coste óptimo para nuestro caso
out.tune <- out.tune <- tune(svm ,label_crim ~ ., data = dataBoston, kernel="linear",
ranges = list(cost=c(0.001, 0.01, 0.1, 1, 5, 10, 100)))

mejor_modelo <- out.tune$best.parameters

for(i in 1:5)
{
  test <- sample(x=1:nrow(dataBoston), prob = probs, size = 0.2*nrow(dataBoston))
```

```
  ##Pongo las probabilidades de los elementos cogidos a test a 0 para
  ##que en la siguiente iteración no sean cogidos de nuevo
  probs[test] <- 0

  training_set <- dataBoston[-test,]
  test_set <- dataBoston[test,]

  ##Aplico svm
  boston.svm <- svm(label_crim ~., data=training_set, kernel="linear", cost=mejor_modelo)
  prediccion_training <- predict(boston.svm, training_set)
  prediccion_test <- predict(boston.svm, test_set)

  ##Calculo los errores de training y test
  error_training <- prediccion_training[prediccion_training!=training_set$label_crim]
  errores_training[i] <- length(error_training)/nrow(training_set)
  error_test <- prediccion_test[prediccion_test!=test_set$label_crim]
  errores_test[i] <- length(error_test)/nrow(test_set)
}

error_test <- mean(errores_test)
error_training <- mean(errores_training)
```

El error medio de test por validación cruzada es:

```
#Error de test
error_test
```

```
## [1] 0.1128713
```

El error medio de training por validación cruzada es:

```
#Error de training
error_training
```

```
## [1] 0.09876543
```

Ejercicio 3

Apartado a

Procedemos a la separación en dos subconjuntos: test y training (80% y 20% respectivamente).

```
##############EJERCICIO 3

##Apartado a
set.seed(1)
dataBoston3 <- Boston

##Cojo el 80% de los índices de forma aleatoria
```

```
##Preparo también nuestra variable que queremos predecir

training <- sample(x=1:nrow(dataBoston3), size=0.8*nrow(dataBoston3))
```

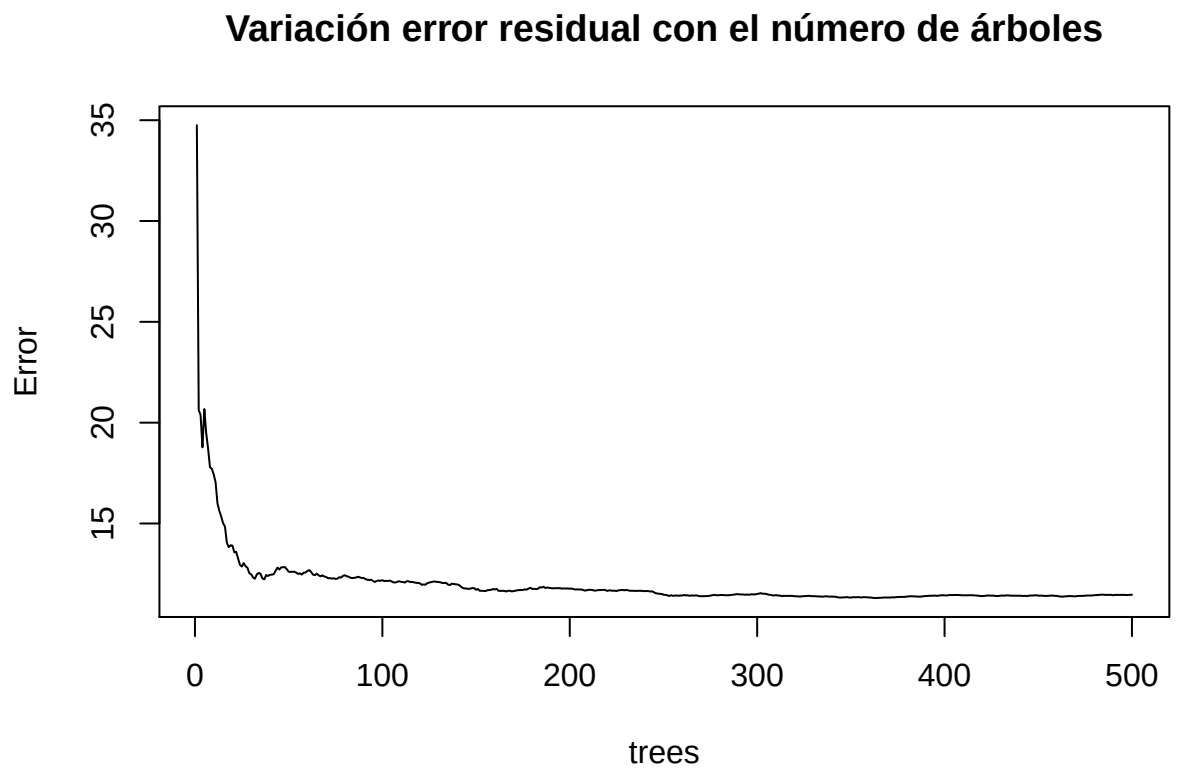
Apartado b

Para aplicar 'bagging', tenemos que tener en cuenta que 'bagging' se puede considerar equivalente a realizar un modelo de Random Forest en el que el número de variables para la ramificación es igual que el número total de variables predictoras. Por lo tanto usaremos la función randomForest de la librería randomForest.

Los parámetros que metemos a la función randomForest se tratan de formula, donde pondremos la variable que queremos predecir, data, que será el dataset que usaremos, subset, que será el conjunto de índices que corresponderá a los índices de la partición de datos y por último mtry que será el número de variables para la ramificación. También daremos el parámetro importance=TRUE para que se evalúe la importancia de las variables predictoras.

```
##Apartado b

set.seed(1)
boston.model <- randomForest(formula= dataBoston3$medv ~ ., data=dataBoston3,
                             subset=training, mtry=ncol(dataBoston3)-1, importance=TRUE)
```

A continuación una gráfica que muestra la variación del error cuadrático respecto el número de árboles usados en el bagging. Hemos puesto como número de árboles 500 (el de por defecto).



Ahora procederemos a calcular el error de test. Para ello usaremos como en ejercicios anteriores la función predict.

```
##Realizamos la predicción con 'predict'
##introducimos como parámetros el modelo que hemos predicho
##con bagging y el subconjunto de test


boston.pred <- predict(boston.model, newdata = dataBoston3[-training,])

y.medv <- dataBoston3$medv[-training]
sq.error <- mean((boston.pred-y.medv)^2)

##El error cuadrático medio es el siguiente
sq.error
```

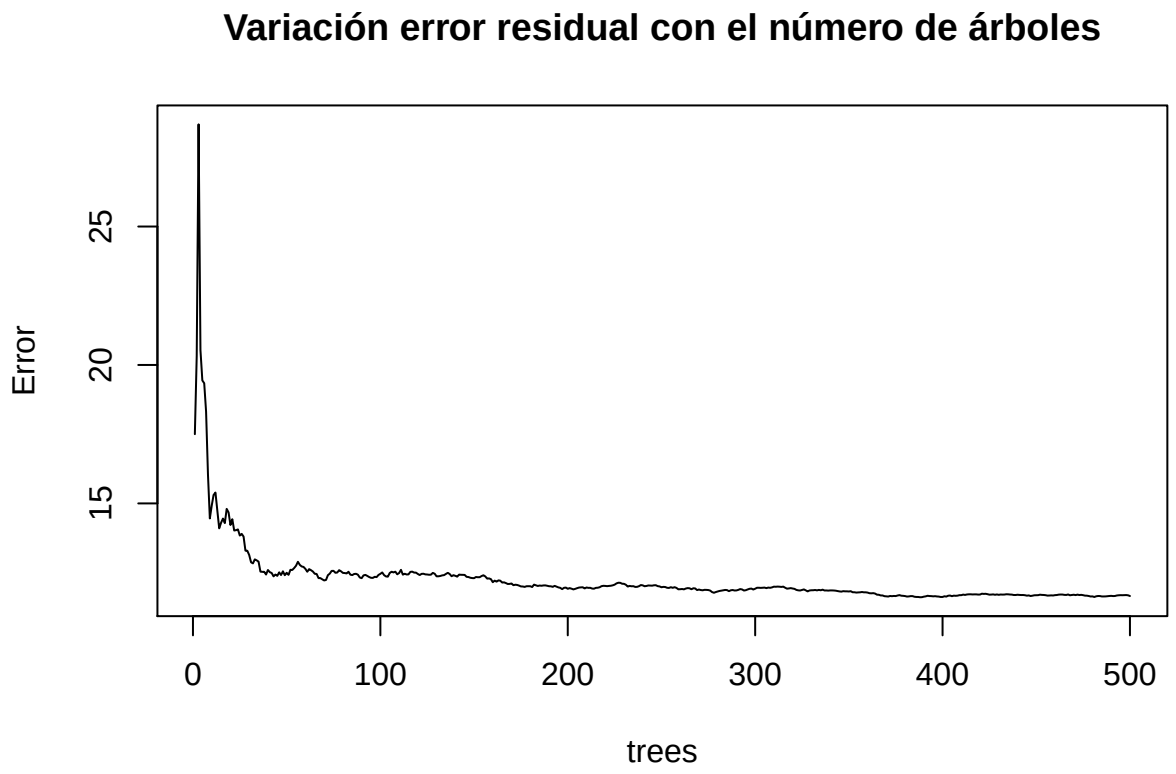
```
## [1] 7.77245
```

Apartado c

Ahora para estimar mediante un modelo RandomForest usaemos de nuevo la misma función que en el apartado anterior. A diferencia del apartado anterior, ahora usaremos como número de variables para ramificación $\frac{p}{3}$ siendo p el número de variables predictoras.

Usamos las mismas particiones para hacer la comparación con bagging en igualdad de condiciones.

```
##Apartado c

set.seed(1)

boston.rf.model <- randomForest(formula= dataBoston3$medv ~ ., data=dataBoston3,
                                subset=training, mtry=(ncol(dataBoston)-1)/3, importance=TRUE)

##Gráfico de la variación del error con el número de árboles
plot(boston.rf.model, main="Variación error residual con el número de árboles")
```



He intentado aplicar la funcón tune.randomForest pero no he conseguido hacerla funcionar.

Ahora calculamos el error de test igual que en el apartado anterior

```
boston.pred.rf <- predict(boston.rf.model, newdata = dataBoston3[-training,])

y.medv.rf <- dataBoston3$medv[-training]
sq.error.rf <- mean((boston.pred.rf-y.medv.rf)^2)

##El error de test es el siguiente
sq.error.rf
```

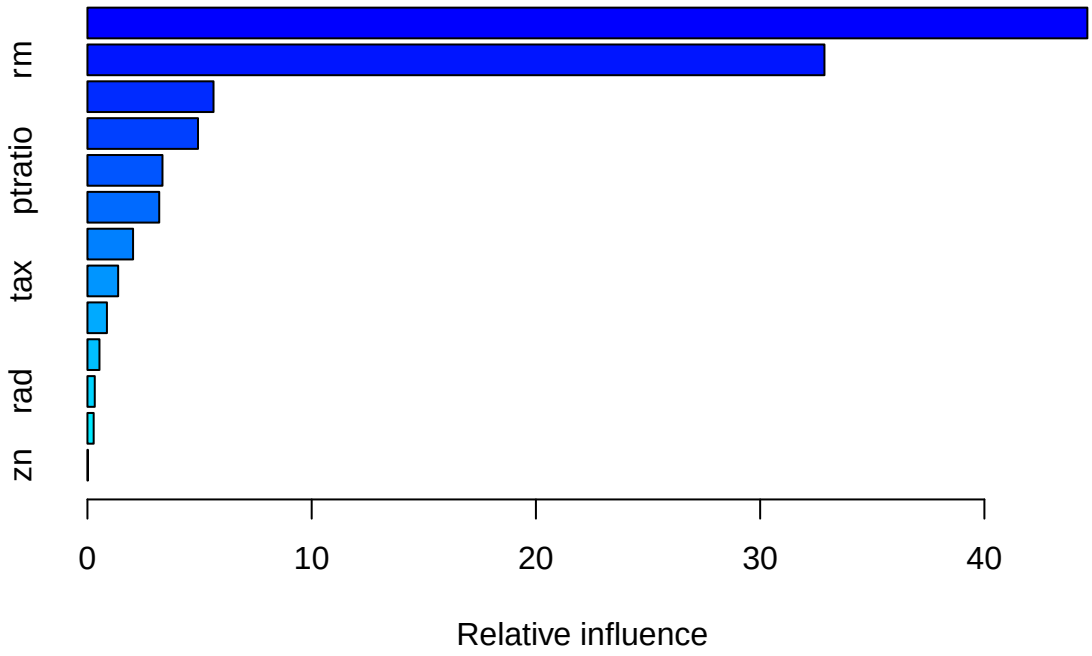
```
## [1] 8.481
```

El error en este caso no varía significativamente con respecto al de 'bagging'. En 'bagging' el error está sólo un 1% por debajo que el de 'Random Forest'.

Apartado d

Para aplicar 'Boosting' usaremos la función gbm() del paquete gbm. Esta función ajuta boosting usando árboles de regresión. Usaremos la misma partición de los datos que en los apartados anteriores para mantenre la igualdad de condiciones. El argumento distribution ="gaussian" es para indicarle que es un problema de regresión. Con n.trees=5000 le indico que quiero 5000 árboles y cno interaction.depth le digo que el límite de profundidad por árbol sea 4.

```r
##Apartado d
set.seed(1)

boston.boost <- gbm(formula=dataBoston3$medv[training] ~., data=dataBoston3[training,],
                    distribution ="gaussian",n.trees =5000 , interaction.depth =4)

summary(boston.boost)
```




```
##             var     rel.inf
## lstat     lstat 44.59185317
## rm           rm 32.86765795
## dis         dis  5.62251509
## nox         nox  4.93079948
## ptratio ptratio  3.34634028
## crim       crim  3.20150126
## age         age  2.03491364
## tax         tax  1.37029189
## black     black  0.86988798
## indus     indus  0.53654575
## rad         rad  0.32667686
## chas       chas  0.27783381
## zn           zn  0.02318283
```

Ahora procedemos a calcular el error de test del modelo. Seguimos los mismos pasos que en apartados anteriores.

```
##Hacemos la predicción

boston.pred.boost <- predict(boston.boost, newdata = dataBoston3[-training,], n.trees=5000)
y.medv.boost <- dataBoston3$medv[-training]
sq.error.boost <- mean((boston.pred.boost-y.medv.boost)^2)

##El error de test es el siguiente
sq.error.boost
```

```
## [1] 8.317748
```

Como vemos, igual que los 2 anteriores modelos, 'boosting' presenta para la misma muestra un error de test similar. Presenta un error prácticamente igual que en randomForest y sólo un 1% por encima del de 'bagging'.

Ejercicio 4

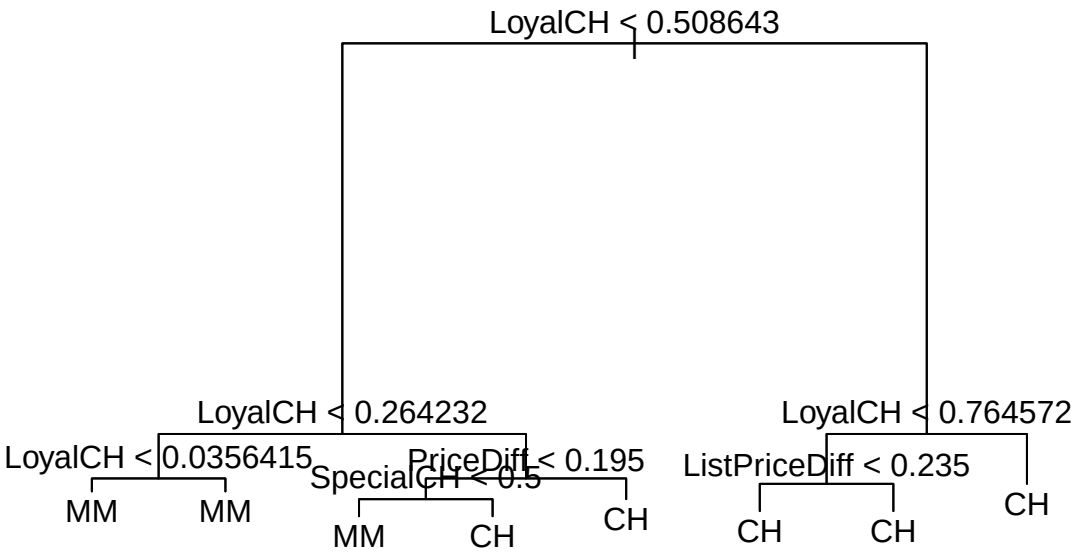
Para este ejercicio usaremos la base de datos OJ de la biblioteca ISLR. Esta base de datos contiene datos sobre zumos de naranja. Lo que tenemos que hacer primero es crear un conjunto de entrenamiento y posteriormente elegir la variable 'Purchase' como variable que queremos predecir. Posteriormente ajustaremos un árbol.

Apartado a, b y c

```
##########EJERCICIO 4
#Apartados a, b y c


datos4 <- OJ

##Creo un array que serán las variables respuesta

set.seed(1)

##creamos la submuestra de 800 observaciones para training
training_4 <- sample(x=1:nrow(datos4), size=800)

training.OJ <- datos4[training_4,]
test.OJ <- datos4[-training_4,]
```

A continuación ajustaremos un árbol a los datos de entrenamiento y lo dibujamos

```
#Ajustamos un árbol y lo dibujamos para interpretarlo (interpretación en el pdf)
set.seed(1)

tree.OJ <- tree(training.OJ$Purchase~., training.OJ)

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



El árbol que hemos ajustado usa sólo 4 variables del conjunto de variables predictoras para ajustarse. Lo que el árbol hace es, si LoyalCH es mayor que 0.508643, entonces Purchase será CH (con distintas probabilidades eso sí, dependiendo también posteriormente de ListPriceDiff). Si es menor de lo que hemos dicho entonces si es menor que 0.264232 Purchase será MM con distintas probabilidades. Si es mayor, entonces si PriceDiff es mayor que 0.195 purchase será CH y si es menor, entonces si SpecialCH es menor de 0.5 será MM y si no será CH.

Ahora analizamos un resumen del ajuste

```
##Resumen del árbol

summary(tree.OJ)
```

```
##
## Classification tree:
## tree(formula = training.OJ$Purchase ~ ., data = training.OJ)
## 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
```

Como vemos, el árbol usa 4 variables predictoras para ajustar el árbol. Tiene 8 nodos terminales y 15 nodos en total (contados a partir del dibujo). El error de training es del 16.5%.

Apartado d

Ahora calcularemos el error de test. Para hacerlo usaremos como en otros ejercicios anteriores la función predict. Le indico mediante el parámetro type que se trata de un problema de clasificación.

```
#Apartado d

purchase.pred <- predict(tree.OJ, newdata = test.OJ, type="class")

##Tabla de confusión
tabla <- table(purchase.pred, datos4$Purchase[-training_4])

tabla
```

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

```
##Calculo la tasa de error de test

errores <- purchase.pred[purchase.pred!=datos4$Purchase[-training_4]]
error <- length(errores)/length(purchase.pred)
error
```

```
## [1] 0.2259259
```

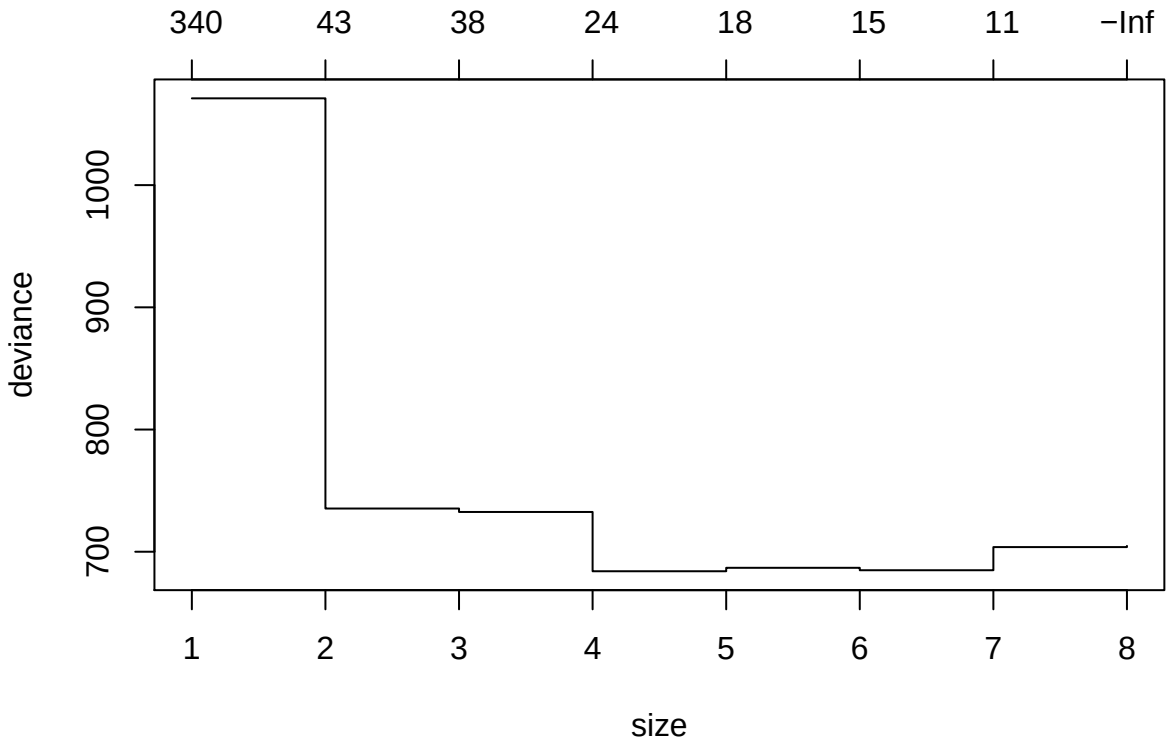
Como vemos la tasa de error de test es de un 22.59%. La precisión la calculamos como: $\frac{VerdaderosPositivos}{FalsosPositivos+VerdaderosPositivos}$ en el caso de la precisión positiva y como $\frac{VerdaderosNegativos}{FalsosNegativos+VerdaderosNegativos}$. Consideraremos respuestas positivas CH, por lo que la precisión es: $147/(147+49) = 0.75$. Hay una precisión positiva del 75%. La precisión negativa será $62/(62+12) = 0.8378$. La precisión negativa es del 83.78%.

En cuanto a la tabla de confusión, se puede observar que hay muchos más falsos negativos que falsos positivos. Esto puede deberse a que en proporción en el conjunto de training haya más datos 'CH' que 'MM' y el árbol se haya ajustado de manera que tienda a coger la clase mayoritaria.

Apartado e

Ahora mediante la función cv.tree, averiguaremos por validación cruzada el tamaño óptimo del árbol. A la función le introducimos como parámetro el árbol anteriormente ajustado con el training set.

```
#Apartado e

#Por validación cruzada comprobamos el mejor árbol
tree.cv <- cv.tree(tree.OJ)

plot(tree.cv)
```



Al ver la gráfica, yo cogería el árbol de tamaño que menos desviación tenga (menos varianza). En este caso según la gráfica es el de tamaño 7, ya que cuando llega a 8 vuelve a subir la varianza.