Aprendizaje Automático

Práctica 3

Braulio Vargas López

12 de mayo de 2016

Índice

1. 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 mediana 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.

b) Seleccionar las variables predictoras que considere más relevantes.

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

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.frame()` para unir en un mismo conjunto de datos la nueva variable $mpg01$ y las otras variables de Auto).

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

e) 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) Ajustar el mejor modelo de regresión posible considerando la variable mpg como salida y el resto como predictoras. Justificar el modelo ajustado en base al patrón de los residuos. Estimar su error de entrenamiento y test.

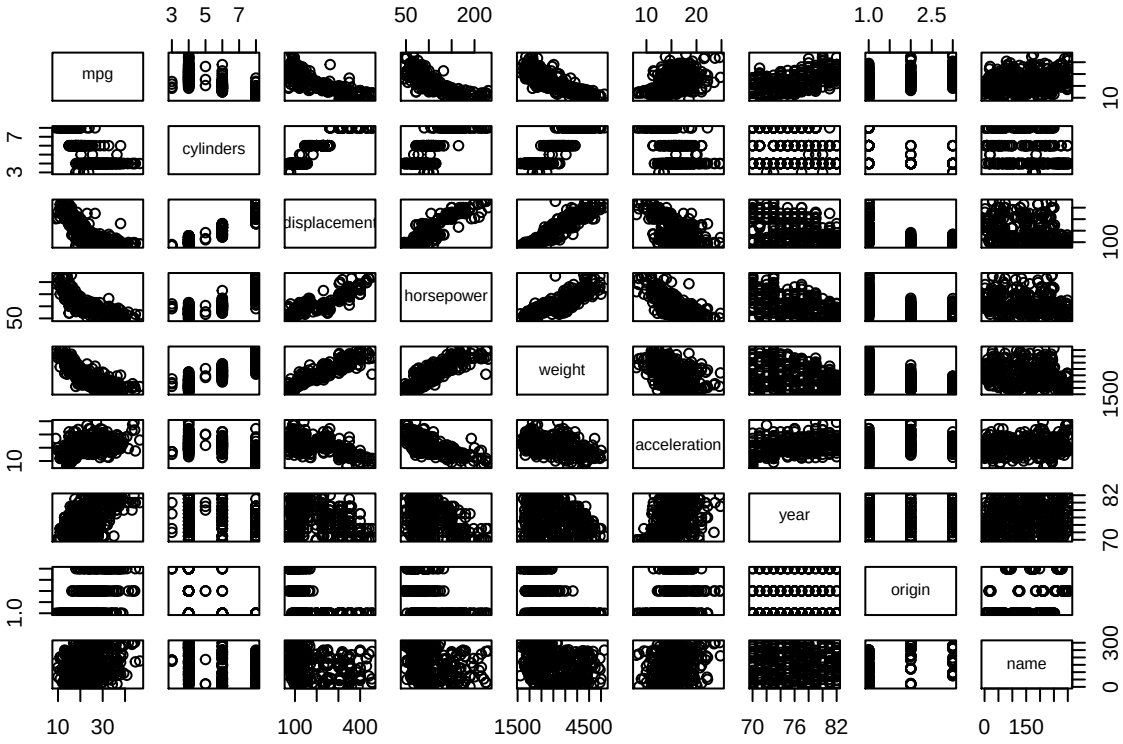
1.1. Solución

a) En este apartado, podemos mostrar las gráficas resultantes de cruzar todos con todos, y ver en cuales de ellas parece haber alguna relación. Para ello, usamos lo siguiente:

```
attach(Auto)
```

```
## The following object is masked from package:ggplot2:
##
##     mpg
```

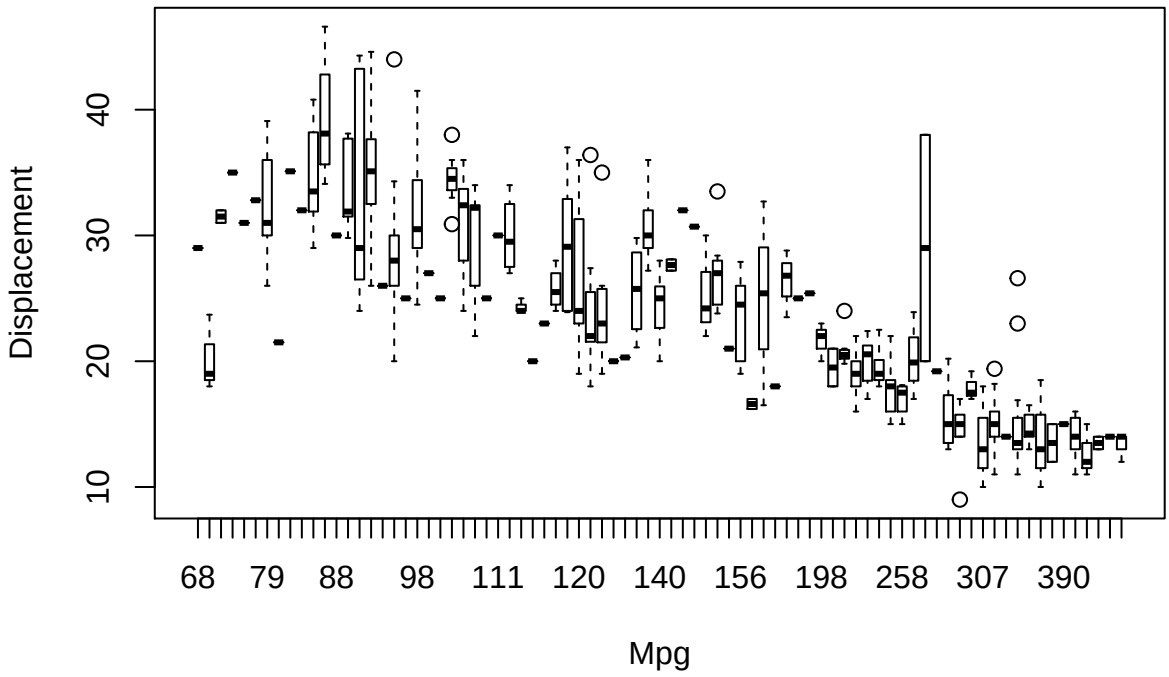
```
pairs(~ ., data=Auto)
```



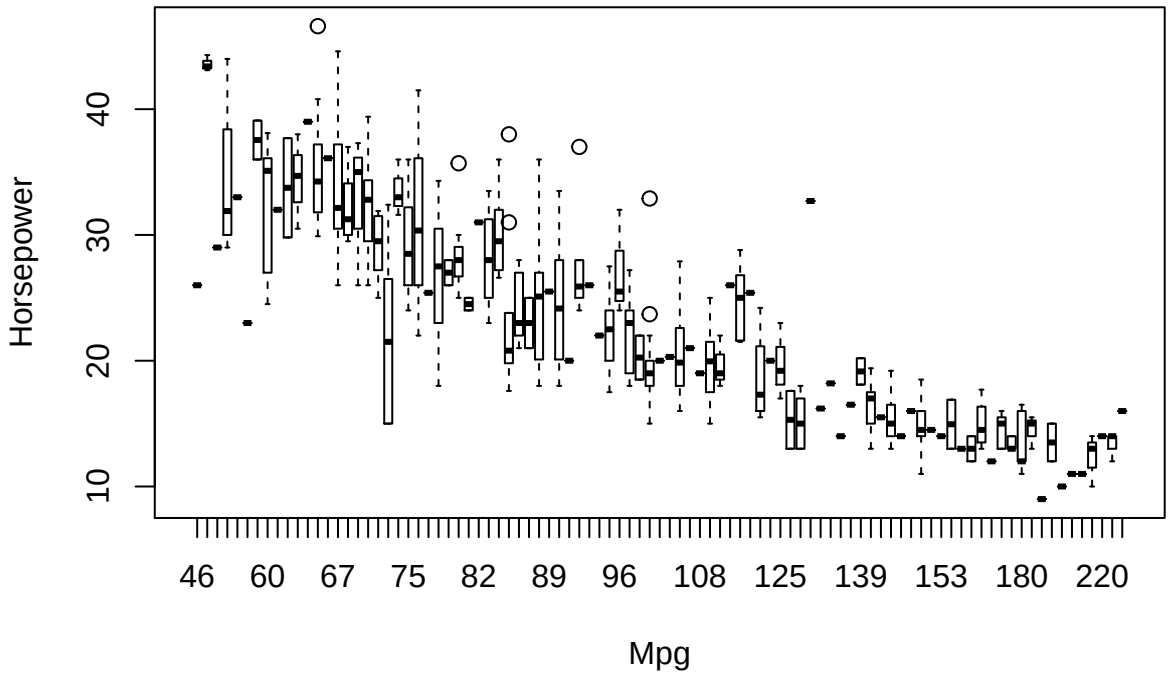
En estas gráficas, vemos que existe una relación entre *mpg* y *displacement*, *mpg* y *horsepower* y entre *mpg* y *weight*, ya que se ve una relación clara entre estas variables, mientras que en otras, podemos ver que los datos varían demasiado o tienen mucho ruido como para poder aprender algo.

Si usamos `boxplot` para representar las variables obtenemos las siguientes gráficas:

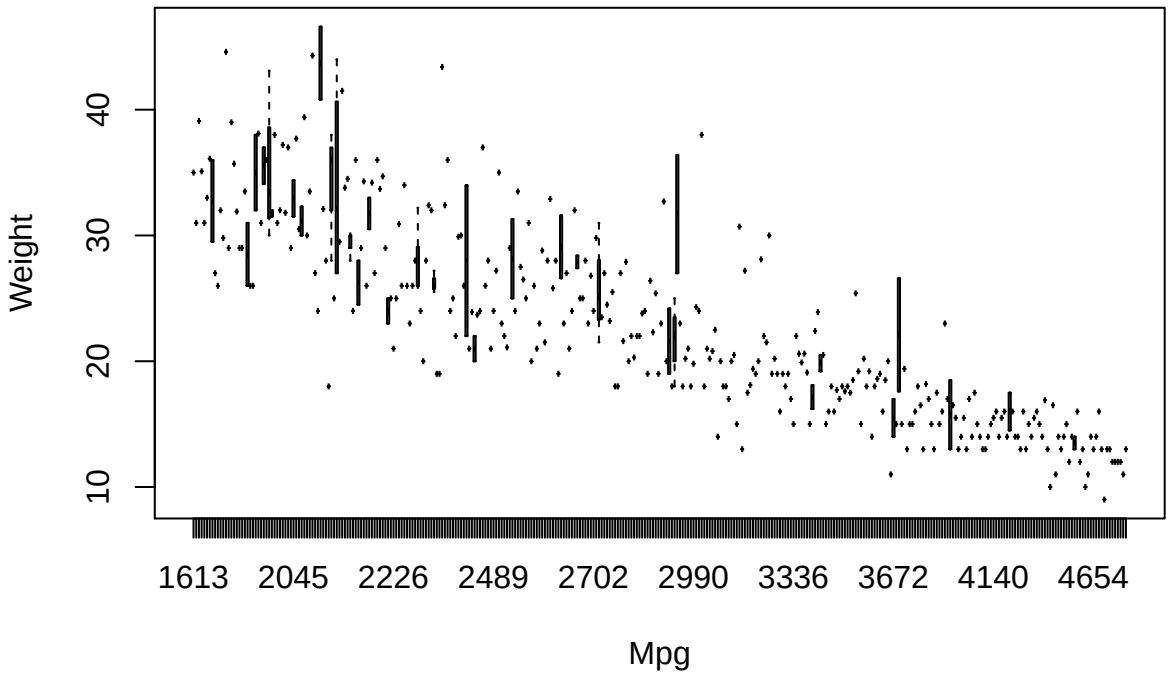
```
boxplot(mpg ~ displacement, data=Auto, xlab="Mpg", ylab = "Displacement")
```



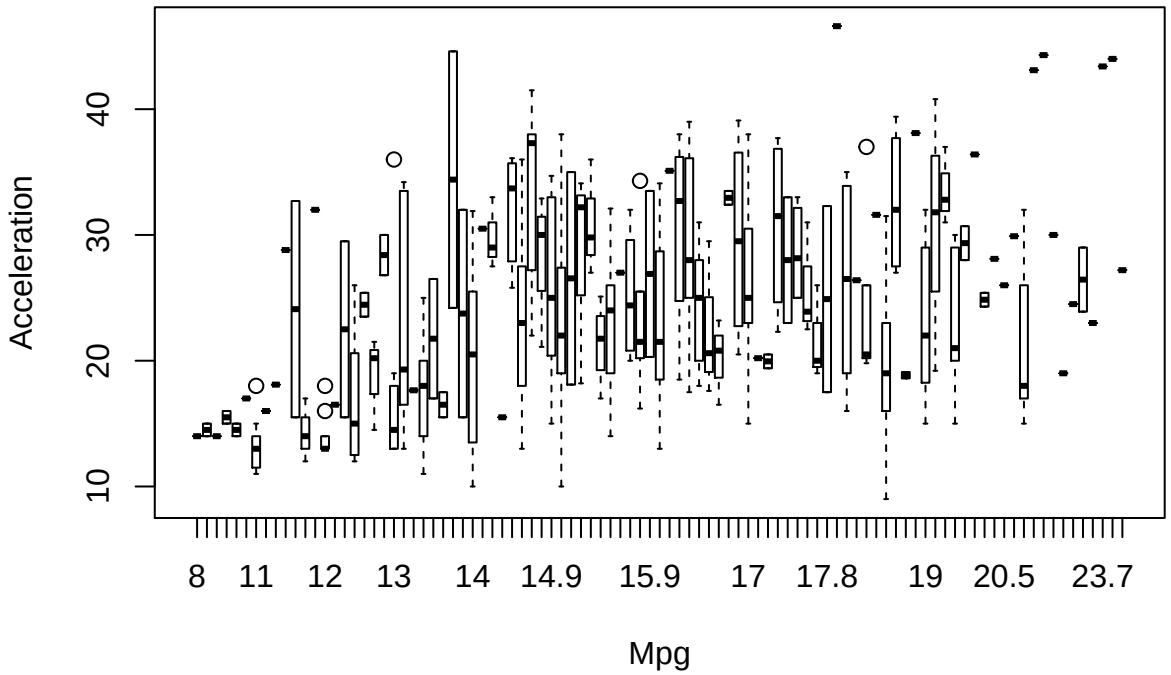

```
boxplot(mpg ~ horsepower, data=Auto, xlab="Mpg", ylab = "Horsepower")
```




```
boxplot(mpg ~ weight, data=Auto, xlab="Mpg", ylab = "Weight")
```




```
boxplot(mpg ~ acceleration, data=Auto, xlab="Mpg", ylab = "Acceleration")
```



En las distintas gráficas, podemos ver la relación que hay entre los dos elementos que estamos comparando. En esta relación podemos ver representados cómo la "caja" representa el intervalo entre el cuartil 1 y el 3, y el rango de valores total. Además, los *outliers* quedan representados como un solo punto. En las distintas gráficas, podemos ver como todas tienen una varianza pequeña, o el ruido en la función es menor, excepto si vemos el resultado de `boxplot`entre *mpg* y *acceleration*, que vemos como se ve incrementado la varianza y el ruido en los datos.

b) Una vez visto lo anterior, las variables más relevantes para poder predecir *mpg* vemos que son *displacement*, *horsepower* y *weight*. Pero, para escoger mejor los datos, podemos hacer un ajuste lineal rápido con la función `lm` y ver qué variable de estas tres es más importante con la función `summary`.

c) Para dividir el conjunto de datos en entrenamiento y test, podemos hacer lo siguiente:

```
i = sample(x=nrow(Auto), size=floor(nrow(Auto))*0.8)
data.train = Auto[i,]
data.test = Auto[-i,]
```

Para dividir los datos, cogemos una muestra aleatoria de índices de longitud igual $0,8 \cdot n_{muestras}$, entre 1 y el número total de muestras que tenemos. Esta muestra de índices, la usaremos para aprender el modelo con el 80 % de los datos, y usar el 20 % para test.

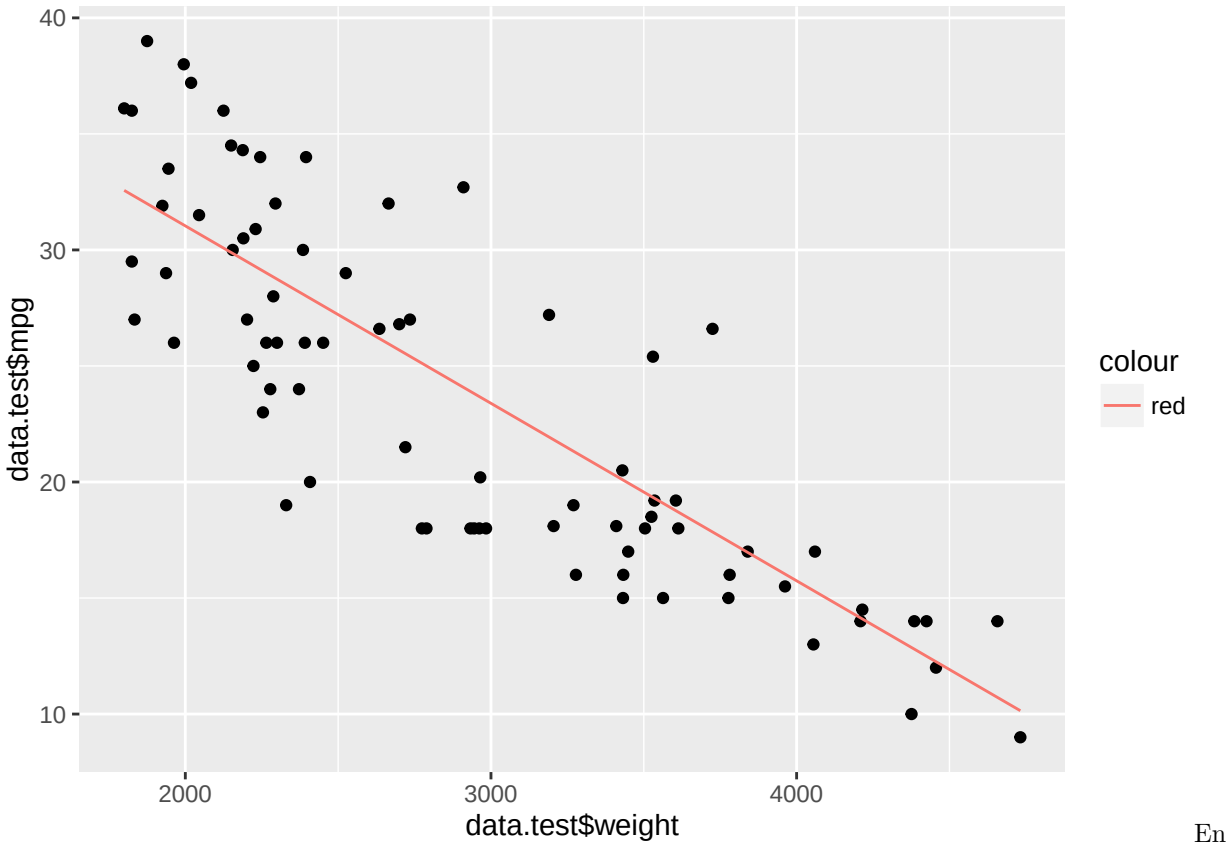
Una vez divididos, vamosa realizar una serie de modelos lineales, para estudiar cuál de ellos es mejor:

1) En este caso, realizamos un modelo lineal simple que tiene en cuenta solo *mpg* en función de *weight*

```
i = sample(x=nrow(Auto), size=floor(nrow(Auto))*0.8)
m01= lm(mpg~weight,data = Auto, subset= i)
summary(m01)
```

```
##
## Call:
## lm(formula = mpg ~ weight, data = Auto, subset = i)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -12.088  -2.867  -0.362   2.228  16.404
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 46.3308178  0.9138897   50.70   <2e-16 ***
## weight      -0.0076470  0.0002958  -25.85   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.402 on 311 degrees of freedom
## Multiple R-squared:  0.6825, Adjusted R-squared:  0.6814
## F-statistic: 668.4 on 1 and 311 DF,  p-value: < 2.2e-16
```

```
(ggplot() + geom_point(data=as.data.frame(data.test),
           aes(x=data.test$weight, y=data.test$mpg))
         + geom_line(aes(x=data.test$weight,
                         y=predict(m01,data.test), color="red")))
```



En el summary, podemos ver cómo el error dentro de la muestra, es del 0.6 %, que para ser un modelo lineal simple, no está nada mal. Pero, al estimar los valores con el conjunto de test, vemos que tampoco lo hace mal, pero se puede mejorar y para ello, tendremos los siguientes modelos.

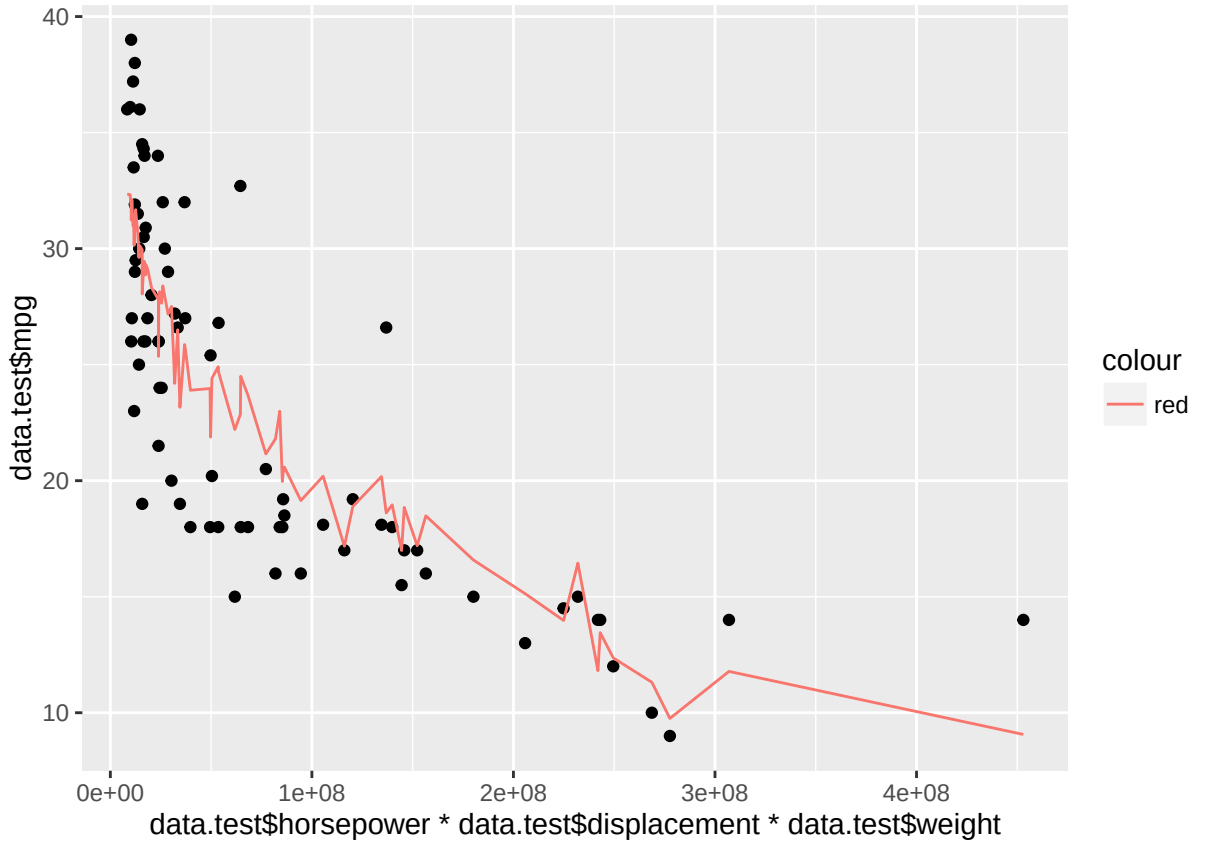
2) En este caso, ya no solo tenemos en cuenta *weight*, si no que ya tenemos en cuenta también *displacement* y *horsepower*. En este caso, obtenemos que el error es de casi el 0.7 %, un poco mayor que en el modelo anterior, pero, fuera de la muestra, lo hace bastante mejor que $m01$.

```
m02 = lm(mpg ~ weight+displacement+horsepower, data=Auto, subset = i)
summary(m02)
```

```
##
## Call:
## lm(formula = mpg ~ weight + displacement + horsepower, data = Auto,
##     subset = i)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -11.4380  -2.7819  -0.4032   2.3437  16.1016
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)  44.9963148  1.3568941  33.161  < 2e-16 ***
## weight       -0.0052854  0.0007916  -6.677 1.13e-10 ***
## displacement -0.0061215  0.0075894  -0.807  0.42052
## horsepower   -0.0433734  0.0149524  -2.901  0.00399 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 4.305 on 309 degrees of freedom
## Multiple R-squared:  0.6982, Adjusted R-squared:  0.6953
## F-statistic: 238.3 on 3 and 309 DF,  p-value: < 2.2e-16
```

```
(ggplot() + geom_point(data=as.data.frame(data.test),
            aes(x=data.test$horsepower*data.test$displacement*data.test$weight,
                y=data.test$mpg))
          + geom_line(aes(x=data.test$horsepower*data.test$displacement*data.test$weight,
                          y=predict(m02,data.test), color="red")))
```



3)

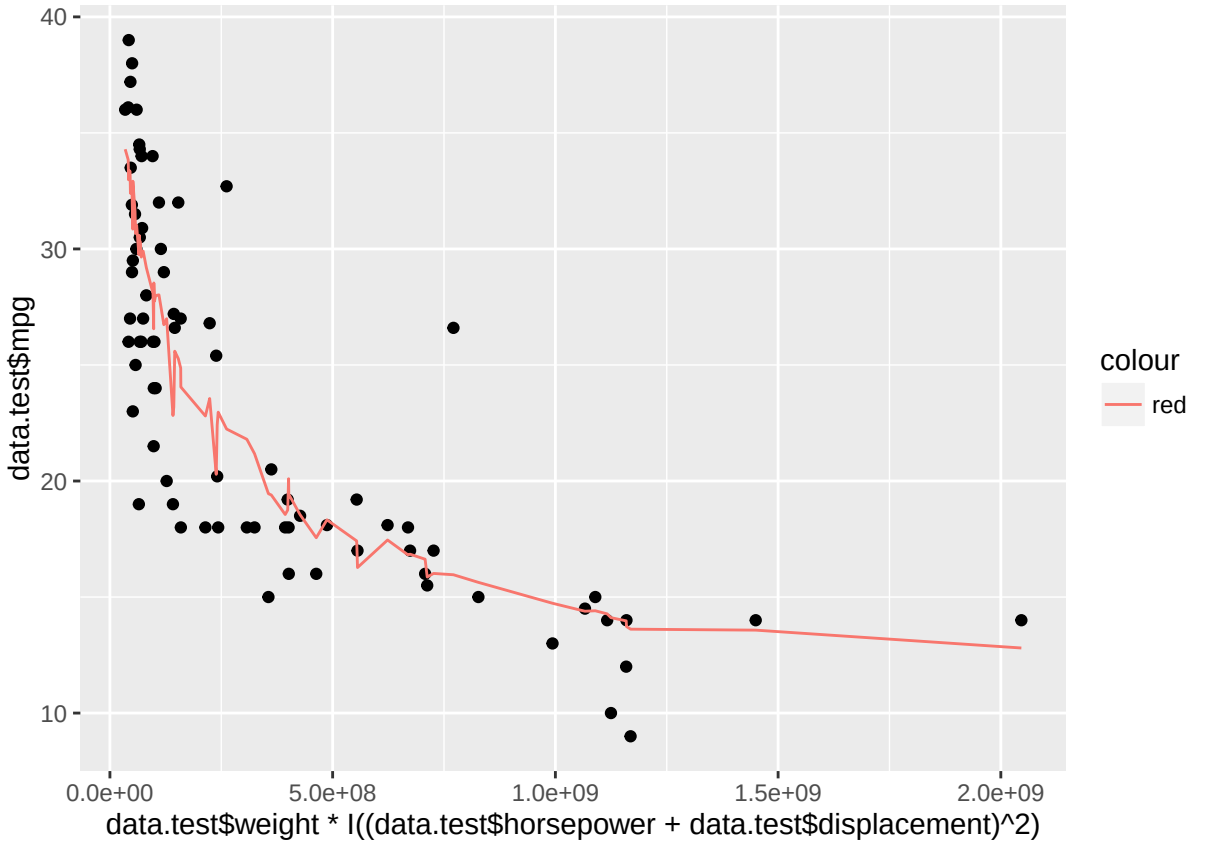
En este modelo, comenzamos a usar modelos con funciones cuadráticas, para calcular el valor del punto, usando *weight* y el cuadrado de *weight* y *horsepower*. En este modelo, `summary` nos indica que las características que usamos para calcular el valor del punto están muy relacionadas y son importantes para el cálculo. Además, aunque el error que obtenemosd dentro de la muestra es mayor que el de los anteriores, 0.7, vemos que el comportamiento fuera de la muestra es mucho mejor que el de los anteriores.

```
m03 = lm(mpg~weight*I(horsepower+displacement)^2,data = Auto, subset= i)
summary(m03)
```

```
##
## Call:
## lm(formula = mpg ~ weight * I(horsepower + displacement)^2, data = Auto,
##     subset = i)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -13.348  -2.438  -0.404   1.853  17.512
##
## Coefficients:
```

```
##                                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          5.633e+01  2.336e+00  24.118  < 2e-16
## weight                              -8.740e-03  9.545e-04  -9.157  < 2e-16
## I(horsepower + displacement)        -7.060e-02  9.833e-03  -7.180 5.24e-12
## weight:I(horsepower + displacement)  1.434e-05  2.336e-06   6.139 2.55e-09
##
## (Intercept)                         ***
## weight                              ***
## I(horsepower + displacement)        ***
## weight:I(horsepower + displacement) ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.087 on 309 degrees of freedom
## Multiple R-squared:  0.728,  Adjusted R-squared:  0.7254
## F-statistic: 275.7 on 3 and 309 DF,  p-value: < 2.2e-16
```

```
(ggplot() + geom_point(data=as.data.frame(data.test),
            aes(x=data.test$weight*I((data.test$horsepower+data.test$displacement)^2),
                y=data.test$mpg))
          + geom_line(aes(x=data.test$weight*I((data.test$horsepower+
                data.test$displacement)^2),
                y=predict(m03,data.test), color="red")))
```



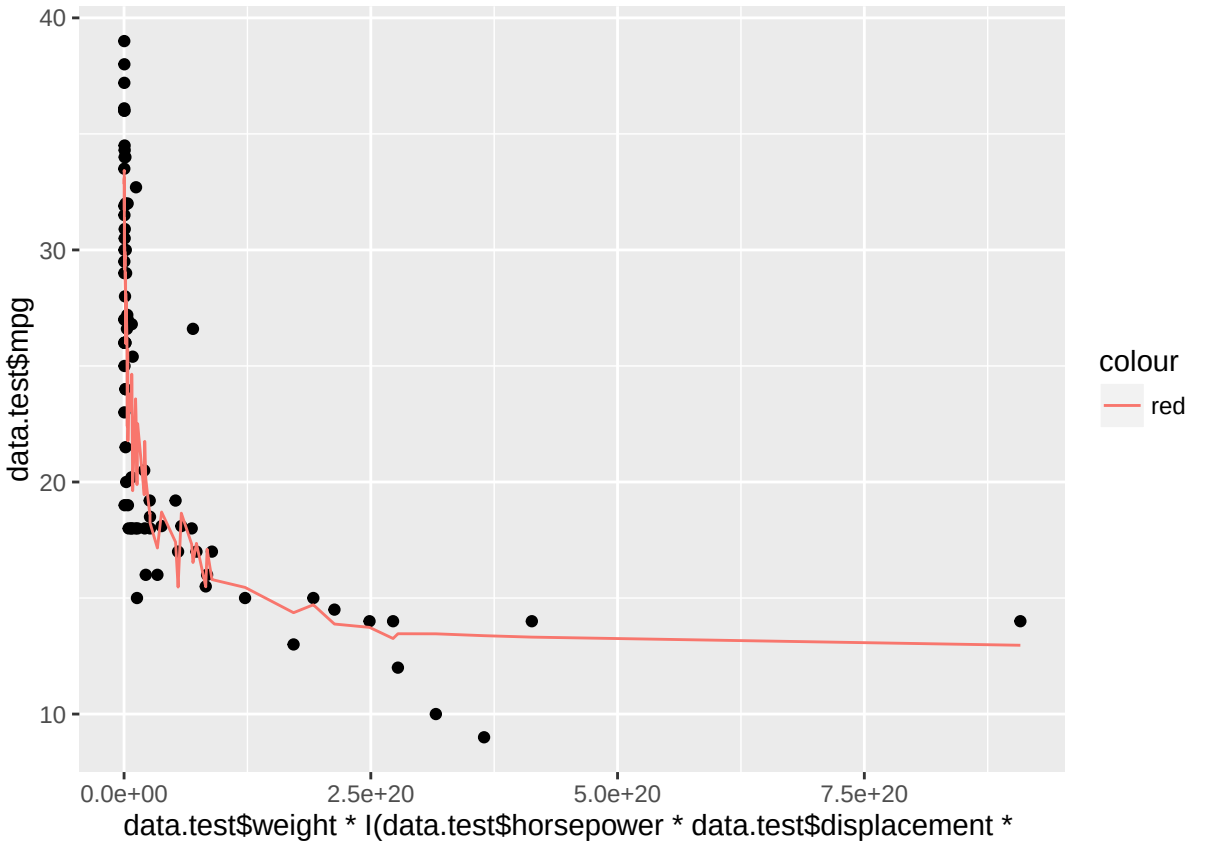
4)

En este último modelo, realizamos un modelo también no lineal de orden 2, relacionando las tres características entre sí y a su vez, con *weight*. Como primera mejora obtenemos que el error dentro de la muestra respecto al modelo anterior se ha reducido. La segunda mejora es que fuera de la muestra, lo hace incluso mejor, como se ve en la gráfica de la función y el ajuste del modelo.

```
m04 = lm(mpg~weight*I(horsepower*displacement*weight)^2,data = Auto, subset= i)
summary(m04)
```

```
##
## Call:
## lm(formula = mpg ~ weight * I(horsepower * displacement * weight)^2,
##     data = Auto, subset = i)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -12.8119  -2.7217  -0.4041   2.0139  15.8454
##
## Coefficients:
##                                              Estimate Std. Error t value
## (Intercept)                                 4.787e+01  1.454e+00  32.928
## weight                                     -7.625e-03  6.107e-04 -12.487
## I(horsepower * displacement * weight)      -1.220e-07  2.141e-08  -5.699
## weight:I(horsepower * displacement * weight)  2.700e-11  4.492e-12   6.010
##                                             Pr(>|t|)
## (Intercept)                                  < 2e-16 ***
## weight                                       < 2e-16 ***
## I(horsepower * displacement * weight)       2.81e-08 ***
## weight:I(horsepower * displacement * weight) 5.22e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.177 on 309 degrees of freedom
## Multiple R-squared:  0.7159, Adjusted R-squared:  0.7132
## F-statistic: 259.6 on 3 and 309 DF,  p-value: < 2.2e-16
```

```
(ggplot() + geom_point(data=as.data.frame(data.test),
            aes(x=data.test$weight*I(data.test$horsepower*
              data.test$displacement*data.test$weight)^2,
              y=data.test$mpg))
          + geom_line(aes(x=data.test$weight*I(data.test$horsepower*
              data.test$displacement*data.test$weight)^2,
              y=predict(m04,data.test), color="red")))
```



d)

Para crear la variable *mpg*01, podemos hacer uso de la función *sapply*, para asignar un uno o un cero en función de si el valor del dato mpg_i está por encima o por debajo de la mediana de *mpg*.

```
# Calculamos la media de mpg
medianMPG = median(mpg)
# Y etiquetamos los datos con valores 0/1 en función de si son
# mayores o menores que la mediana de mpg
mpg01 = (mpg >= medianMPG)*1
# Y creamos un dataframe con estos datos, eliminando
# los nombres del dataframe. También generamos un
# dataframe con los datos de test
datos = data.frame(Auto,as.factor(mpg01))
colnames(datos)[ncol(datos)]="mpg01"
datos$name = NULL
datos.test = data.frame(datos[-i,])
```

- Para ajustar la regresión logística, hacemos uso de *glm*, con *mpg* en función de las variables *weight*, *displacement* y *horsepower*. Crearemos varios modelos en función con regresión logística, teniendo en cuenta varios parámetros.

En este primer modelo, realizamos la regresión logística de *mpg* en función de *weight*. Una vez aprendido el modelo, calculamos las etiquetas con el modelo aprendido y generamos la matriz de confusión. Tras esto, calculamos el error dentro de la muestra (E_{in}) y fuera de la muestra (E_{out}) con los datos de test.

```
# Calculamos la regresión logística
attach(datos)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     mpg01
```

```
## The following objects are masked from Auto:
##
##     acceleration, cylinders, displacement, horsepower, mpg,
##     origin, weight, year
```

```
## The following object is masked from package:ggplot2:
##
##     mpg
```

```
glmModelErrorTest <- function(glm_model, trainingIndex, data, testData, getPredicts = F) {
  # Hacemos la predicción con los datos de test
  glm.prediction.model.train = predict(glm_model, type="response")
  glm.prediction.model.test = predict(object=glm_model, data.frame(testData), type="response")
  # Y calculamos las etiquetas con predict
  glm.prediction.model.train.y = glm.prediction.model.train
  glm.prediction.model.train.y[glm.prediction.model.train.y <= 0.5] = 0
  glm.prediction.model.train.y[glm.prediction.model.train.y > 0.5] = 1
  glm.prediction.model.test.y = glm.prediction.model.test
  glm.prediction.model.test.y[glm.prediction.model.test.y <= 0.5] = 0
  glm.prediction.model.test.y[glm.prediction.model.test.y > 0.5] = 1
  # Generamos la matriz de confusión con los datos de test
  print(table(glm.prediction.model.train.y, data$mpg01[trainingIndex]))
  print(table(glm.prediction.model.test.y, testData$mpg01))
  if(!getPredicts){
    cat("Ein: ", mean(glm.prediction.model.train.y != mpg01[trainingIndex]) ,"\n")
    cat("Eout: ", mean(glm.prediction.model.test.y != testData$mpg01) ,"\n")
  }
  else
    list(glm.prediction.model.test, glm.prediction.model.train)
}

glm.model1 = glm(formula = mpg01 ~ weight,
     data = datos, subset = i, family = "binomial")

glmModelErrorTest(glm.model1, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
##                            0 131  17
##                            1  22 143
##
## glm.prediction.model.test.y  0  1
##                            0 37  2
##                            1  6 34
## Ein:  0.1246006
## Eout:  0.1012658
```

En este segundo modelo, realizamos lo mismo, pero con *mpg* en función de *horsepower*.

```
glm.model2 = glm(mpg01 ~ horsepower, data = datos,
   subset = i, family = "binomial")

glmModelErrorTest(glm.model2, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
```

```
##                             0 126  17
##                             1  27 143
##
## glm.prediction.model.test.y  0  1
##                            0 31  4
##                            1 12 32
## Ein:  0.1405751
## Eout:  0.2025316
```

Como tercer modelo, tenemos la regresión logística de *mpg* en función de *displacement*.

```
glm.model3 = glm(mpg01 ~ displacement, data = datos,
  subset = i, family = "binomial")
glmModelErrorTest(glm.model3, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
##                             0 128   8
##                             1  25 152
##
## glm.prediction.model.test.y  0  1
##                            0 36  3
##                            1  7 33
## Ein:  0.1054313
## Eout:  0.1265823
```

En este modelo, realizamos algo más complejo, donde calculamos la regresión logística de *mpg* en función de *horsepower*, *displacement* y *weight*, estando relacionados los tres. Tras esto, al igual que en los anteriores, obtenemos las matrices de confusión, y calculamos el error dentro y fuera de la muestra.

```
glm.model4 = glm(mpg01 ~ weight*horsepower*displacement,
  data = datos, subset = i, family = "binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
glmModelErrorTest(glm.model4, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
##                             0 134  11
##                             1  19 149
##
## glm.prediction.model.test.y  0  1
##                            0 38  2
##                            1  5 34
## Ein:  0.09584665
## Eout:  0.08860759
```

En estos dos últimos modelos, realizamos una regresión logística con un modelo cuadrático, ajustando los datos. Una vez aprendido, predecimos con los datos de test, y calculamos el error dentor y fuera de la muestra. Estos dos modelos son los que calculamos en el apartado anterior:

```
glm.model5 = glm(mpg01 ~ weight*I(horsepower+displacement)^2,
  data = datos, subset = i, family="binomial")
glmModelErrorTest(glm.model5, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
```

```
##                             0 132  11
##                             1  21 149
##
## glm.prediction.model.test.y  0  1
##                            0 38  3
##                            1  5 33
## Ein:  0.1022364
## Eout:  0.1012658
```

En este caso, vemos que el modelo aprendido resulta ser un poco mejor que el modelo anterior, ya que, aunque el error dentro de la muestra sea peor en este modelo, vemos que el error fuera de la muestra mejora al modelo anterior. Y por último, vamos a probar con el último modelo que realizamos en el apartado anterior.

```
glm.model6 = glm(mpg01 ~ weight*I(horsepower*displacement*weight)^2,
  data = datos, subset = i, family = "binomial")
glmModelErrorTest(glm.model6, i, datos, datos.test)
```

```
##
## glm.prediction.model.train.y   0   1
##                             0 131  11
##                             1  22 149
##
## glm.prediction.model.test.y  0  1
##                            0 38  3
##                            1  5 33
## Ein:  0.1054313
## Eout:  0.1012658
```

Como vemos, el error dentro de la muestra en este caso viene a ser un poco peor que el modelo anterior, pero fuera de la muestra, se comportan más o menos igual, diferenciandose en el número de falsos positivos y falsos negativos, pero obteniendo el mismo error.

Por lo tanto, los mejores modelos son *glmmodel*5 y *glmmodel*6 ya que obtienen los errores más pequeños fuera de la muestra. Y, al elegir entre estos dos, cogeremos *glmmodel*5, ya que al ser el modelo más simple tiene preferencia si usamos el criterio de la *navaja de Ockham*.

- Ahora vamos a proceder a utlizar el KNN para ajustar el modelo. La función `knn` de la biblioteca *class*, realiza directamente la predicción de los datos, a diferencia de los modelos lineales anteriores. Para ello, la función toma como valores de entrada el conjunto de training, de test, las etiquetas del conjunto de training y el k, que inicialmente será $k = 1$.

Además de esto, para que el KNN funcione correctamente, tenemos que normalizar los datos, para que las dimensiones de uno no condicionen el resultado, al ser distintas.

```
set.seed(1)

normalize <- function(data) {
    apply(X=data, MARGIN=2, FUN=function(x) {
        max <- max(x)
        min <- min(x)
        sapply(X=x, FUN=function(xi) (xi-min)/(max-min))
    })
}
# normalizamos los datos

caracteristicas = data.frame(horsepower, weight, displacement)
names(caracteristicas) = c("horsepower", "weight", "displacement")
```

```
vs_train = data.frame(data.frame(normalize(caracteristicas[i,])), as.factor(mpg01[i]))
colnames(vs_train) = c("horsepower", "weight", "displacement", "mpg01")
vs_test = data.frame(data.frame(normalize(caracteristicas[-i,])), datos.test$mpg01)
colnames(vs_test) = c("horsepower", "weight", "displacement", "mpg01")

# esta función clasifica los datos con KNN
learnKNN <- function(train, test, label, testLabel, k, pintarM = T, useProb=F){
   knn.model=knn(train, test, label, k = k, prob = useProb)
   if(pintarM) print(table(knn.model, testLabel))
     knn.model
   }
knn1=learnKNN(train=subset(vs_train, select=-mpg01) , test=subset(vs_test, select=-mpg01),
  vs_train$mpg01, vs_test$mpg01, k = 1)
```

```
##          testLabel
## knn.model  0  1
##         0 38  3
##         1  5 33
```

```
error1 = mean(knn1 != datos.test$mpg01)
cat("Eout: ", error1, "\n")
```

```
## Eout:  0.1012658
```

Como vemos, aproximadamente el 90 % de las muestras están bien etiquetadas. Para ver si se mejora o no, vamos a cambiar el tamaño de k. Esto lo haremos con la función `tune.knn`, la cual, nos devolverá el modelo con el mejor k y el mejor k.

```
model.tune.knn = tune.knn(x=subset(vs_train, select=-mpg01), y=vs_train$mpg01, k=1:20)
print(model.tune.knn)
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  7
##
## - best performance: 0.07711694
```

En el resultado de la función, podemos ver cómo la función después de ajustar el modelo con los distintos k que puede haber en el intervalo que hemos establecido, escoge el k que ajusta mejor al modelo.

- Para pintar las curvas ROC, haremos uso del paquete *ROCR* de la siguiente manera, como se puede ver en el libro "*An Introduction to Statistical Learning*":

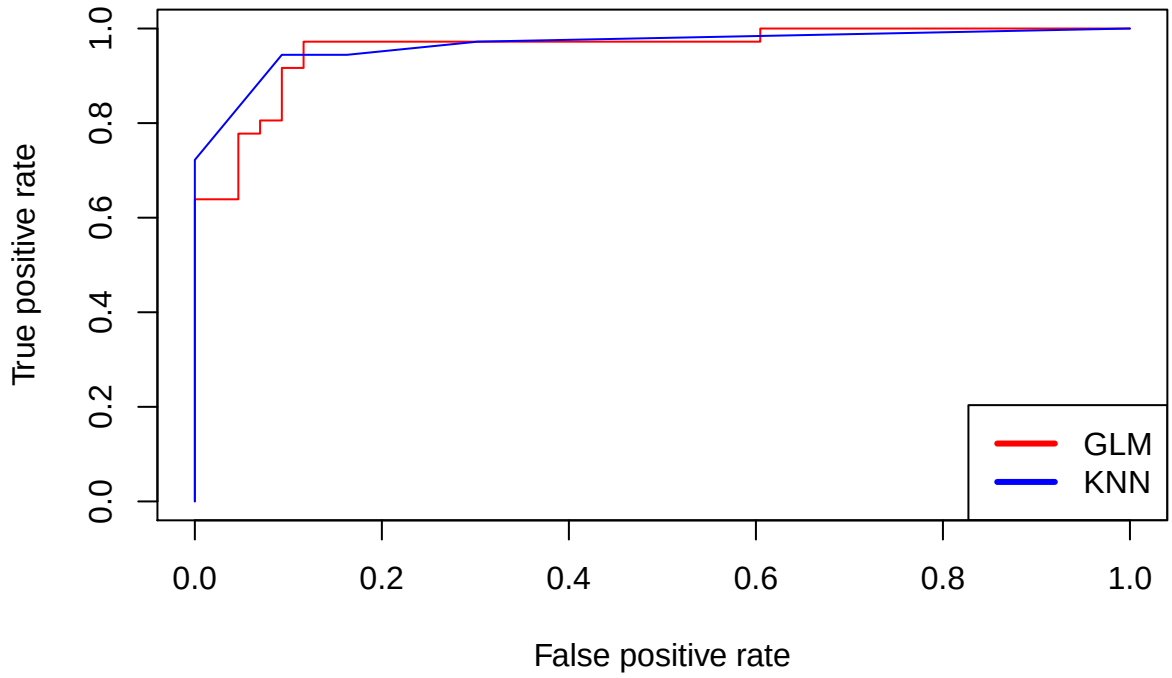
```
rocplot <- function(model, test, Add_plot = F, colour = "red"){
  y_pred = prediction(model, test)
  perf = performance(y_pred, "tpr", "fpr")

  plot(perf, add = Add_plot, col = colour)
}
pred=glmModelErrorTest(glm.model6, i, datos, datos.test, getPredicts = T)[1]
```

```
##
```

```
## glm.prediction.model.train.y   0   1
##                             0 131  11
##                             1  22 149
##
## glm.prediction.model.test.y  0  1
##                            0 38  3
##                            1  5 33
```

```
rocplot(model = pred, test = datos.test$mpg01)
# Curva ROC para KNN
bestKNN=learnKNN(train=subset(vs_train, select=-mpg01) , test=subset(vs_test, select=-mpg01),
  vs_train$mpg01, vs_test$mpg01, k = model.tune.knn$best.model$k, useProb = T, pintarM = F)
probKNN = attr(bestKNN, "prob")
probKNN = ifelse(bestKNN == 0, 1 - probKNN, probKNN)
rocplot(model = probKNN, test = datos.test$mpg01, Add_plot = T, colour = "blue")
legend('bottomright', c("GLM","KNN"), col=c('red', 'blue'), lwd=3)
```



Como podemos ver en el resultado de la ejecución, la curva ROC de KNN es un poco mejor que la de GLM, con lo que obtenemos que el modelo KNN es un poco mejor que regresión para este conjunto de datos, ya que ajusta mejor los datos y clasifica mejor que GLM.

- Para estimar el error usando validación cruzada, usaremos la librería `boot`. Como $k = 5$, haremos un bucle de 1 a 5, e iremos almacenando los errores de CV.

```
attach(Auto)
```

```
## The following objects are masked from datos:
##
##     acceleration, cylinders, displacement, horsepower, mpg,
##     origin, weight, year
```

```
## The following objects are masked from Auto (pos = 4):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year
```

```
## The following object is masked from package:ggplot2:
##
##     mpg
```

```
lm.fit = glm(mpg01 ~ weight*I(horsepower*displacement*weight)^2,
  data = datos, family = "binomial")
cv.error.5 = cv.glm(data = datos, glmfit=lm.fit, K=5)$delta[1]

print(cv.error.5)
```

```
## [1] 0.08130801
```

Como vemos, estos son los errores que podemos calcular con validación cruzada usando regresión lineal, y con el cuarto modelo de regresión calculado. A continuación, podemos ver los errores de CV para el KNN:

```
model.tune.knn = tune.knn(x=subset(vs_train, select=-mpg01),
    y=vs_train$mpg01, k=1:20, tunecontrol=tune.control(cross=5))
print(model.tune.knn)
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 5-fold cross validation
##
## - best parameters:
##  k
##  8
##
## - best performance: 0.07332309
```

- En este caso, el mejor modelo obtenido es el modelo KNN donde hemos podido comprobar a lo largo del ejercicio, como ajusta mejor los datos y obtiene mejores resultados que el resto. Además, podemos comprobarlo facilmente en la curva ROC.

2. Ejercicio 2

Usar la base de datos Boston (en el paquete MASS de R) para ajustar un modelo que prediga si dado un suburbio este tiene una tasa de criminalidad (crim) por encima o por debajo de la mediana. Para ello considere la variable crim como la variable salida y el resto como variables predictoras.

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

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

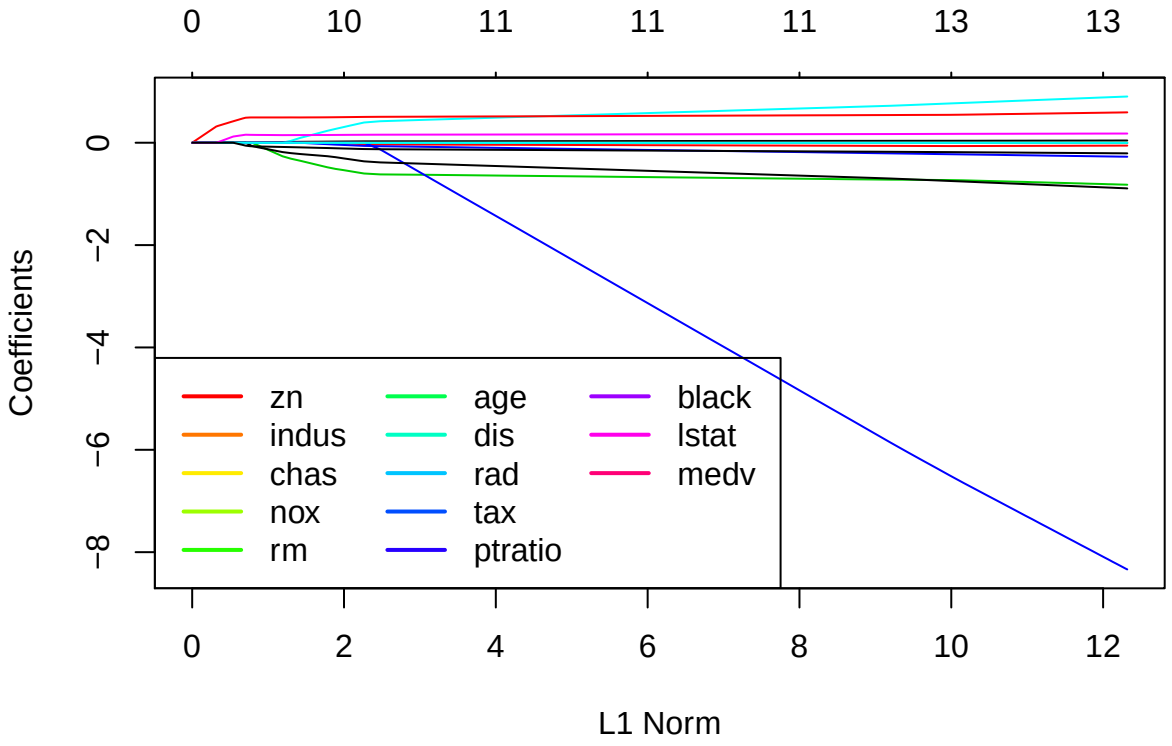
Bonus-3: Estimar el error de entrenamiento y test por validación cruzada de 5 particiones.

a) Para realizar la selección del subconjunto ótpimo, haremos uso de la función `glmnet`, con el parámetro `alpha = 1`.

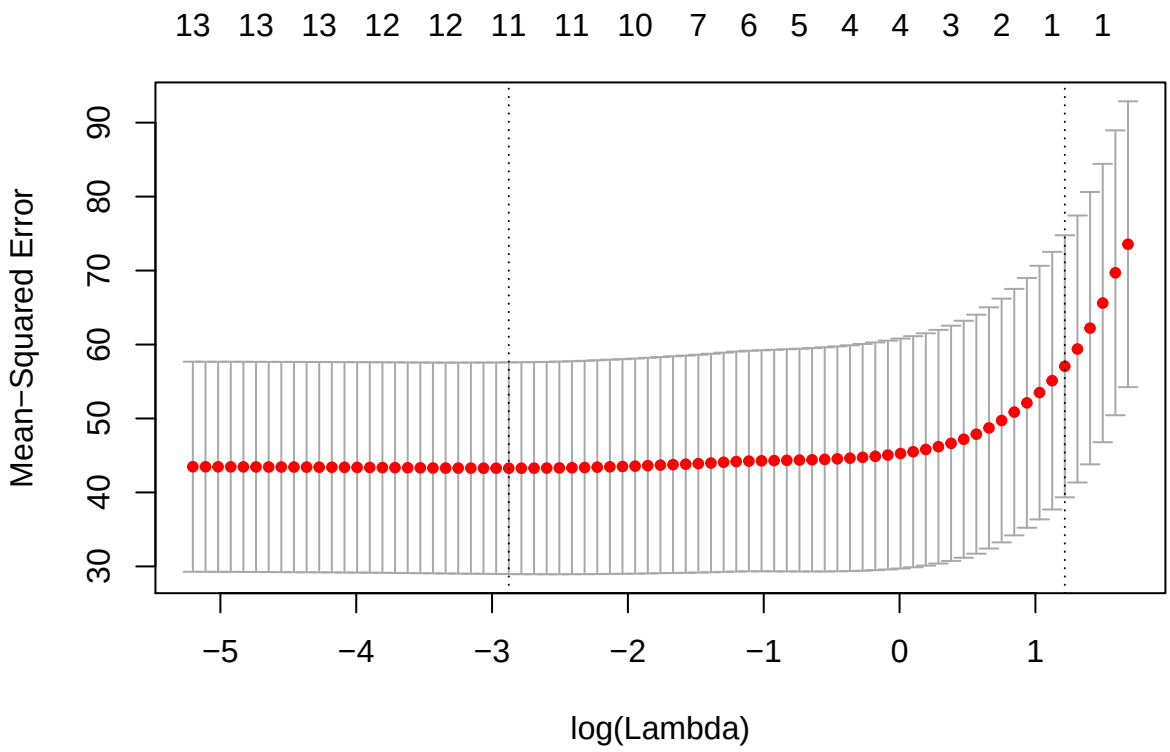
```r
grid =10^seq(10, -2, length=length(Boston$crim))

datos = model.matrix(Boston$crim ~., Boston)[,-1]
y_datos = as.vector(Boston$crim)
# indices para los datos de train
i = sample(x=nrow(datos), size=floor(nrow(datos))*0.8)

lasso.mod = glmnet(datos[i,], y_datos[i], alpha = 1, lambda = grid)
plot(lasso.mod)
legend('bottomleft', rownames(lasso.mod$beta),
  col=rainbow(n=length(rownames(lasso.mod$beta))), lwd=2, ncol = 3)
```




```r
cv.out = cv.glmnet(datos, y_datos, alpha = 1)
plot(cv.out)
```




```
mejorLambda = cv.out$lambda.min
lasso.pred = predict(lasso.mod, s = mejorLambda, newx=datos[-i,])
mean(lasso.pred - y_datos[-i])
```

```
## [1] 0.3521201
```

```
out = glmnet(datos, y_datos, lambda = grid, alpha = 1)
lasso.coef = predict(out, type="coefficients", s = mejorLambda)
subgrupo = abs(lasso.coef[,1])
names(subgrupo[subgrupo >= 0.5])
```

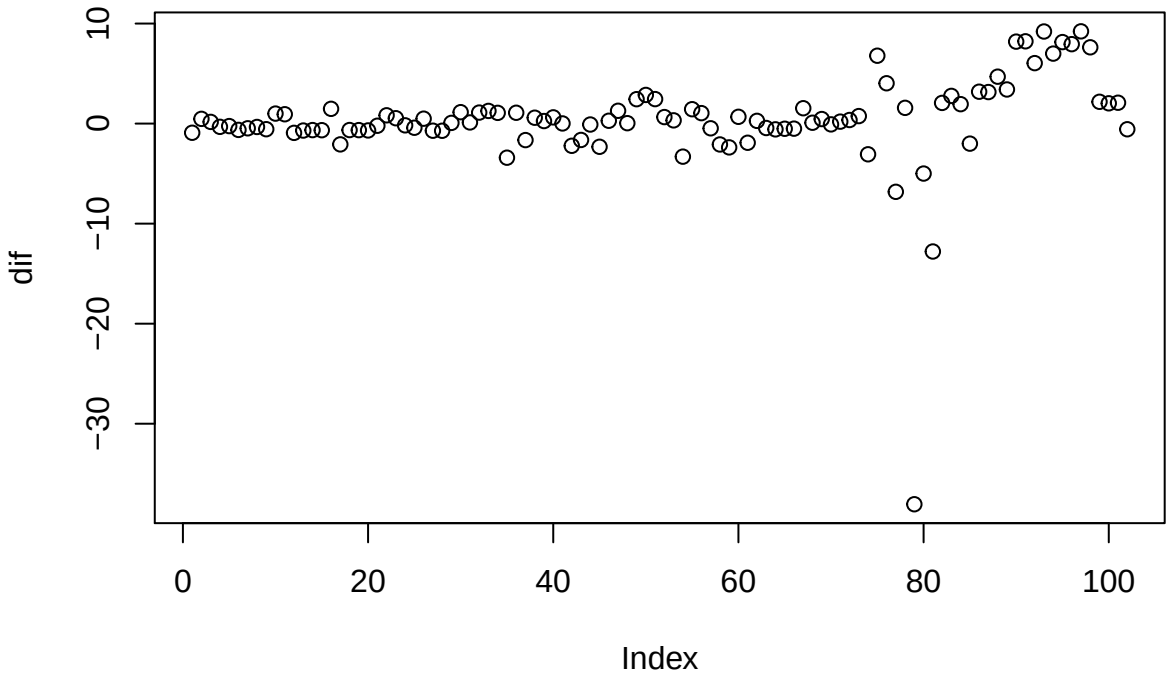
```
## [1] "(Intercept)" "chas"        "nox"         "dis"         "rad"
```

Como vemos, el número de variables ha bastante reducido, quedando solo las más relevantes, dentro del umbral establecido, tras haber realizado un cálculo inicial, con los datos regularizados, para unos datos de training.

A continuación, obtenemos un λ realizando validación cruzada para predecir los datos y vemos la media de errores cuadráticos en los datos de test. Por último, tras haber escogido el mejor λ volvemos a llamar al método `glmnet` y realizamos la predicción con el mejor λ para obtener los coeficientes para cada variable, con la opción `type = "coefficients"`, y nos quedamos con los que superen el umbral.

b) Para hacer la *ridge-regression*, haremos uso de los datos de antes, ya preparados, gracias a la función `model.matrix`.

```
attach(Boston)
x = cbind(chas, nox, dis, rad)
ridge.mod = glmnet(x[i,], y_datos[i], alpha=0, lambda = grid)
ridge.pred = predict(ridge.mod, s = mejorLambda, newx=x[-i,])
dif = ridge.pred - y_datos[-i]
plot(dif)
```




```
mean(dif)
```

```
## [1] 0.365185
```

Como podemos ver en la gráfica, el modelo es capaz de ajustar bastate bien alrededor del 60 % de los datos de test, mientras que en el resto, vemos como hay indicios de *underfitting* ya que se empiezan a producir errores en los datos, teniendo algunos errores considerables entre el modelo ajustado y el real, pero puede ser que el dato que ofrece el mayor error con diferencia a los demás, puede ser que sea un *outlayer*.

c) Para hacer la nueva variable, procederemos de forma similiar a como se hacía en el ejercicio 1 para calcular $mpg01$.

```
crimMean = mean(crim)
newCrim = (crim >= crimMean)*1
newCrim[newCrim == 0] = -1
datos = data.frame(Boston,as.factor(newCrim))
colnames(datos)[ncol(datos)]="newCrim"
attach(datos)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     newCrim
```

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

```
SVM <- function(kernel = "linear", control = 10){
  # Realizamos la llamada al SVM
  svmfit = tune(svm, newCrim ~., data=datos, kernel = kernel,
      ranges = list(cost=c(0.001,0.01,0.1,1,5,10,100,1000)),
      tunecontrol=tune.control(cross=control))
  svmfit.bestmodel = svmfit$best.model
  ys = predict=predict(svmfit.bestmodel, data.frame(datos[-i,]))
```

```
  print(table(predict=ys, truth=datos$newCrim[-i]))
  cat("CV error: ", svmfit$best.performance ,"\n")
  cat("Eout: ", mean(ys != datos[-i,]$newCrim) ,"\n")
}

SVM()
```

```
##        truth
## predict -1  1
##      -1 79  0
##      1   0 23
## CV error:  0.009882353
## Eout:  0
```

Como vemos, el ajuste que produce SVM es de una calidad muy superior a los modelos, ya que el modelo ajusta los datos según el kernel elegido, y prueba con la función **tune** los distintos rangos que aparecen en la lista del parámetro **ranges**. Para estimar el error del modelo, utiliza validación cruzada, y se queda con el modelo que obtiene mejor error, (almacenado en la variable *best.perfomance*).

Tras realizar las pruebas con los distintos parámetros, escogemos el mejor modelo de todos los modelos posibles, con *svmfit.bestmodel = svmfit$best.model*. A continuación, mostramos la matriza de confusión y como podemos ver, los datos de la matriz nos indica que el error utilizando *SVM* es muy pequeño, y al mostrar el error de validación cruzada en la muestra y el error fuera de la muestra, vemos que ajusta muy bien los datos.

En un principio, vemos que con el modelo lineal generaliza muy bien los datos y no solo eso, sino que los ajusta muy bien también para ser un kernel tan "simple". Pero podemos ver si cambiando de kernel, ajustamos mejor los datos.

```
# KERNEL POLYNOMIAL
SVM(kernel="polynomial")
```

```
##        truth
## predict -1  1
##      -1 79  0
##      1   0 23
## CV error:  0.01388235
## Eout:  0
```

Como podemos ver, en un principio, para este set de datos, cambiar entre un kernel lineal y polinómico, no supone un cambio sustancial, por lo que podemos seguir eligiendo el kernel lineal.

```
#KERNEL DE BASE RADIAL
SVM(kernel="radial")
```

```
##        truth
## predict -1  1
##      -1 79  0
##      1   0 23
## CV error:  0.01188235
## Eout:  0
```

```
#KERNEL BASADO EN SIGMOIDES
SVM(kernel="sigmoid")
```

```
##        truth
## predict -1  1
##      -1 78  0
##      1   1 23
```

```
## CV error:  0.01180392
## Eout:  0.009803922
```

Como podemos ver, cambiar de kernel para este set de datos, con validación cruzada de 10 particiones, no suponen mejoría ante un kernel lineal, utilizando los parámetros por defecto, a excepción del kernel con base radial, que es capaz de ajustar un buen modelo también.

d) Para estimar el error de test y el error de validación cruzada con 5 particiones, podemos usar la misma función que en el apartado anterior, pero, cambiando el parámetro `tunecontrol`, haciendo que tome el valor 5, ya que por defecto realiza una partición de $K = 10$.

Para ello, en la función *SVM* implementada, cambiaremos el valor del parámetro `control` a 5 para que se realice una validación cruzada con $k = 5$.

```
SVM(kernel="linear",control=5)
```

```
##        truth
## predict -1  1
##      -1 79  0
##      1   0 23
## CV error:  0.009862163
## Eout:  0
```

Como vemos, el error que se produce es mayor que si hacemos una validación cruzada con un k más pequeño ya que el modelo entrena con menos datos y realiza las pruebas de test con conjuntos de datos más grandes, con lo que la probabilidad de error se hace mayor.

3. 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 %).**
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.**
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.**
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.**

a) Para realizar la partición en datos de training y test, seguiremos el mismo método que antes, obteniendo un vector de índcies aleatorio.

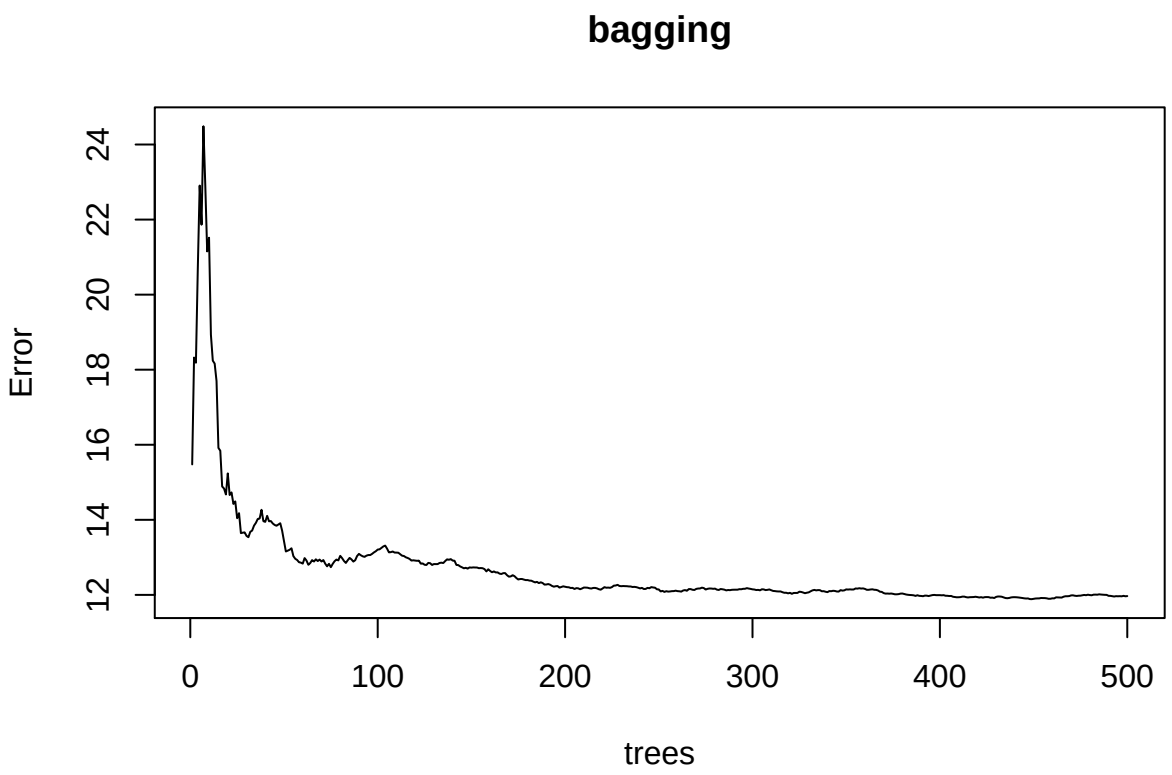
b) Para ajustar el modelo de regresión usando bagging hacemos lo siguiente:

```
set.seed(1)
datos = Boston
attach(datos)
```

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

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

```r
# Indices de los datos de training
i = sample(x=nrow(datos), size=floor(nrow(datos))*0.8)
#-----------------------------------------------------------------
# - subset => indices del conjunto de entrenamiento
# - mtry => número de variables predictoras para cada nodo
# - importance => parámetro que indica si la función debe tener
#   en cuenta la importancia que tienen los distintos parámetros
#   para ajustar el modelo.
bagging = randomForest(medv ~ ., data=datos, subset = i, mtry=length(datos)-1, importance=T)
plot(bagging)
```

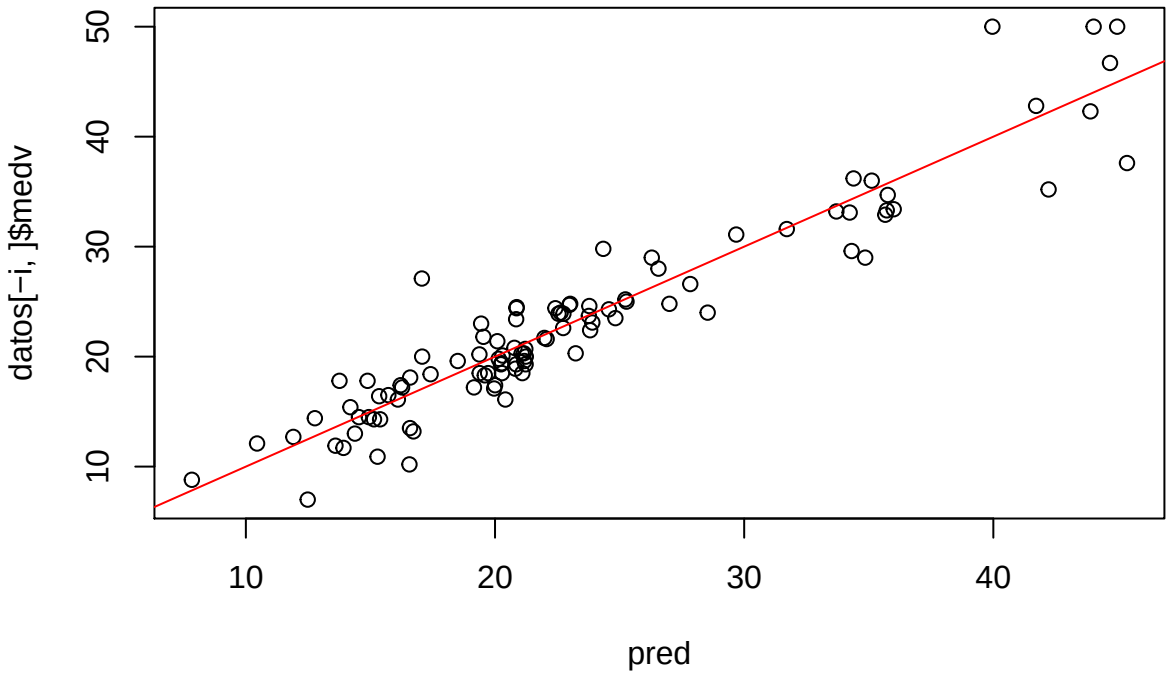



```r
print(bagging)
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = datos, mtry = length(datos) -      1, importance = T, subse
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.968
##                     % Var explained: 86.08
```

En la gráfica anterior vemos cómo conforme se van aumentando el número de árboles va disminuyendo el error, hasta que más o menos se mantiene constante alrededor de un error con valor 12, aproximadamente. Para predecir el error de test, usaremos los índices anteriores y el modelo aprendido, como se ve a continuación:

```r
pred = predict(bagging, newdata=datos[-i,])
plot(pred, datos[-i,]$medv)
abline(0,1,col="red")
```




```
Eout = mean((pred - datos[-i,]$medv)^2)
cat("Eout = ", Eout, "\n")
```

```
## Eout =  8.385359
```

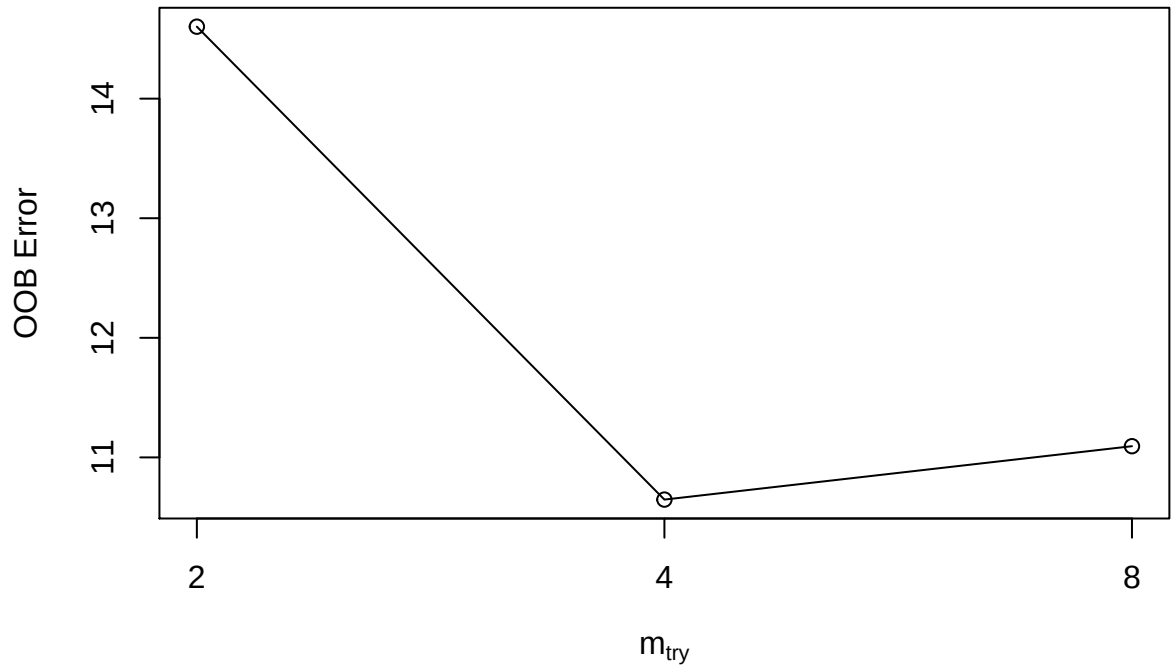
En la gráfica podemos ver cómo los datos difieren entre la predicción y los datos reales, viendo el error que se produce. Si el error fuera 0, todos los datos se situarían sobre la línea de la gráfica.

c) Para ajustar un modelo randomForest, usaremos la función *tuneRF* que al igual que hacíamos en apartados anteriores, para obtener el mejor valor de *mtry* posible para el conjunto de datos dado. *tuneRF* busca el modelo óptimo haciendo uso del error *Out-of-Bag*.

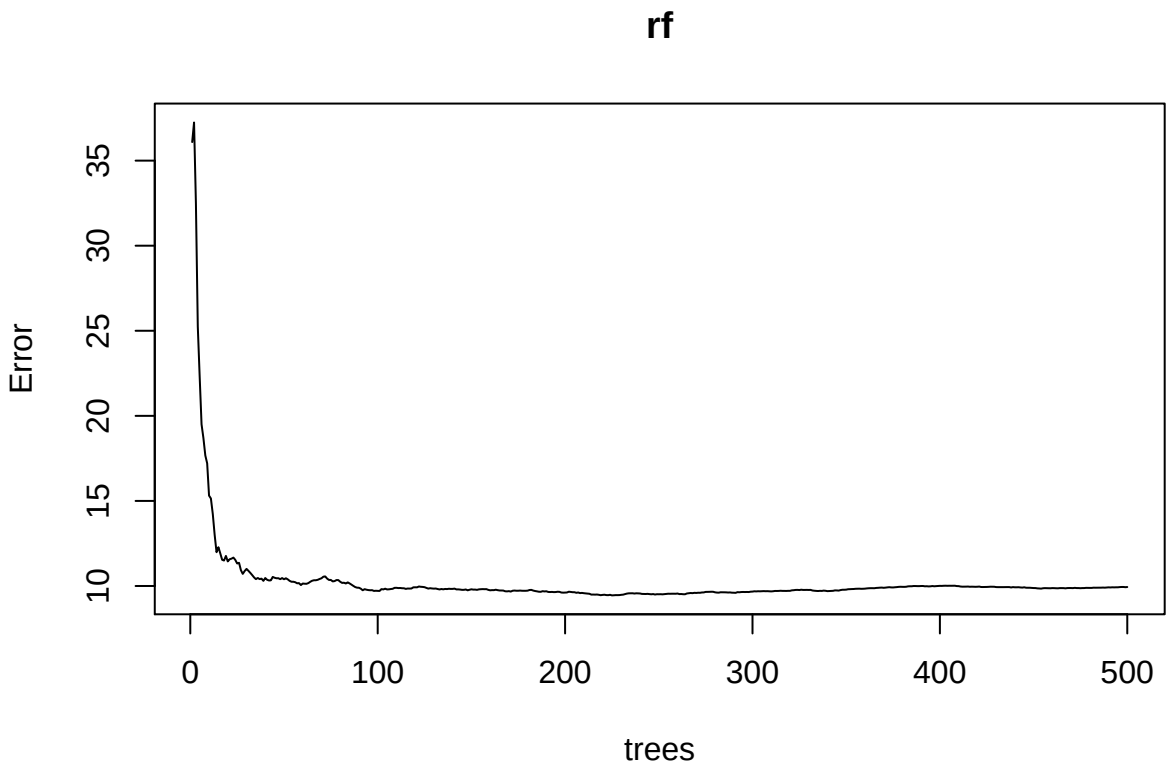
Este valor *mtry* es el número de variables que se escogen aleatoriamente para particionar los datos en cada árbol. Por defecto en problemas de regresión usa $\frac{p}{3}$ donde p es el número de variables predictoras de las que disponemos.

```
# Con el parámetro "doBest" devolvemos directamente el mejor modelo
rf=tuneRF(x=subset(datos, select=-medv), y=datos$medv, doBest=T)
```

```
## mtry = 4  OOB error = 10.64729
## Searching left ...
## mtry = 2     OOB error = 14.60176
## -0.3714053 0.05
## Searching right ...
## mtry = 8     OOB error = 11.09366
## -0.04192266 0.05
```




```
#En la gráfica podemos ver como el mejor modelo es con mtry = 4
# Y podemos haceder al randomForest de la siguiente manera
rf.best = rf$forest
plot(rf)
```

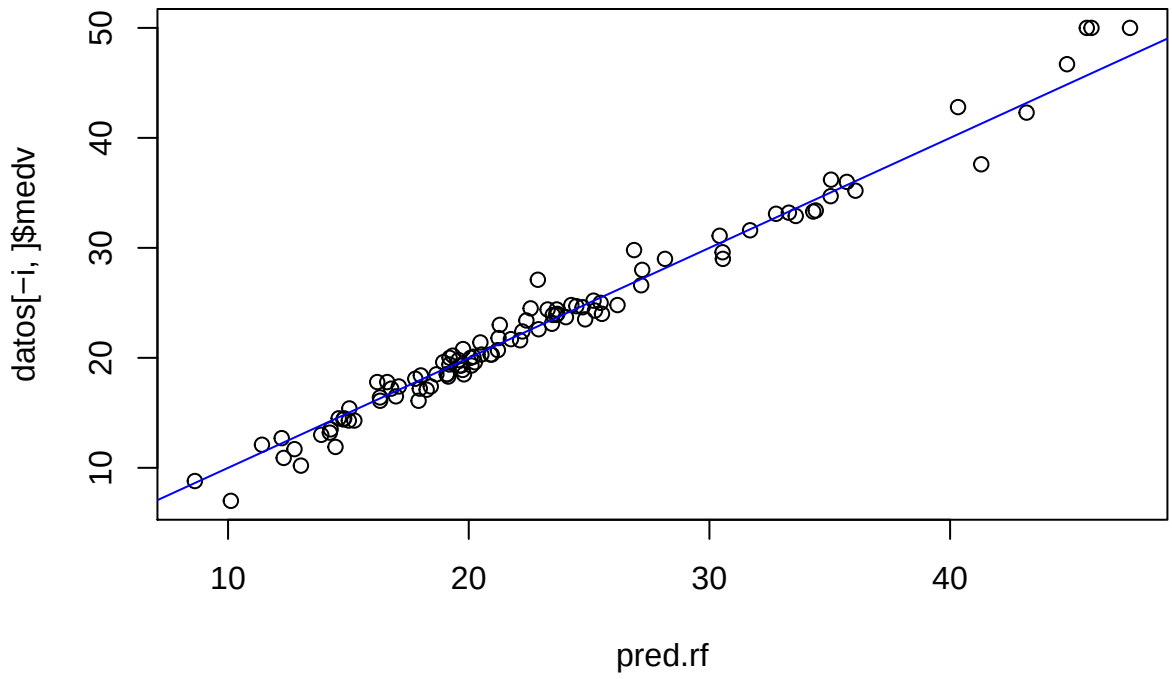



```
print(rf)
```

```
##
## Call:
```

```
##  randomForest(x = x, y = y, mtry = res[which.min(res[, 2]), 1])
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 9.938944
##                     % Var explained: 88.23
```

```
pred.rf = predict(rf, newdata=datos[-i,])
plot(pred.rf, datos[-i,]$medv)
abline(0,1,col="blue")
```




```
Eout.rf = mean((pred.rf - datos[-i,]$medv)^2)
cat("Eout = ", Eout.rf, "\n")
```

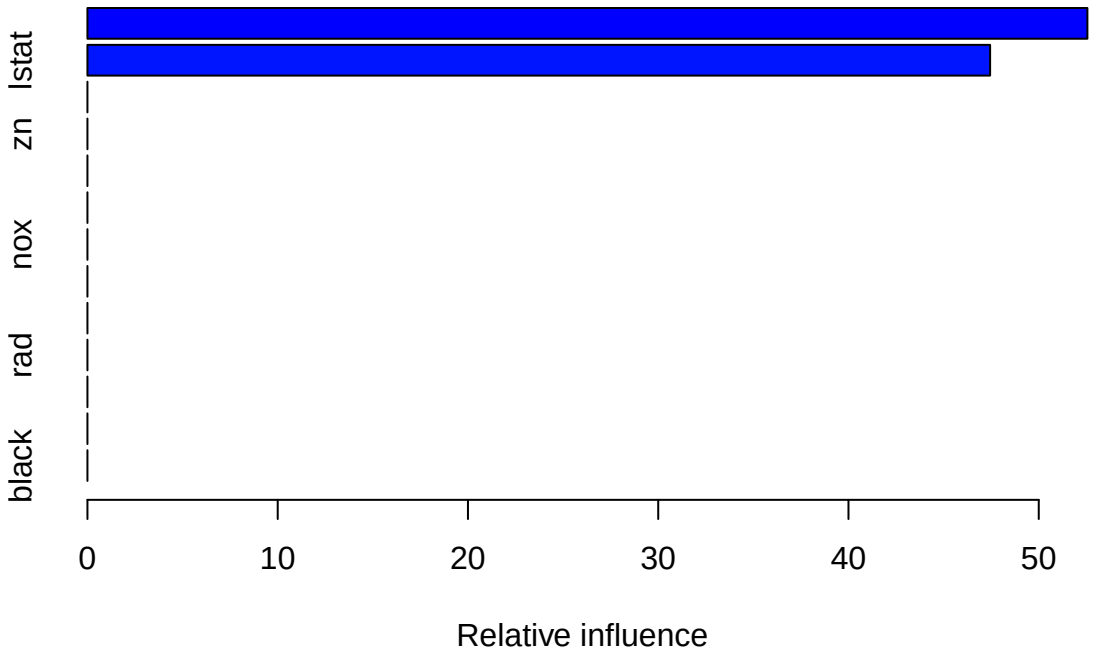
```
## Eout =  1.712158
```

En la gráfica y en el error que obtenemos fuera de la muestra, vemos como el modelo de randomForest es altamente superior al modelo de bagging, ya que mientras que en el modelo anterior teníamos un $E_{out} \approx 8,3$, en este modelo nos quedamos muy por debajo, viendo cómo se ajustan mejor los datos.

Respecto al número de árboles que usa el modelo, en las gráficas anteriores podemos ver como a medida que va creciendo el número de árboles, baja también el error de training, pero llega un momento en el que por más que aumentemos el número de árboles de los que dispone el entrenamiento, no conseguimos hacer mejoras significativas. Además, si no hacemos más que incrementar el número de árboles, empezaremos a ajustar también el ruido que puedan tener los datos, por lo que empezaríamos a tener problemas de sobreajuste en el modelo.

d) Para realizar un modelo haciendo uso de la técnica de boosting, haremos uso del paquete *gbm* y de la función *gbm*, que estimará *medv* con todas las variables predictoras, se quedará con aquellas que más importancia tengan. Habrá que establecer el parámetro *distribution* a *gaussian* ya que es un problema de regresión

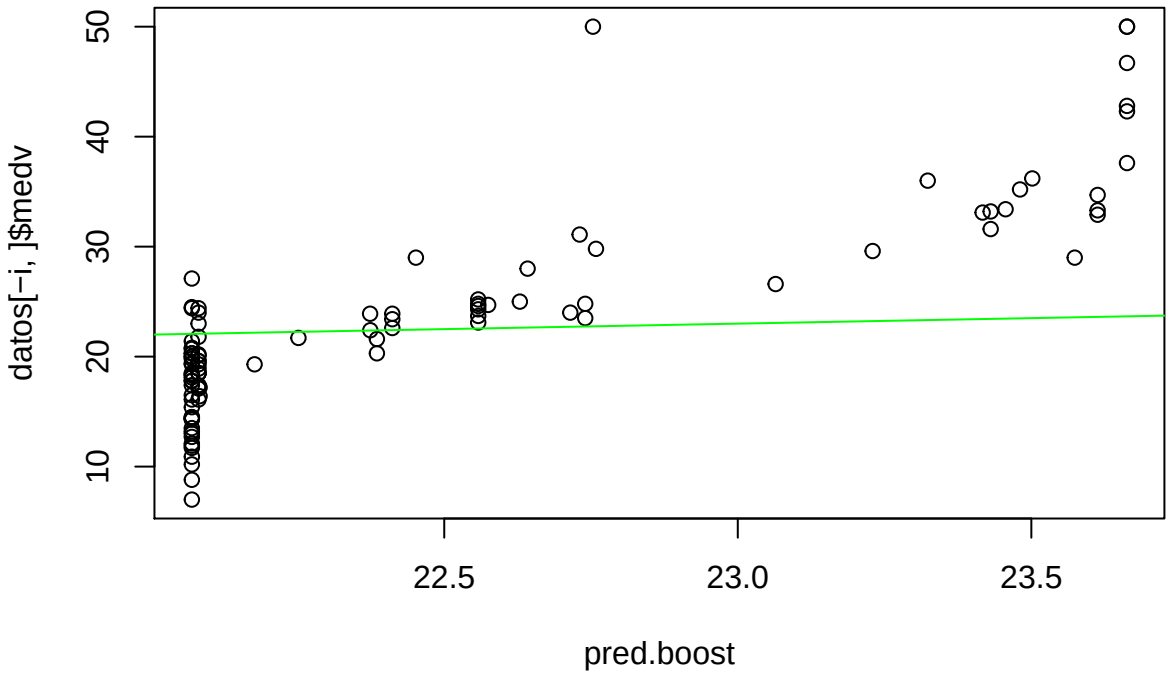
```
boosting = gbm(medv ~., data=datos[i,], distribution = "gaussian",cv.folds = 10)
summary(boosting)
```




```
##             var  rel.inf
## rm           rm 52.55646
## lstat     lstat 47.44354
## crim       crim  0.00000
## zn           zn  0.00000
## indus     indus  0.00000
## chas       chas  0.00000
## nox         nox  0.00000
## age         age  0.00000
## dis         dis  0.00000
## rad         rad  0.00000
## tax         tax  0.00000
## ptratio ptratio  0.00000
## black     black  0.00000
```

En la gráfica podemos ver cómo *lstat* y *rm* son las variables más importantes para predecir el modelo y justo debajo de ella, el valor de importancia de cada una de las variables. Ahora, vamos a ver cómo se comporta el modelo fuera de la muestra.

```
pred.boost = predict(boosting , newdata = datos[-i,] , n.trees =boosting$n.trees)
plot(pred.boost, datos[-i,]$medv)
abline(0,1,col="green")
```



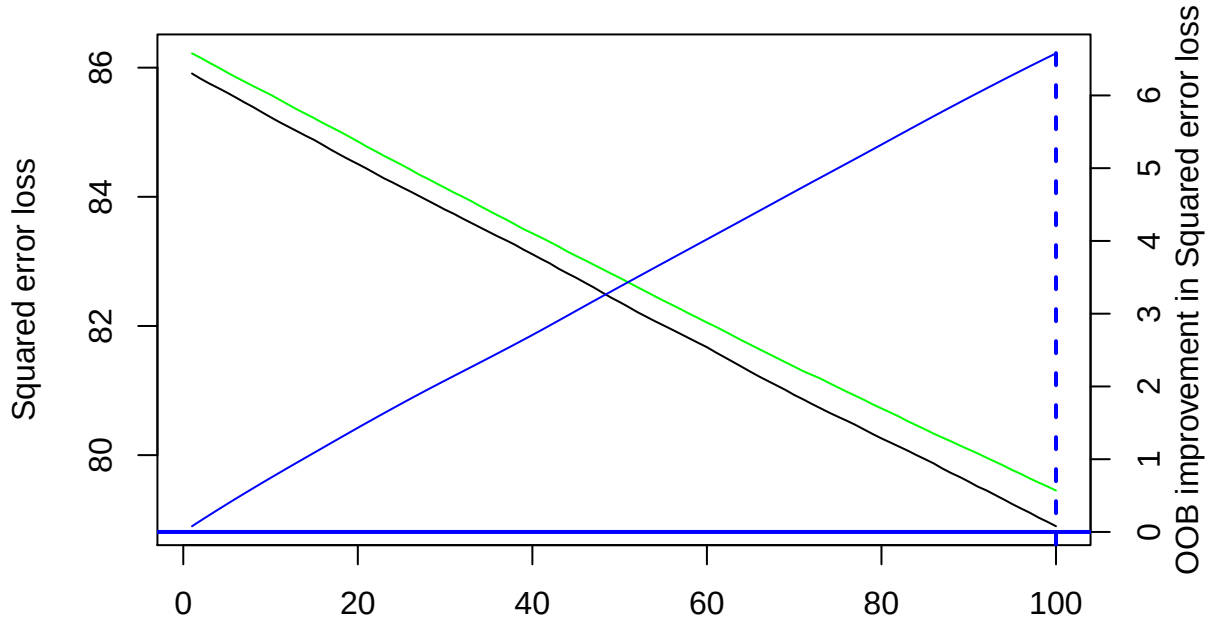

```
Eout.boosting = mean((pred.boost - datos[-i,]$medv)^2)
cat("Eout = ", Eout.boosting, "\n")
```

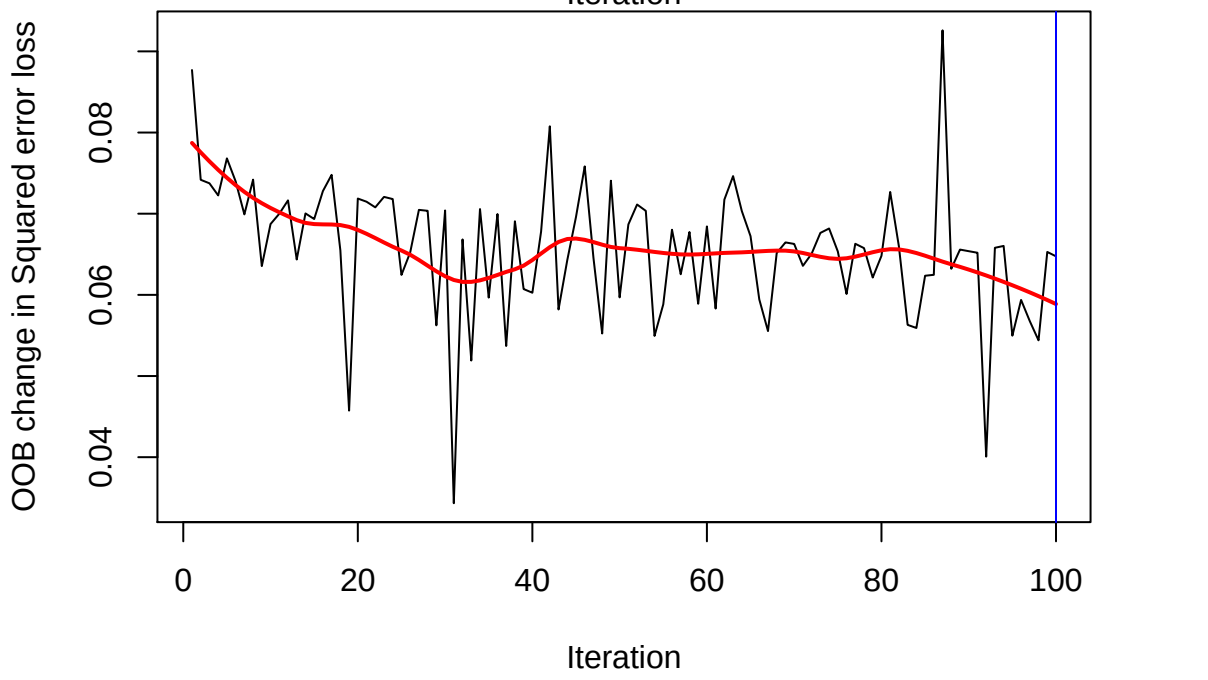
```
## Eout =  70.20361
```

Como vemos, el error fuera de la muestra es terrible en comparación con los otros modelos, y casi con cualquiera. Esto se debe a que el modelo en sí, utiliza una cantidad muy pequeña de árboles para aprender el modelo, con lo que no es capaz de generar un buen modelo.

Para solucionar esto, haremos uso de *gbmperf* que estima el número óptimo de iteraciones que necesita un modelo de boosting.

```
gbm.perf(boosting, oobag.curve = T, method="cv")
```

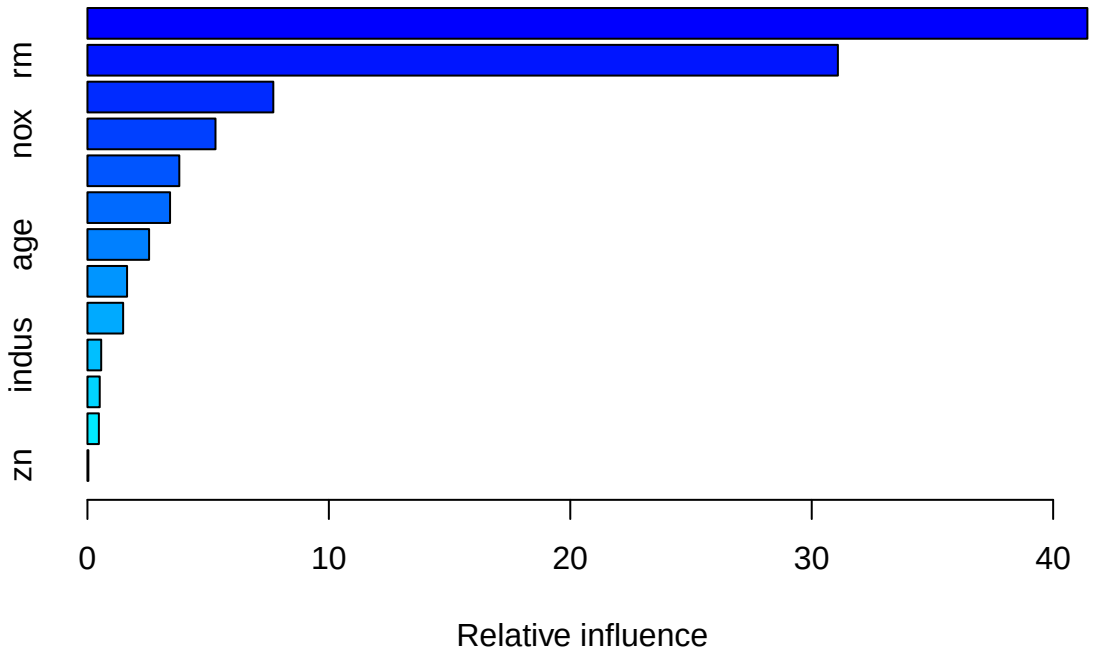





```
## [1] 100
```

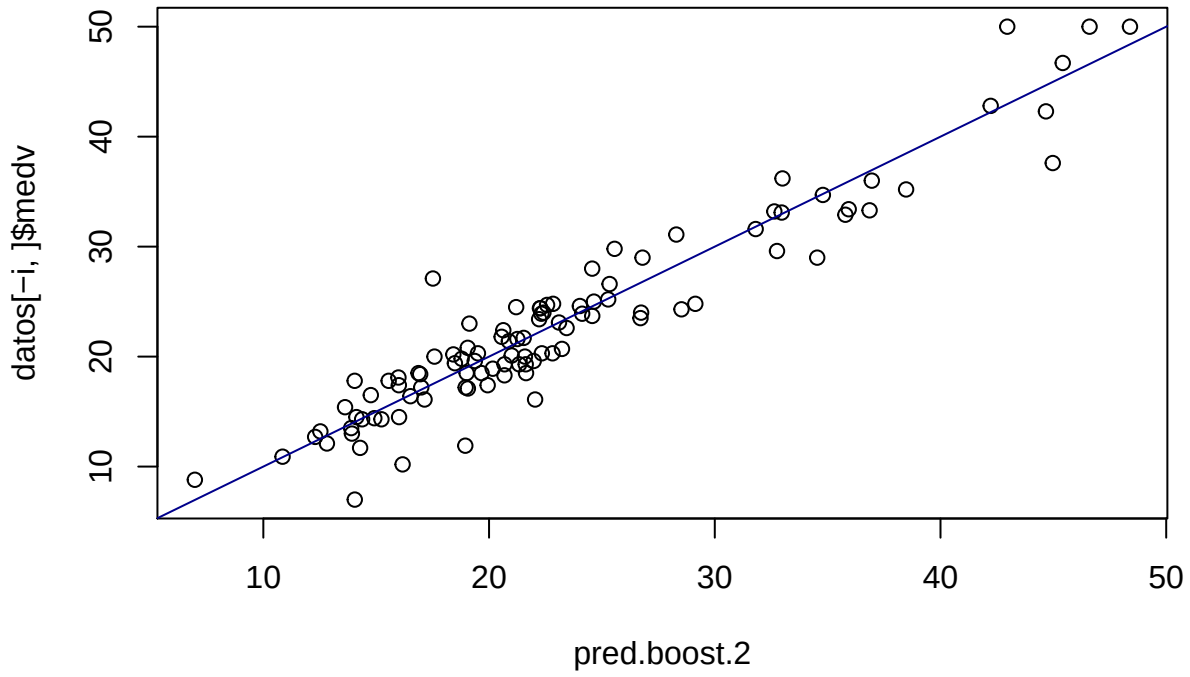
Como vemos, el resultado es que el número de árboles que nos indica como óptimo es 100, al igual que el número de árboles que ha generado boosting, por lo que, esto quiere decir, que necesitamos una mayor cantidad de árboles para ajustar el modelo, y que la profundida del árbol no es correcta.

```
boosting.model2 = gbm(medv ~., data=datos[i,], distribution = "gaussian",
    cv.folds = 10, n.trees = 15000, interaction.depth = floor(sqrt(ncol(datos))))
summary(boosting.model2)
```




```
##             var     rel.inf
## lstat     lstat 41.41853070
## rm           rm 31.08434141
## dis         dis  7.69871560
## nox         nox  5.30455585
## crim       crim  3.80508084
## ptratio ptratio  3.42216162
## age         age  2.55348402
## black     black  1.64176881
## tax         tax  1.47859264
## indus     indus  0.57043091
## chas       chas  0.50956355
## rad         rad  0.47381398
## zn           zn  0.03896005
```

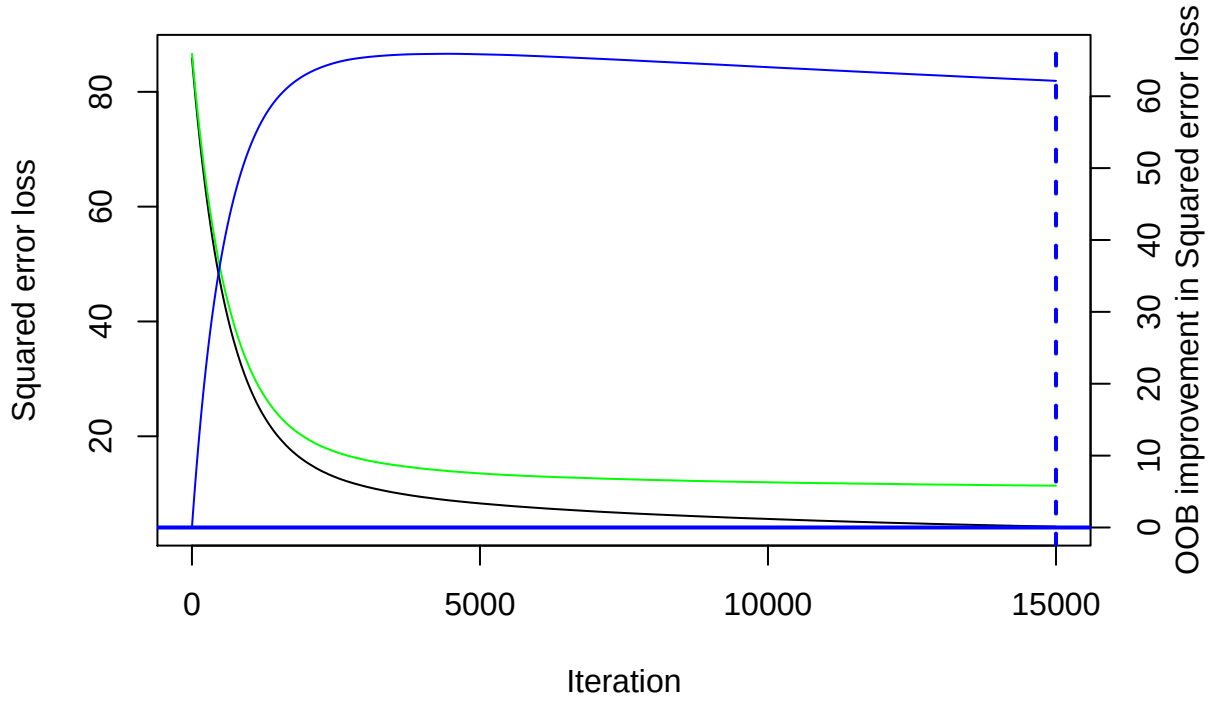
```
pred.boost.2 = predict(boosting.model2, newdata = datos[-i,] , n.trees =boosting.model2$n.trees)
plot(pred.boost.2, datos[-i,]$medv)
abline(0,1,col="darkblue")
```

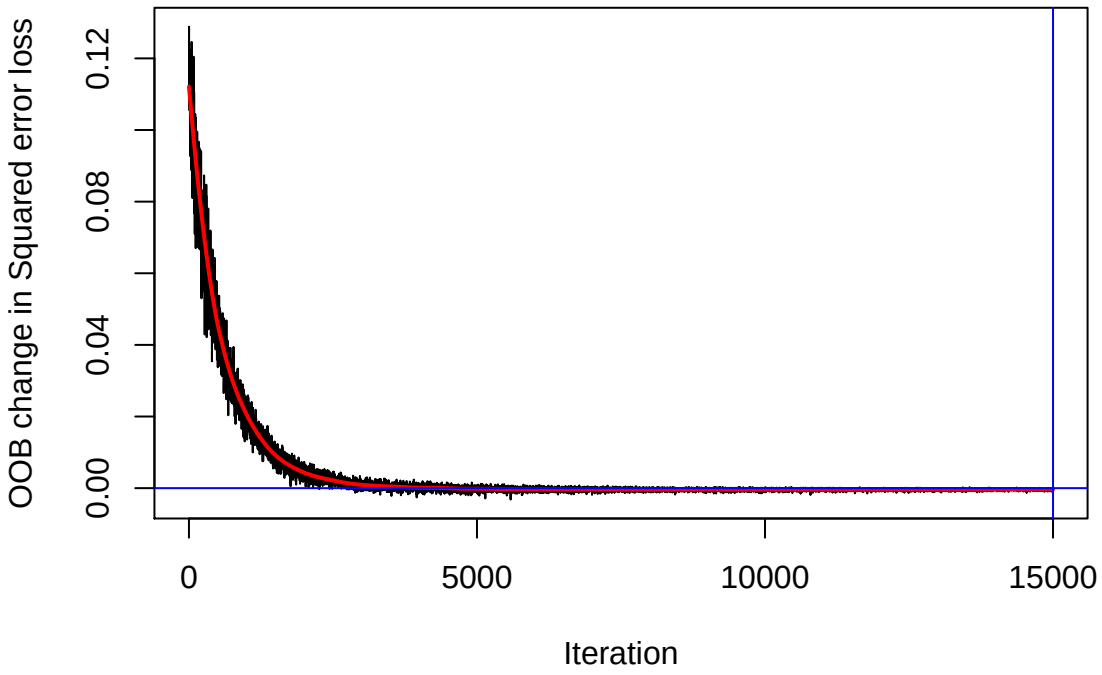



```
Eout.boosting.2 = mean((pred.boost.2 - datos[-i,]$medv)^2)
cat("Eout = ", Eout.boosting.2, "\n")
```

```
## Eout =  7.559393
```

```
gbm.perf(boosting.model2, oobag.curve = T, method="cv")
```






```
## [1] 15000
```

Como vemos, aumentando 100 veces el número de árboles máximo, el ajuste a los datos es significativamente mayor, pero, como contrapartida, el coste computacional que tiene es bastante mayor que otros modelos, pero a cambio obtenemos un modelo bastante mejor. También debemos hacer que la profundidad del árbol sea pequeña, y como cota superior se establece $\lfloor \sqrt{p} \rfloor$ donde p es el número de variables predictoras que disponemos.

Comparando con los otros modelos, el boosting requiere de un coste computacional mucho mayor que los otros modelos, para obtener unos errores peores que en los otros modelos, quedando de estos tres como claro ganador randomForest, ya que en no requiere de un coste computacional muy alto, y obtiene unos resultados muy muy buenos.

4. Ejercicio 4

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

1. **Crear un conjunto de entrenamiento conteniendo una muestra aleatoria de 800 observaciones, y un conjunto de test conteniendo el resto de 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).**
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.**
3. **Crear un dibujo del árbol e interpretar los resultados (0.5 puntos)**
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?**
5. **Aplicar la función cv.tree() al conjunto de "training" y determinar el tamaño óptimo del árbol. ¿Qué hace cv.tree?**
6. **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?

1) Vamos a proceder a realizar el conjunto de test para el set OJ.

```
datos = data.frame(OJ)
training = sample(x=nrow(datos), size=800)
```

Para generar el árbol de entrenamiento, usamos la función *tree* de la biblioteca *tree*.

```
attach(datos)
tr = tree(Purchase ~ ., data=datos[training,])
print(tr)
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1072.00 CH ( 0.60750 0.39250 )
##    2) LoyalCH < 0.5036 350  413.20 MM ( 0.27714 0.72286 )
##      4) LoyalCH < 0.280875 166  122.10 MM ( 0.12048 0.87952 ) *
##      5) LoyalCH > 0.280875 184  250.20 MM ( 0.41848 0.58152 )
##       10) PriceDiff < 0.115 80   80.06 MM ( 0.20000 0.80000 ) *
##       11) PriceDiff > 0.115 104  141.00 CH ( 0.58654 0.41346 ) *
##    3) LoyalCH > 0.5036 450  357.10 CH ( 0.86444 0.13556 )
##      6) LoyalCH < 0.764572 188  217.80 CH ( 0.73404 0.26596 )
##       12) ListPriceDiff < 0.235 76  105.40 MM ( 0.50000 0.50000 )
##         24) PctDiscMM < 0.196196 56   73.00 CH ( 0.64286 0.35714 ) *
##         25) PctDiscMM > 0.196196 20   13.00 MM ( 0.10000 0.90000 ) *
##       13) ListPriceDiff > 0.235 112   76.27 CH ( 0.89286 0.10714 ) *
##      7) LoyalCH > 0.764572 262   91.28 CH ( 0.95802 0.04198 ) *
```

Como podemos ver en el resultado del código anterior, vemos en el resultado del árbol, como para cada una de las variables predictoras tenemos una probabilidad que nos clasifica en MM o CH según el valor de entrada, y el número de datos que se han clasificado siguiendo el árbol según el número de variables usadas para clasificar.

2) Ahora vemos los resultados que nos dan la función *summary*

```
s = summary(tr)
print(s)
```

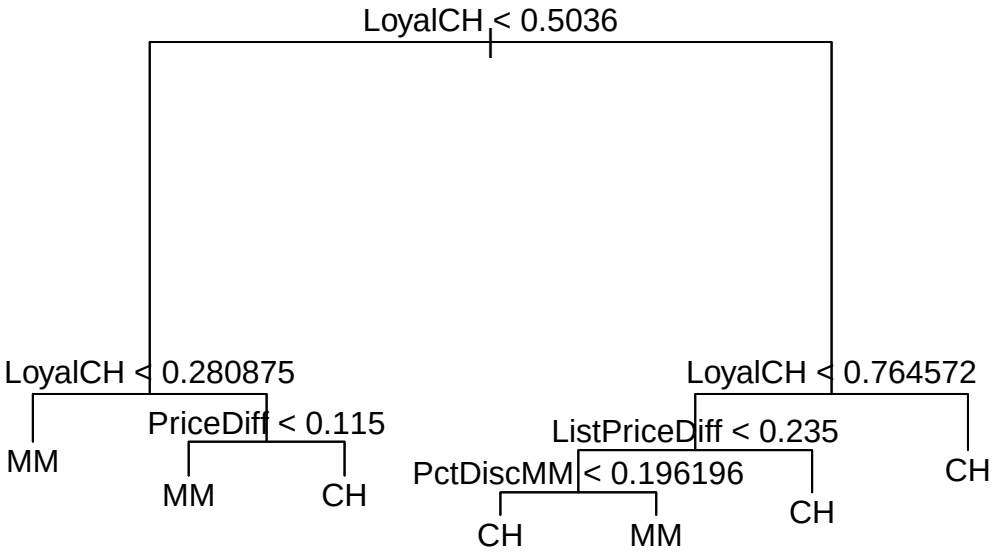
```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = datos[training, ])
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff" "PctDiscMM"
## Number of terminal nodes:  7
## Residual mean deviance:  0.7526 = 596.8 / 793
## Misclassification error rate: 0.155 = 124 / 800
```

En la salida de *summary*, vemos como se nos indica las variables que se han usado para realizar el árbol, ya que son las más relevantes para clasificar los datos del conjunto de test. Además de esto, vemos el número de nodos de los que dispone el árbol y el error resiudal que se preoduce, y el error dentro de los datos de training.

Como vemos, se produce un modelo de clasificación decente y simple de explicar.

3) Ahora, procedemos a pintar el árbol de clasificación que obtenemos:

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



En el resultado de la función, vemos cómo se desarrolla el árbol en dos ramas principales, en las que una dato d' se clasificará en una clase u otra, de acuerdo a la probabilidad que vemos en el árbol.

4) Ahora vamos a comprobar qué error de test tiene el modelo, usando la función *pred*.

```
pred = predict(tr, datos[-training,], type="class")
table(pred, Purchase[-training])
```

```
##
## pred  CH  MM
##   CH 150  34
##   MM  17  69
```

```
cat("Precision: ", mean(pred != Purchase[-training]) ,"\n")
```

```
## Precision:  0.1888889
```

```
cat("Etest = ", s$dev/s$df,"\n")
```

```
## Etest =  0.7525837
```

Al realizar la matriz de confusión, vemos que la imprecisión del modelo es todo aquello que no esté en la diagonal principal y la precisión del test, la podemos obtener en el summary del modelo. Para obtenerla, dividimos la desviación entre el número de muestras usadas.

5) *cvtree* realiza un proceso de validación cruzada en el árbol para estimar cuál es el mejor tamaño de este. Para ello, el la función recibe un modelo *tree*, un k para indicar el tamaño de validación cruzada y una función de poda en el árbol.

```
# K = 10 por defecto
cvTree = cv.tree(tr, FUN = prune.misclass)
data.frame(cvTree$size, cvTree$dev)
```

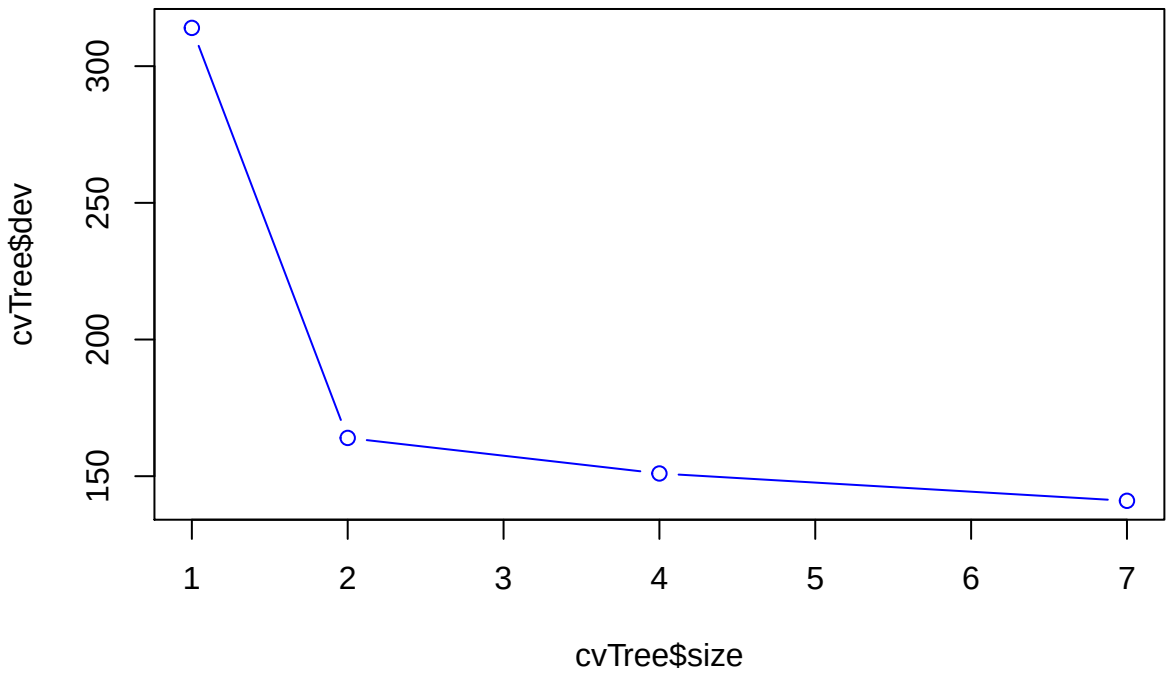
```
##   cvTree.size cvTree.dev
## 1           7        141
## 2           4        151
## 3           2        164
## 4           1        314
```

Como vemos, en la "tabla" que generamos con data.frame, vemos el tamaño y la desviación que tiene cada modelo. El mejor modelo es aquel que obtenga menos desviación, en este caso, el modelo que tiene el árbol

con tamaño 7.

6) Para generar el gráfico, solo tenemos hacer la siguiente llamada a *plot*.

```
plot(cvTree$size, cvTree$dev, type="b", col = "blue")
```



En la gráfica, podemos ver como efectivamente, el mejor modelo es el que tiene un tamaño de 7, ya que es el que tiene menor desviación.