Práctica 3 - Aprendizaje Automático

Marta Gómez Macías

12 de Mayo de 2016

Índice

1 Ejercicio 1 **1**

1.1 Representando los datos con `pairs` y `boxplot` . . . 2
1.2 Particionando los datos . . . 9
1.3 Clasificando los datos en función de su mediana . . . 9
1.4 Haciendo Validación Cruzada de 5 particiones sobre los modelos obtenidos . . . 12
1.5 Generando el mejor modelo de regresión posible . . . 13

2 Ejercicio 2 **16**

2.1 Seleccionando variables con Regresión Lasso con umbral 0,5 . . . 16
2.2 Ajustando un modelo de Regresión Ridge . . . 18
2.3 Clasificando los datos en función de su mediana con un modelo SVM . . . 20
2.4 Estimando el error de test y train por validación cruzada de 5 particiones . . . 21

3 Ejercicio 3 **22**

3.1 Dividiendo los datos en train y test . . . 22
3.2 Ajustando un modelo de regresión usando *bagging* . . . 22
3.3 Ajustando un modelo de regresión *Random Forest* . . . 23
3.4 Ajustando un modelo de regresión *Boosting* . . . 26

4 Ejercicio 4 **28**

4.1 Dividiendo los datos en train y test . . . 28
4.2 Ajustando un árbol a los datos de entrenamiento . . . 28
4.3 Representando el árbol . . . 28
4.4 Prediciendo los datos de test . . . 29
4.5 Determinando el tamaño óptimo del árbol . . . 30
4.6 Representando gráficamente el tamaño en función del error de CV . . . 30

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.**

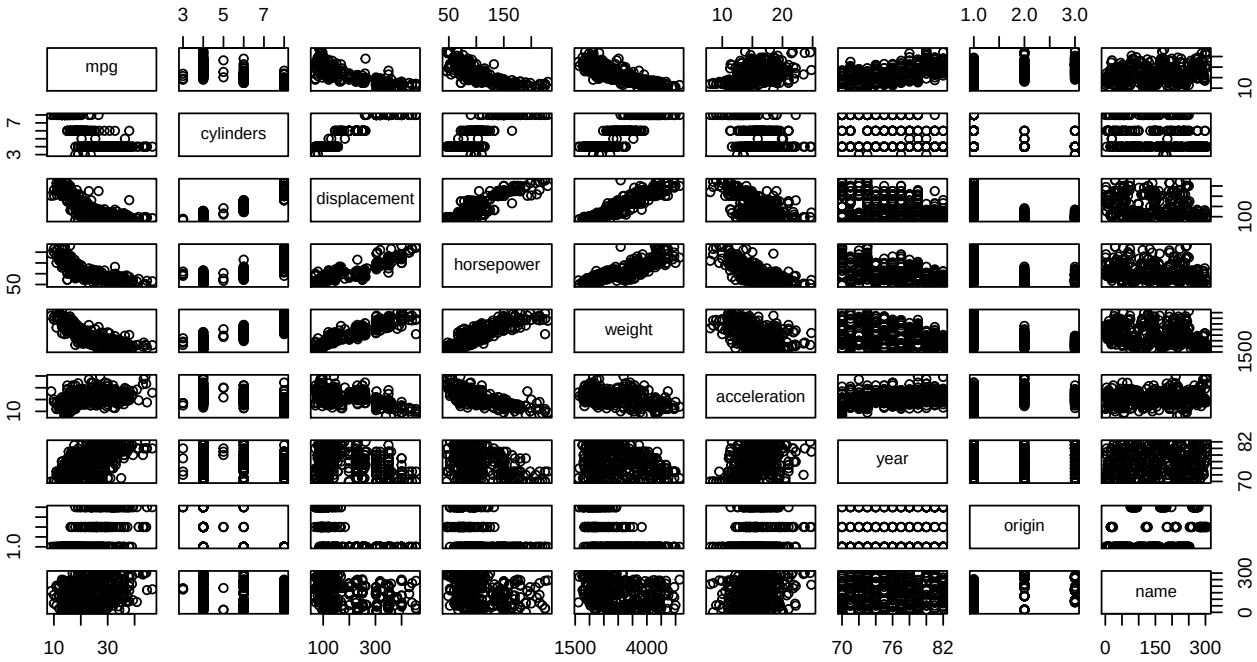
e) (Bonus)Estimar el error de test de ambos modelos (RL, K-NN) pero usando Validación Cruzada de 5-particiones. Comparar con los resultados obtenidos en el punto anterior.

f) (Bonus)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 Representando los datos con `pairs` y `boxplot`

La gráfica que obtenemos con `pairs` es:

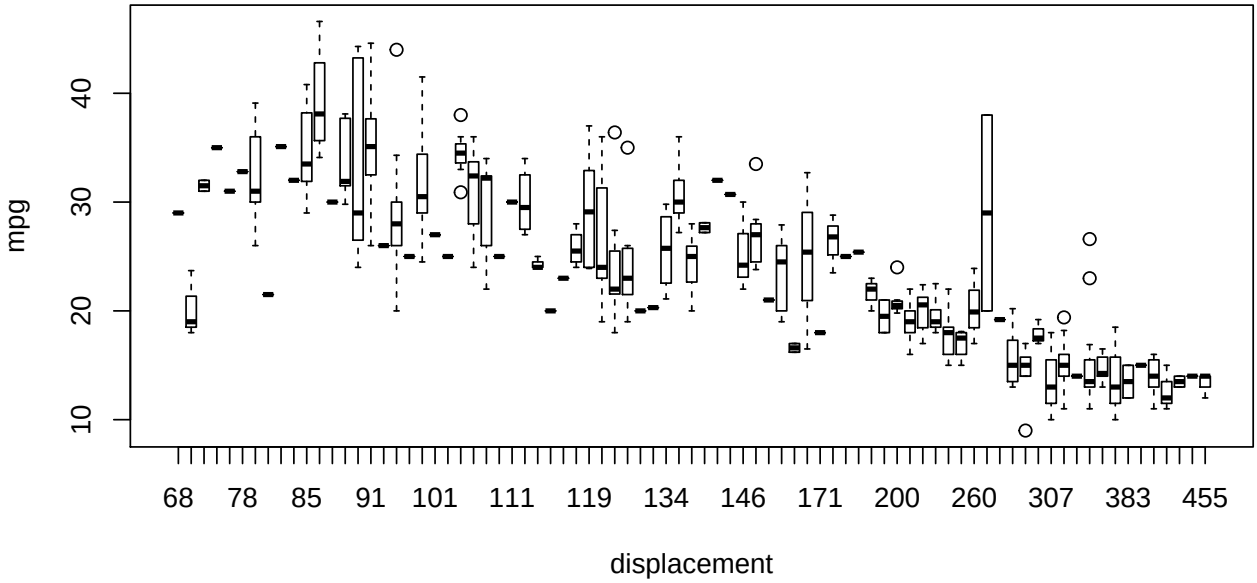
```
library("ISLR")
pairs(~ ., data=Auto)
```



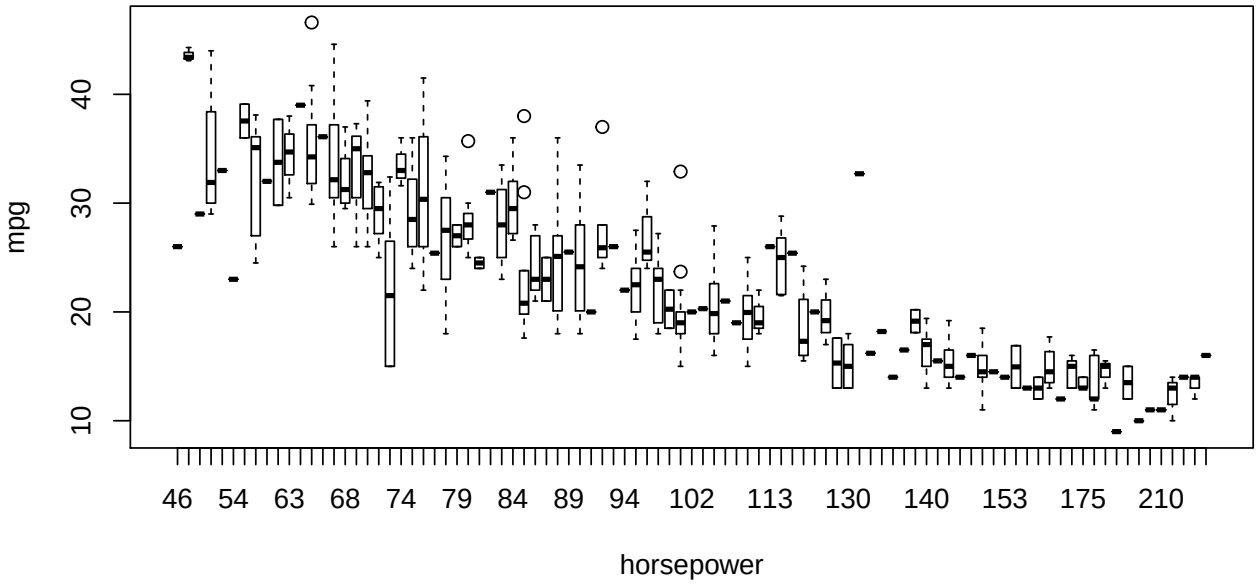
Según esta gráfica, las variables que más dependencia parecen tener con *mpg* son *displacement*, *horsepower* y *weight* porque son las únicas gráficas que parecen seguir la tendencia de una función. Otras variables, como *acceleration*, simplemente parecen tener demasiado ruido en la gráfica y por tanto, creo que no están tan relacionadas con *mpg* como las indicadas.

Representando *mpg* en función de cada una de las variables nombradas con `boxplot` obtenemos los siguientes resultados:

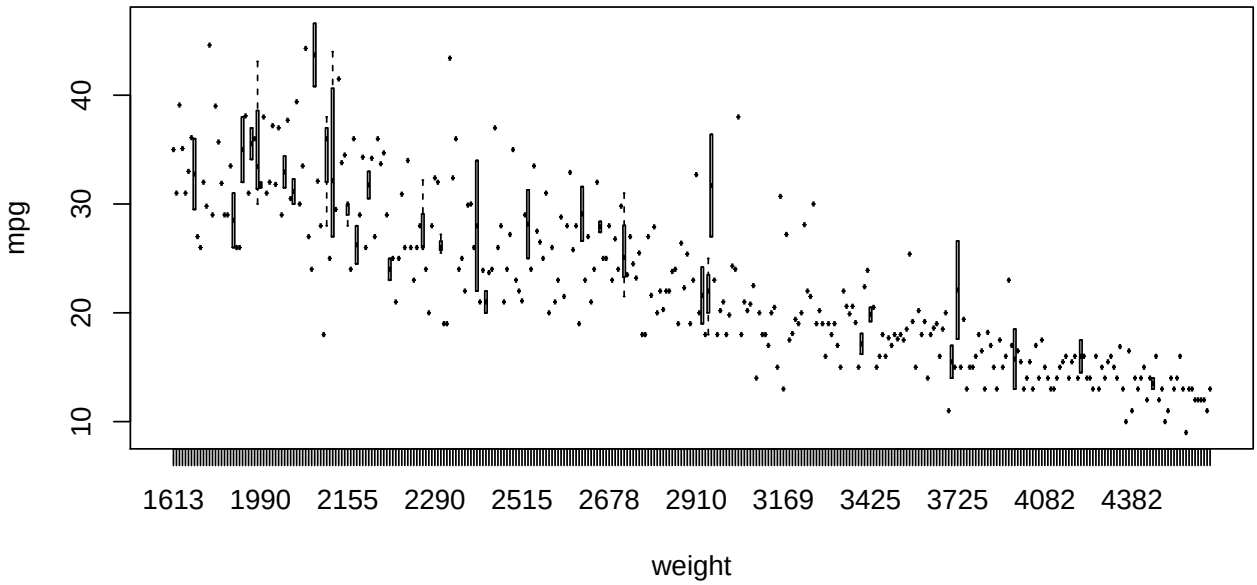
```
boxplot(mpg ~ displacement, data=Auto, ylab="mpg", xlab="displacement")
```



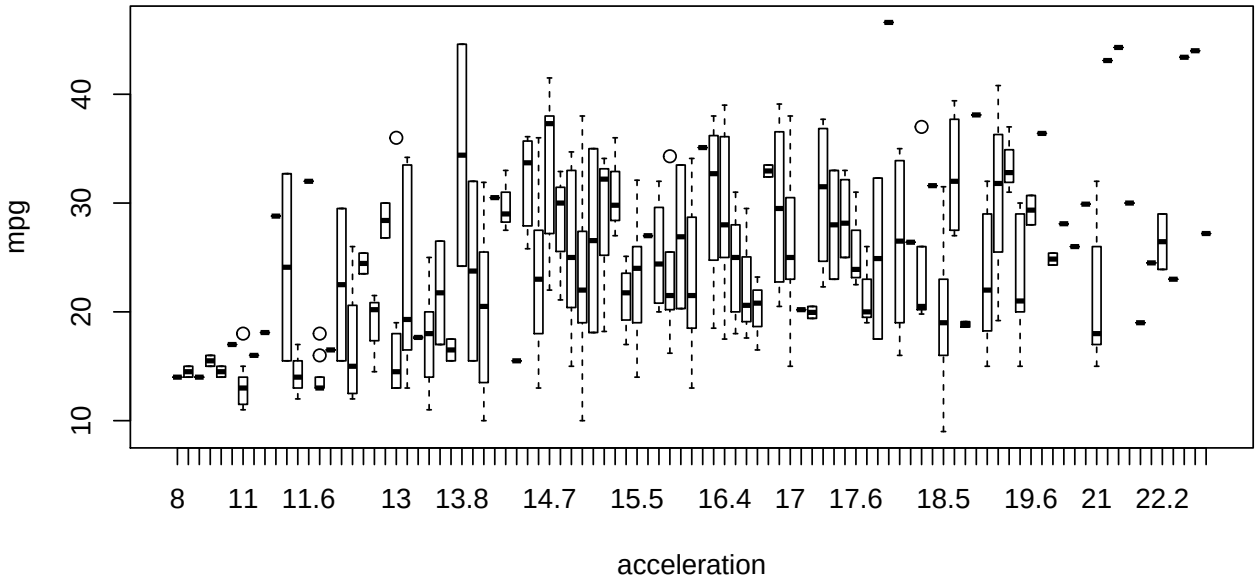

```
boxplot(mpg ~ horsepower, data=Auto, ylab="mpg", xlab="horsepower")
```




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




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



Tal y como hemos dicho antes, las variables *displacement*, *horsepower* y *weight* son las que más relación presentan con *mpg*. Con todas ellas obtenemos una gráfica parecida donde vemos que la función sigue una tendencia determinada pero con algo de varianza en los datos, que puede ser ruido. En cambio, con la variable *acceleration* obtenemos una gráfica en la que no queda muy claro cuál es la tendencia de los datos y además los datos presentan bastante varianza.

1.1.1 Generando modelos lineales de datos

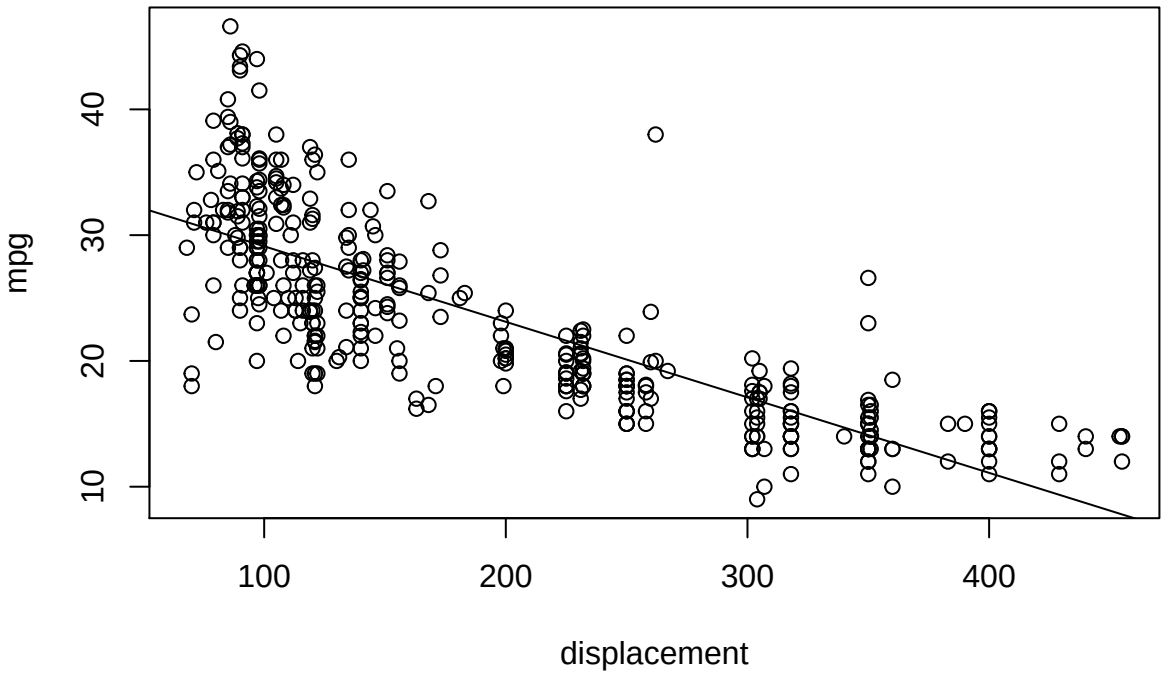
Para hacer una comprobación final de las variables seleccionadas, usamos `lm` para comparar los modelos generados por cada variable.

En el caso de la variable **displacement**, obtenemos el siguiente modelo:

```
attach(Auto)
model1 = lm(mpg ~ displacement)
summary(model1)
```

```
##
## Call:
## lm(formula = mpg ~ displacement)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -12.9170  -3.0243  -0.5021   2.3512  18.6128
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  35.12064    0.49443   71.03   <2e-16 ***
## displacement -0.06005    0.00224  -26.81   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.635 on 390 degrees of freedom
## Multiple R-squared:  0.6482, Adjusted R-squared:  0.6473
## F-statistic: 718.7 on 1 and 390 DF,  p-value: < 2.2e-16
```

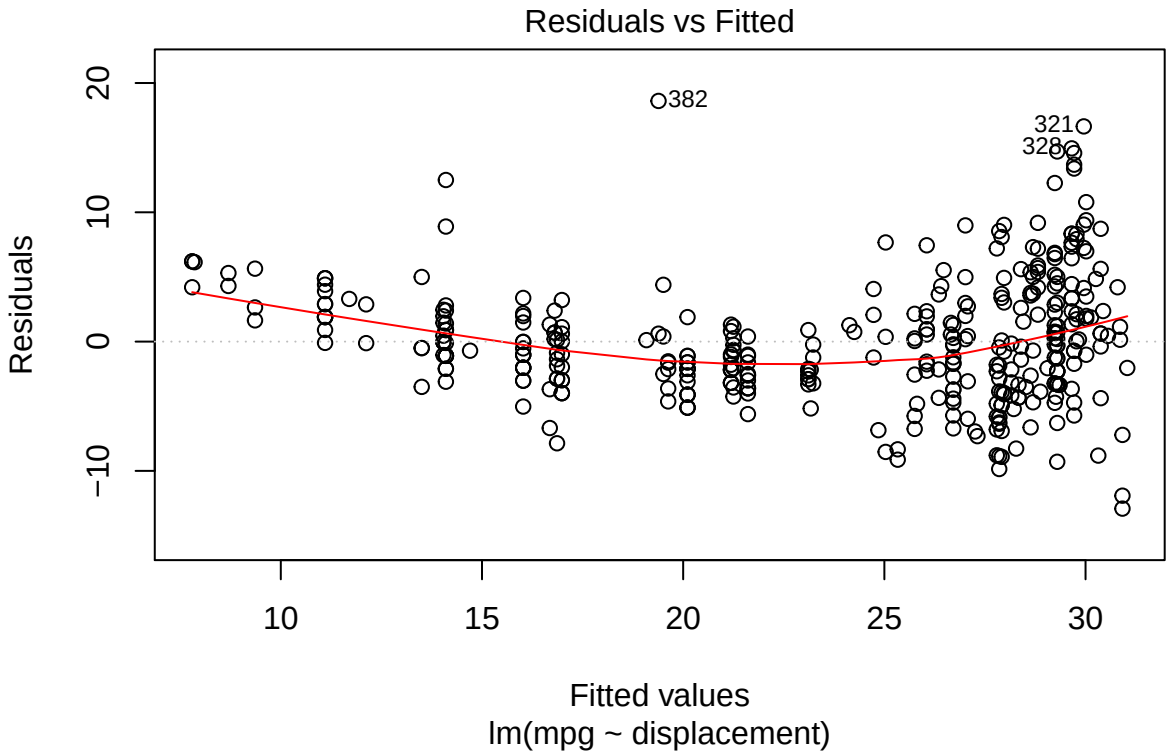
```
plot(x=displacement, y=mpg)
abline(model1$coefficients)
```



En principio, por la gráfica obtenida, vemos que la recta lineal que relaciona el *displacement* con el *mpg* se ha ajustado de manera aceptable, pero según los resultados estadísticos que obtenemos, el error del modelo es de $0,6473$ lo cual es bastante mejorable.

Si representamos el error del modelo, obtenemos la siguiente gráfica:

```
plot(model1, which=c(1))
```



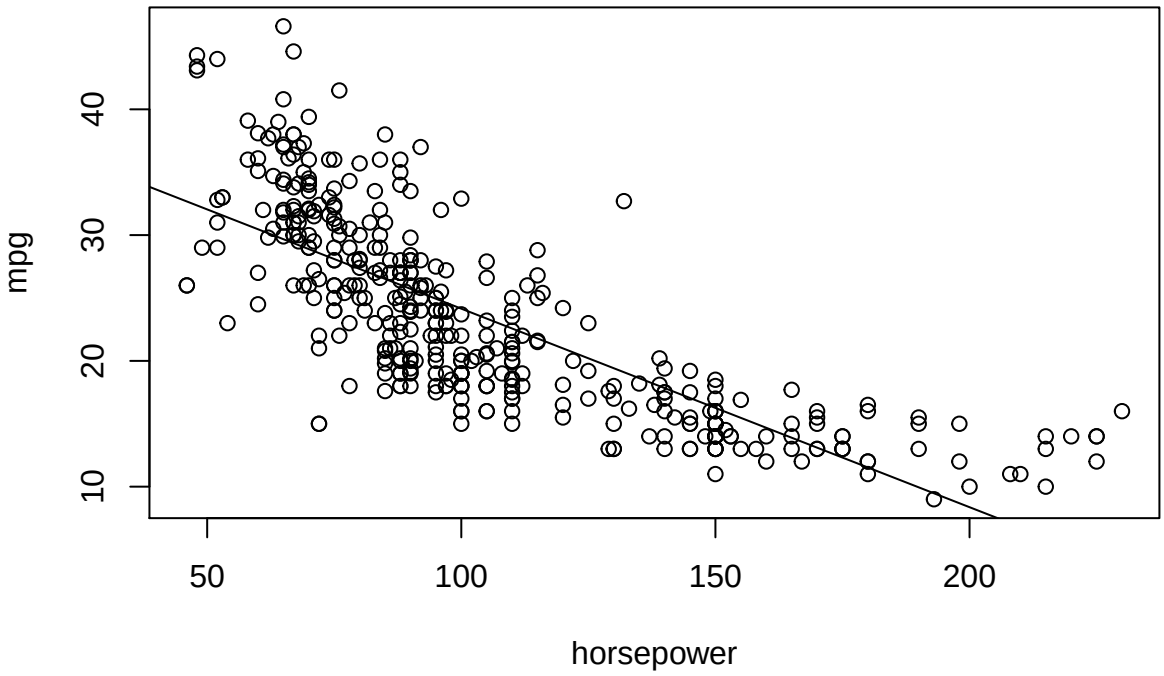
En esta gráfica vemos cómo el error del modelo no es lineal, sino una curva. Por tanto, nuestro modelo tampoco será lineal.

En el caso de la variable **horsepower**, obtenemos el siguiente modelo:

```
model2 = lm(mpg ~ horsepower)
summary(model2)
```

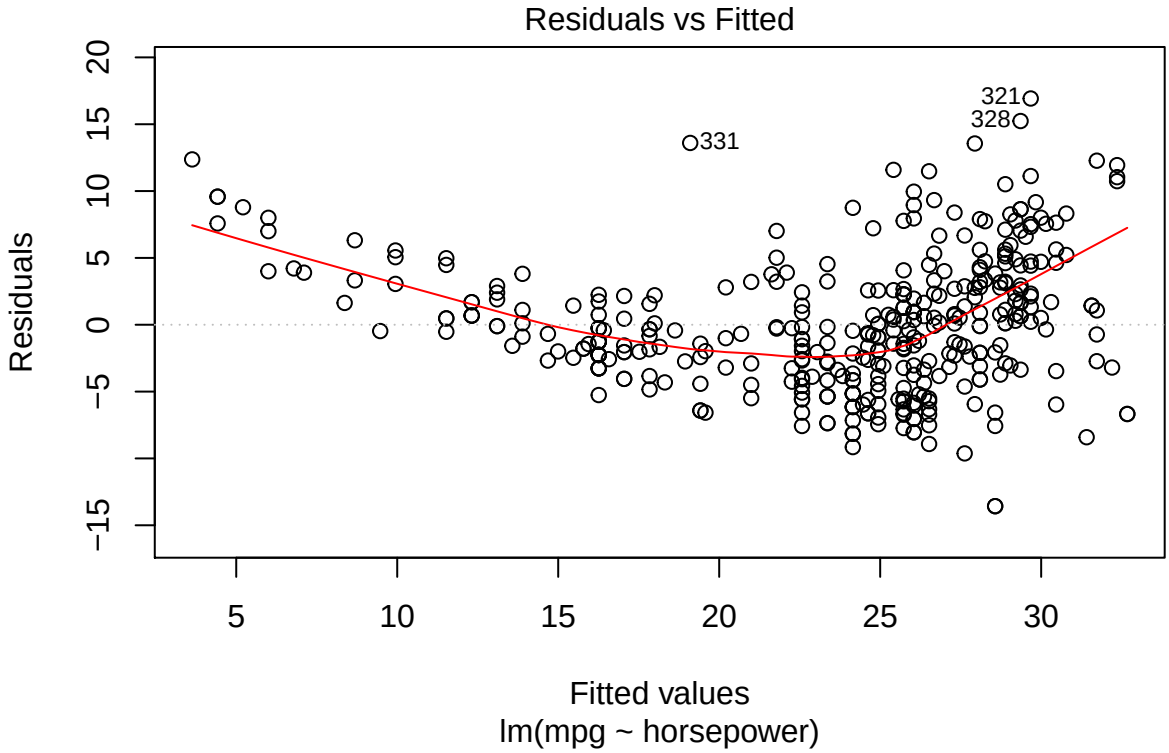
```
##
## Call:
## lm(formula = mpg ~ horsepower)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -13.5710  -3.2592  -0.3435   2.7630  16.9240
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 39.935861   0.717499   55.66   <2e-16 ***
## horsepower  -0.157845   0.006446  -24.49   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.906 on 390 degrees of freedom
## Multiple R-squared:  0.6059, Adjusted R-squared:  0.6049
## F-statistic: 599.7 on 1 and 390 DF,  p-value: < 2.2e-16
```

```
plot(x=horsepower, y=mpg)
abline(model2$coefficients)
```



El modelo obtenido es muy parecido al obtenido con la variable *displacement*, pero el error es algo más bajo: $0,6049$. Si representamos el error del modelo obtenemos la siguiente gráfica:

```
plot(model2, which=c(1))
```



Al igual que antes, vemos que el modelo obtenido no es lineal sino, como mínimo, cuadrático.

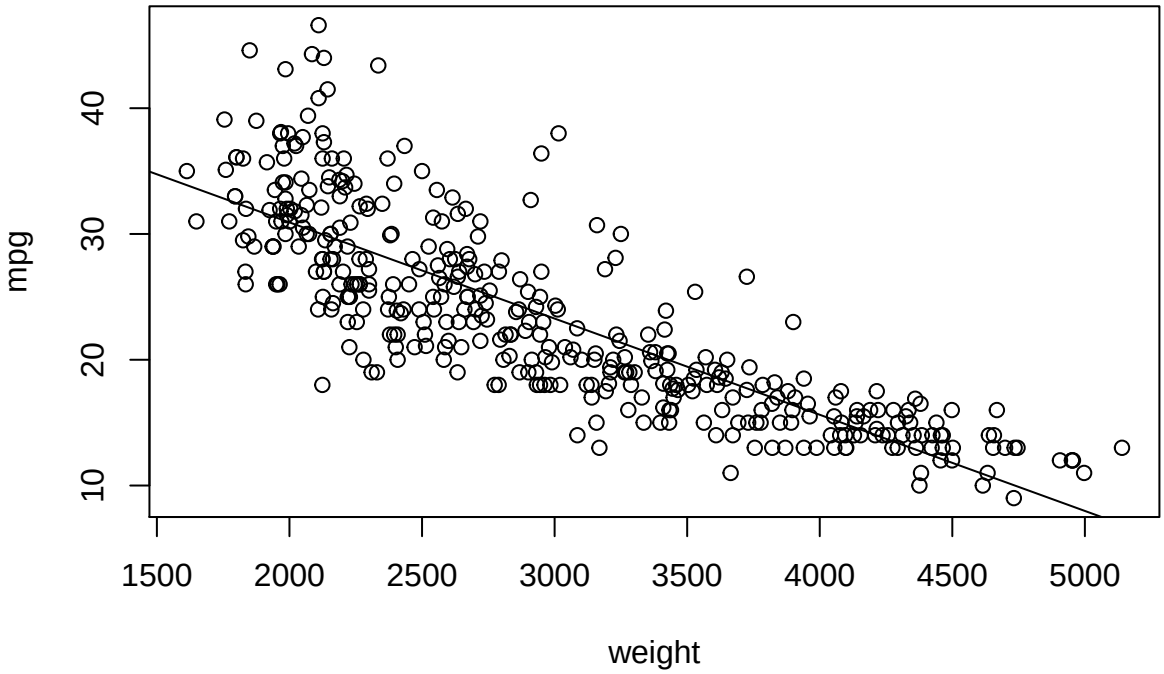
Por último, con la variable **weight** obtenemos el siguiente modelo:

```
model3 = lm(mpg ~ weight)
```

```
summary(model3)
```

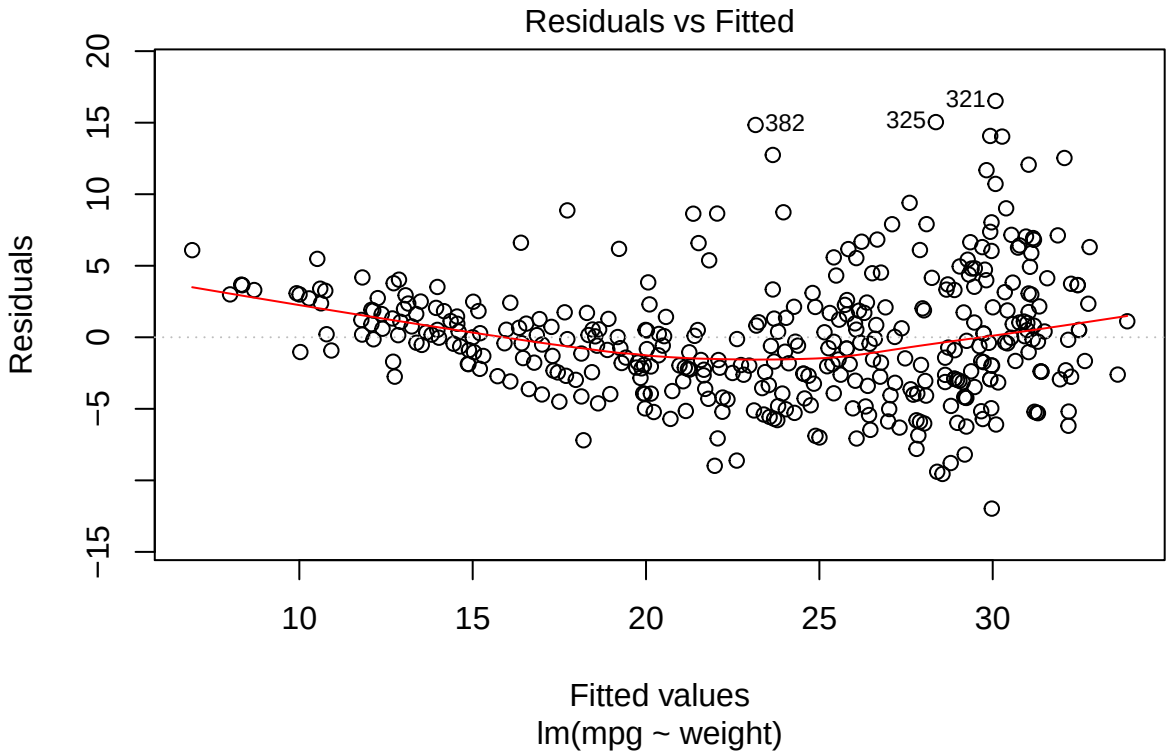
```
##
## Call:
## lm(formula = mpg ~ weight)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -11.9736  -2.7556  -0.3358   2.1379  16.5194
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 46.216524   0.798673   57.87   <2e-16 ***
## weight      -0.007647   0.000258  -29.64   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.333 on 390 degrees of freedom
## Multiple R-squared:  0.6926, Adjusted R-squared:  0.6918
## F-statistic: 878.8 on 1 and 390 DF,  p-value: < 2.2e-16
```

```
plot(x=weight, y=mpg)
abline(model3$coefficients)
```



El modelo obtenido es muy parecido a los dos anteriores, pero el error en este caso es el mayor de todos: $0,6918$. Si representamos el error del modelo obtenemos la siguiente gráfica:

```
plot(model3, which=c(1))
```



En este caso, al igual que los anteriores, vemos que el modelo no es lineal, ya que obtenemos una curva en el error del modelo.

Todas las variables estudiadas tienen un comportamiento parecido sobre *mpg* y por tanto, pienso que son las más indicadas para predecirlo.

1.2 Particionando los datos

Para particionar los datos he desarrollado la siguiente función:

```
divide_datos <- function(percent_train = 0.8, dataset=Auto) {
    train = sample(x=1:nrow(dataset), size=nrow(dataset)*percent_train)
    test = c(1:nrow(dataset))[-train]
    l = list(train,test)
    names(l) = c("train", "test")
    l
}

division = divide_datos() # dividimos los datos en test y training con 20 y 80% de tamaño
```

En ella, hago en primer lugar un *shuffle* de los índices posibles, para que el orden no afecte a la asignación de test y train. Uso sólo los índices para poder especificar a la hora de calcular modelos el subconjunto de todos los datos totales a través de estos índices.

1.3 Clasificando los datos en función de su mediana

Para clasificar los datos con la variable *mpg*01 he hecho la siguiente función:

```
etiquetas <- function(datos=Auto$mpg) {
    mediana = median(datos) # calculamos la mediana de todos los valores
    mpg01 = (datos >= mediana)*1 # multiplicamos por 1 para que sea numérico
```

```
    mpg01
}
```

En la cual, calculamos la mediana, etiquetamos cada coche según el criterio indicado y devolvemos el vector de etiquetas obtenido.

1.3.1 Aplicando regresión logística a los datos de entrenamiento.

Para aplicar *regresión logística*, he hecho la siguiente función:

```
Auto = data.frame(Auto, as.factor(etiquetas())) # añadimos la columna de etiquetas
names(Auto)[ncol(Auto)] = "mpg01"
Auto$name = NULL # eliminamos la columna de nombres pues sólo queremos valores numéricos

convert_classification <- function(pred, useprob=TRUE) {
    if (useprob) {
        clasificacion = ifelse(pred > 0.5, 1, 0) # lo convertimos en clasificación
    } else {
        clasificacion = pred
    }
    print(table(clasificacion,Auto$mpg01[division$test])) # generamos la matriz de confusión
    cat("Error = ",clasification_error(clasificacion, Auto$mpg01[division$test]),"\n")
}

logistic_regression_Auto <- function() {
    model = glm(formula = mpg01 ~ (displacement + horsepower + weight), data=Auto,
                subset=division$train, family="binomial")
    pred = predict(object=model, newdata=Auto[division$test,],type="response") # lo testeamos
    list(pred, model)
}
```

Para calcular el error del modelo obtenido, he desarrollado la siguiente función:

```
clasification_error <- function(modelo, etiquetas) {
    mean((modelo != etiquetas)*1) # el error será la media del número de puntos mal clasificado
}
```

Por tanto, el error de test obtenido es:

```
convert_classification(logistic_regression_Auto()[[1]])
```

```
##
## clasificacion  0  1
##             0 33  4
##             1  6 36
## Error =  0.1265823
```

Como se ve en la matriz de confusión, tenemos cuatro ejemplos mal clasificados en cada clase. Ésto nos da un error del 12 %, lo cual es mejorable.

1.3.2 Aplicando KNN a los datos de entrenamiento

Para aplicar *KNN* he desarrollado la siguiente función:

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

normalizar_01 <- function(datos) {
    apply(X=datos, MARGIN=2, FUN=function(x) {
```

```
        max <- max(x)
        min <- min(x)
        sapply(X=x, FUN=function(xi) (xi-min)/(max-min))
    })
}

vars_seleccionadas = data.frame(Auto$displacement, Auto$horsepower, Auto$weight)
names(vars_seleccionadas) = c("displacement", "horsepower", "weight")
vs_train = data.frame(normalizar_01(vars_seleccionadas[division$train,]),Auto$mpg01[division$train])
names(vs_train)[ncol(vs_train)] = "mpg01"
vs_test = data.frame(normalizar_01(vars_seleccionadas[division$test,]), Auto$mpg01[division$test])
names(vs_test)[ncol(vs_test)] = "mpg01"

knn_Auto <- function(kfit=1, useprob=F) {
    knn(train=vs_train[,-4], test=vs_test[,-4], cl=vs_train[,4], k=kfit, prob=useprob)
}

get_best_k <- function() {
    clasificacion = tune.knn(x=subset(vs_train, select=-mpg01), y=vs_train$mpg01, k=1:5)
    clasificacion$best.parameters$k
}

print("knn con k=1")
```

```
## [1] "knn con k=1"
```

```
convert_classification(knn_Auto(), useprob=F)
```

```
##
## clasificacion  0  1
##             0 34  7
##             1  5 33
## Error =  0.1518987
```

```
print("knn con el mejor knn calculado por tune.knn")
```

```
## [1] "knn con el mejor knn calculado por tune.knn"
```

```
mejork = get_best_k()
cat("mejor k = ",mejork,"\n")
```

```
## mejor k =  4
```

```
convert_classification(knn_Auto(kfit=mejork), useprob=F)
```

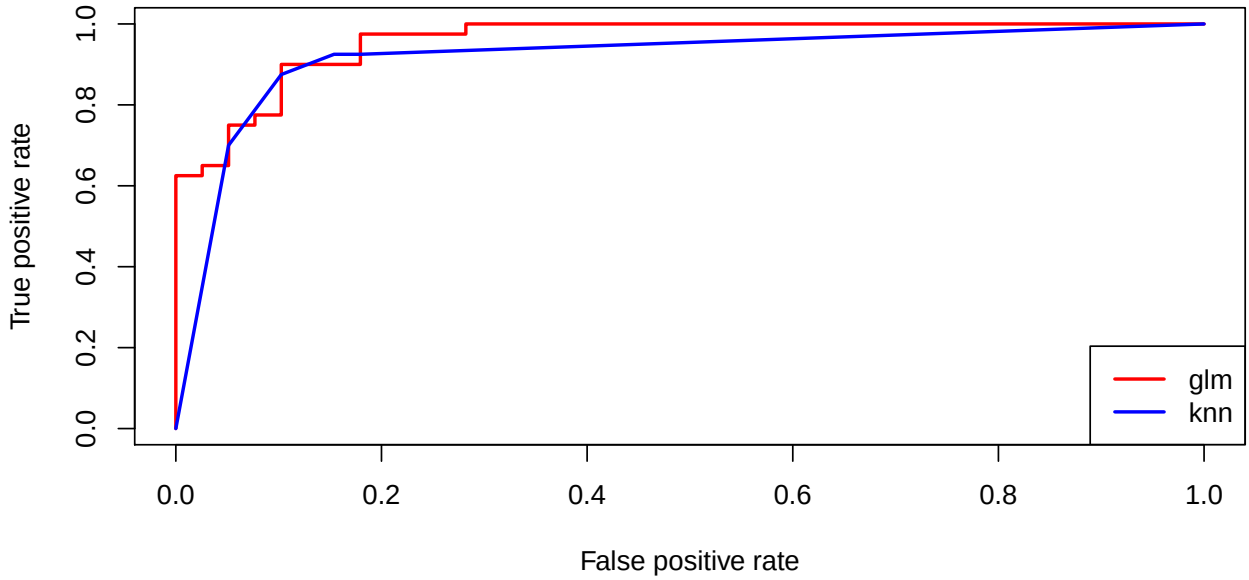
```
##
## clasificacion  0  1
##             0 34  3
##             1  5 37
## Error =  0.1012658
```

En primer lugar, con $k = 1$ obtenemos un error del 15 %. Con `tune.knn` obtenemos $k = 4$, que nos da un error de un 10 %. Un error que es bajo, pero mejorable. Este error es algo menor que el obtenido en el modelo `glm`.

1.3.3 Representando la curva ROC de los modelos obtenidos

```
library(ROCR)
```

```
rocplot = function(pred, truth=Auto$mpg01[division$test], ...) {
    predob = prediction(pred, truth)
    perf = performance(predob, "tpr", "fpr")
    plot(perf,...)
}
rocplot(pred=logistic_regression_Auto()[[1]],col="red", lwd=2)
pred_knn = knn_Auto(kfit=mejork, useprob=T)
prob_knn = attr(pred_knn, "prob")
# debemos modificar los resultados para que puedan ser interpretados de igual forma que el glm
prob_knn = ifelse(pred_knn == 0, 1 - prob_knn, prob_knn)
rocplot(pred=prob_knn, add=T, lwd=2, col="blue")
legend('bottomright', c("glm","knn"), col=c('red', 'blue'), lwd=2)
```



Ambas curvas salen aproximadamente iguales, lo cual es normal porque al analizar los modelos obtenidos por separado, vimos que obteníamos errores muy parecidos con ambos, de un 12 y 10 % respectivamente.

1.4 Haciendo Validación Cruzada de 5 particiones sobre los modelos obtenidos

Para obtener el error de validación cruzada del modelo `KNN` podemos usar la función `tune.knn` para obtener el mejor error usando validación cruzada con particiones:

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

```
## [1] 0.07997952
```

Para calcular la validación cruzada para el modelo obtenido con `glm` podemos usar la función `cv.glm` del paquete `boot` de R. El vector δ nos devuelve dos errores, el primero es el error de validación cruzada sin ningún ajuste posterior:

```
library(boot)
cv.glm(data=Auto[division$train,], glmfit = logistic_regression_Auto()[[2]], K=5)$delta[1]
```

```
## [1] 0.08164864
```

Con KNN, obtenemos un error del 7 % mientras que con *Regresión Logística*, lo obtenemos del 8 %. Al igual que antes, obtenemos que KNN nos da un menor error que *Regresión Logística*, aunque en este caso los errores obtenidos son menores en ambos casos.

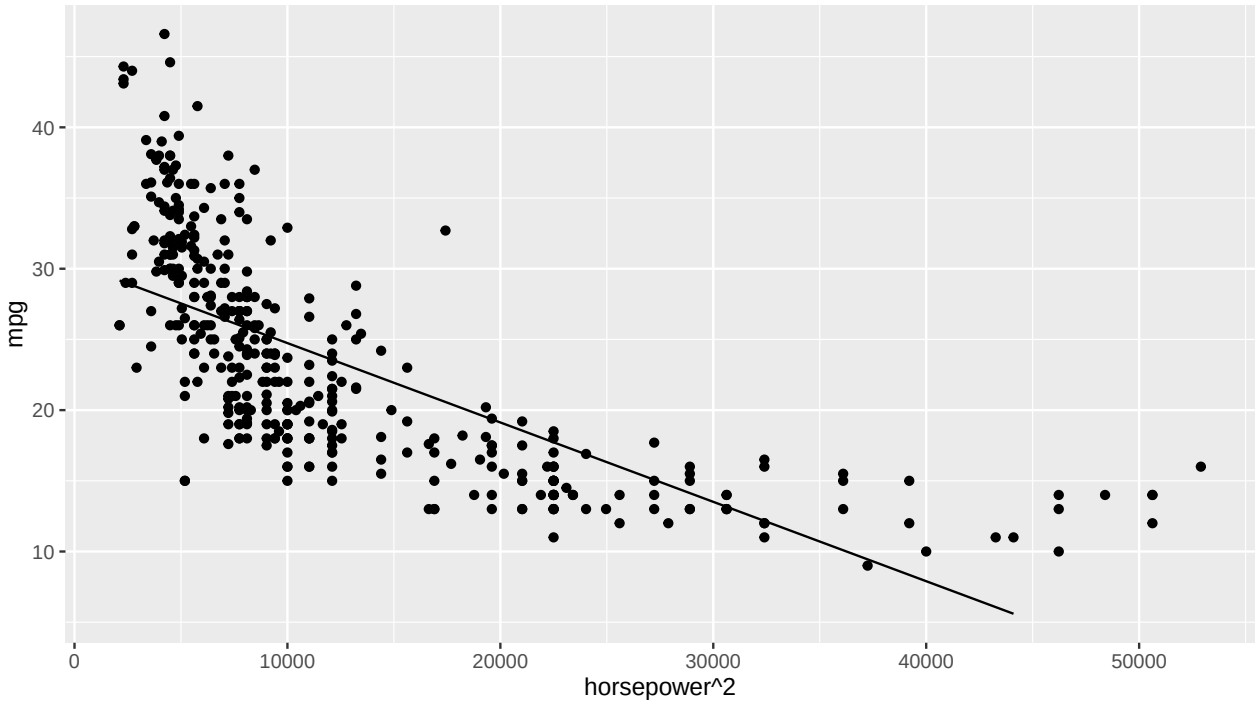
1.5 Generando el mejor modelo de regresión posible

Como se dijo anteriormente, los datos no siguen un modelo lineal, por lo que en este apartado vamos a probar a hacer un modelo cuadrático, usando la variable que mejor resultado ha dado en el modelo lineal: *horsepower*.

```
library(ggplot2)

attach(Auto)
modelf1 = lm(mpg ~ I(horsepower^2), subset=division$train, data=Auto)
summary(modelf1)
```

```
##
## Call:
## lm(formula = mpg ~ I(horsepower^2), data = Auto, subset = division$train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -12.444  -3.938  -1.071   3.242  18.618
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)      3.035e+01  5.040e-01   60.23   <2e-16 ***
## I(horsepower^2) -5.614e-04  3.184e-05  -17.63   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.571 on 311 degrees of freedom
## Multiple R-squared:  0.4999, Adjusted R-squared:  0.4983
## F-statistic: 310.9 on 1 and 311 DF,  p-value: < 2.2e-16
```

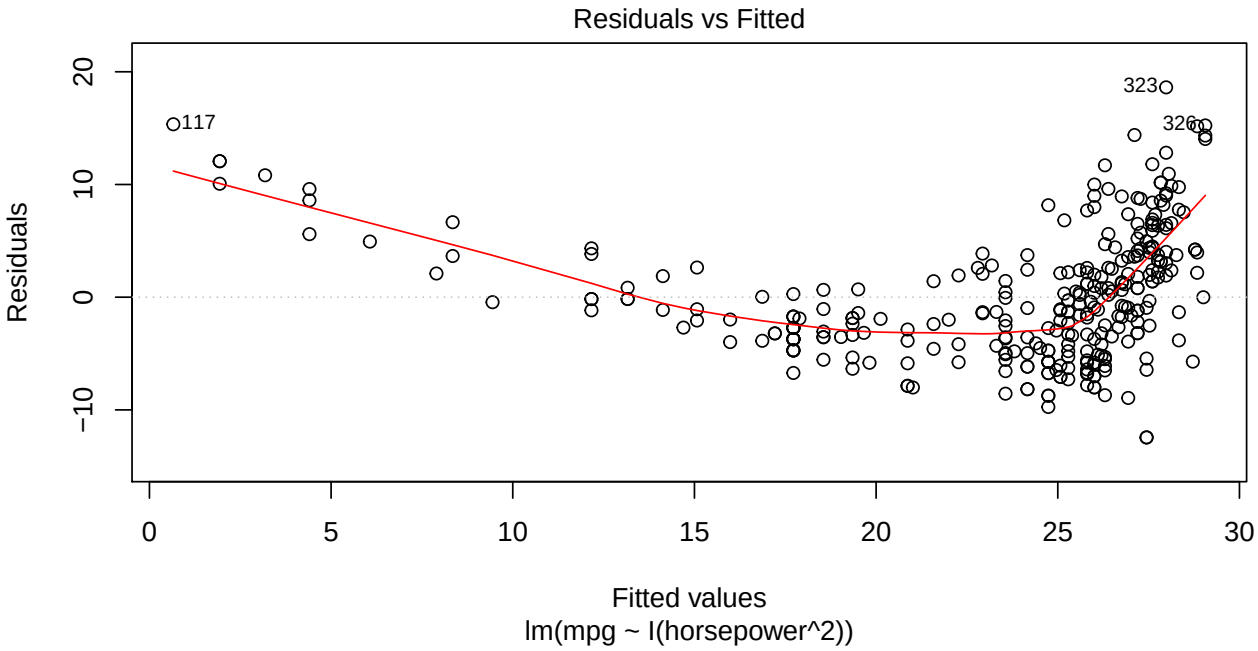
```
(ggplot() + geom_point(data=Auto, aes(x=horsepower^2, y=mpg))
    + geom_line(aes(x=Auto[division$test,]$horsepower^2, y=predict(modelf1, Auto[division$test,]))))
```



Hemos conseguido bajar el error con respecto al modelo lineal obtenido anteriormente, de $0,6049$ a $0,4983$.

Pero la función obtenida no sigue correctamente la tendencia de los puntos.

```
plot(modelf1, which=c(1))
```



Vemos que la gráfica sigue la tendencia del error del modelo, pero no de forma exacta. Aún tenemos que refinar nuestro modelo algo más.

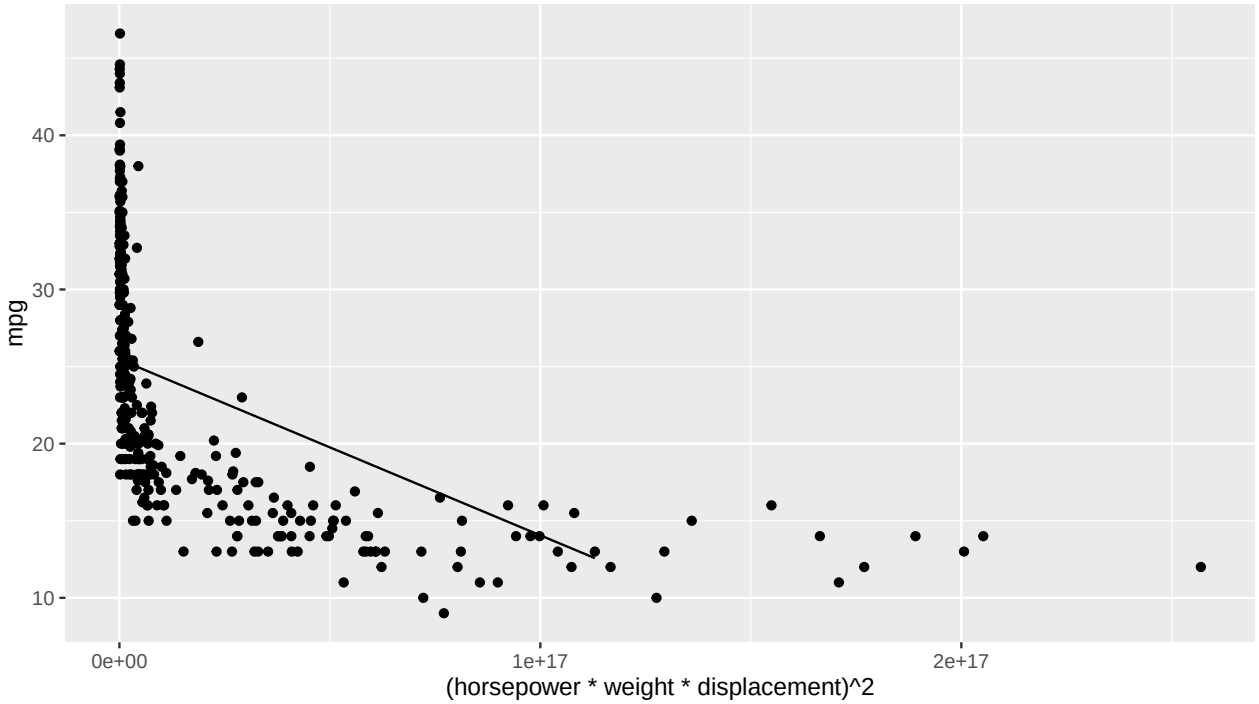
Vamos a probar a hacer una mezcla cuadrática de todas las variables seleccionadas:

```
modelf2 = lm(mpg ~ I((horsepower * weight * displacement)^2), data=Auto, subset=division$train)
summary(modelf2)
```

```
##
## Call:
## lm(formula = mpg ~ I((horsepower * weight * displacement)^2),
##     data = Auto, subset = division$train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.726  -5.427  -1.208   4.576  21.150
##
## Coefficients:
##                                          Estimate Std. Error t value
## (Intercept)                             2.547e+01  4.124e-01   61.74
## I((horsepower * weight * displacement)^2) -1.141e-16  9.871e-18  -11.56
##                                         Pr(>|t|)
## (Intercept)                               <2e-16 ***
## I((horsepower * weight * displacement)^2)   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.588 on 311 degrees of freedom
## Multiple R-squared:  0.3006, Adjusted R-squared:  0.2984
## F-statistic: 133.7 on 1 and 311 DF,  p-value: < 2.2e-16
```

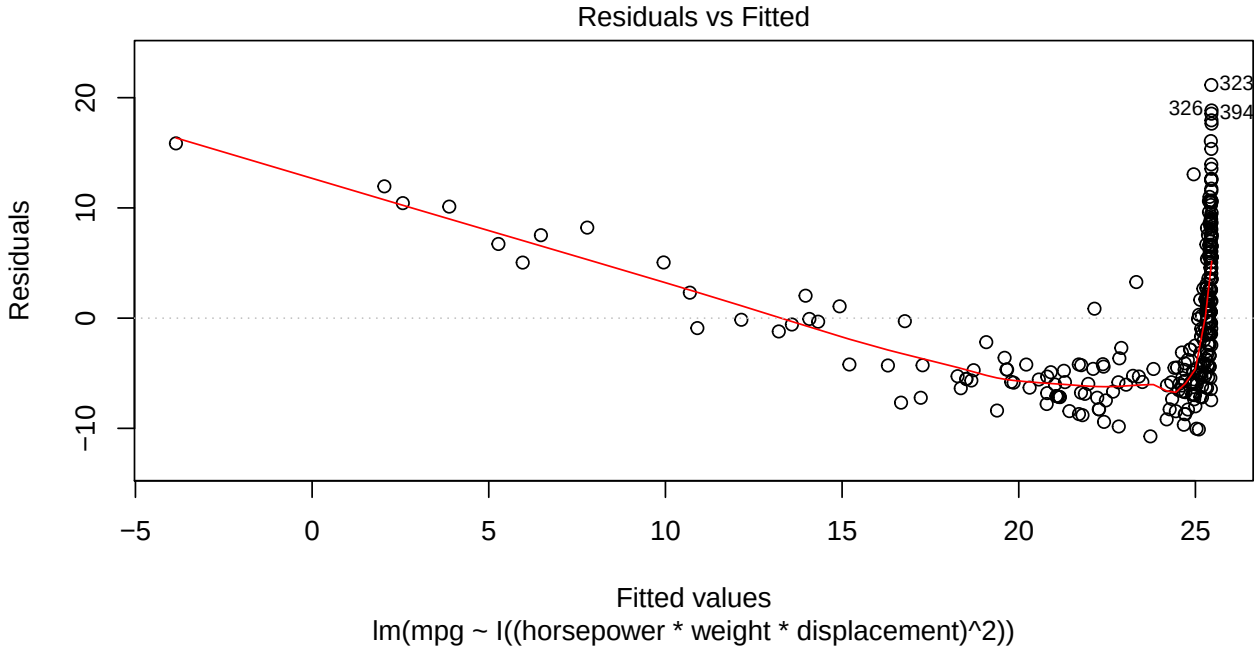
```
(ggplot() + geom_point(data=Auto, aes(x=(horsepower * weight * displacement)^2, y=mpg))
```

```
  + geom_line(aes(x=(Auto[division$test,]$horsepower
      * Auto[division$test,]$weight * Auto[division$test,]$displacement)^2,
      y=predict(modelf2, Auto[division$test,])))
```



En este caso, obtenemos un error de $0,2984$. Frente al error obtenido en el modelo anterior de $0,4983$ considero que este modelo es bastante aceptable. Los datos no se ajustan de manera perfecta a la gráfica, pero a pesar de eso considero que he obtenido un buen modelo.

```
plot(modelf2, which=c(1))
```



El error de este modelo se ajusta de forma prácticamente perfecta a la gráfica, por lo que podemos decir que el error de nuestro modelo sigue la misma tendencia que el error de los datos.

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 éste 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 *regressión-LASSO* (usar paquete `glmnet` de `R`) donde seleccionamos sólo 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 algún núcleo. Valorar los resultados del uso de distintos núcleos.**

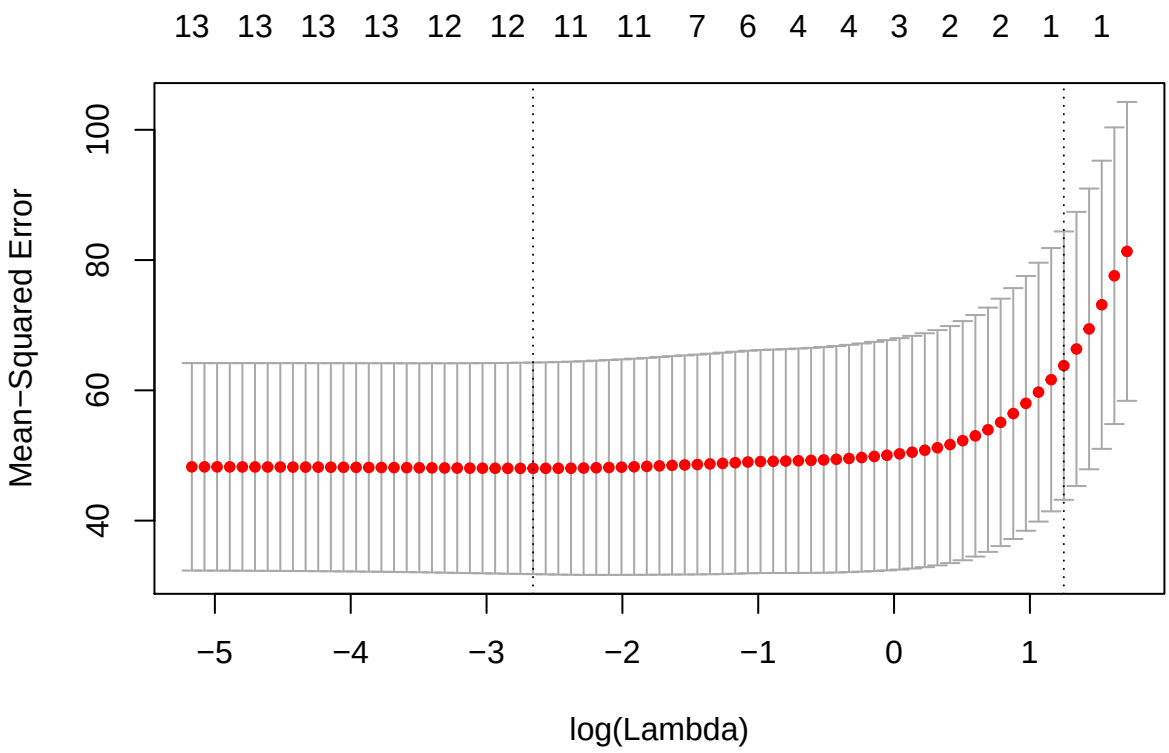
d) **(Bonus)Estimar el error de entrenamiento y test por validación cruzada de 5 particiones.**

2.1 Seleccionando variables con Regresión Lasso con umbral 0,5

Para obtener un modelo de regresión lasso, usamos la función `glmnet` con el parámetro $\alpha = 1$, ya que si usáramos $\alpha = 0$, obtendríamos un modelo de regresión Ridge.

Para poder estimar el error de test del modelo obtenido y el mejor λ, partimos los datos en un conjunto de test y en otro de entrenamiento para poder hacer una validación cruzada.

```
library(MASS)
library(glmnet)

attach(Boston)

divB = divide_datos(dataset=Boston)
x = as.matrix(subset(Boston, select=-crim))
y = Boston$crim

regresion_glmnet <- function(a=1) {
    cv.out = cv.glmnet(x=x[divB$train,], y = y[divB$train], alpha=a)
    plot(cv.out)
    bestlam = cv.out$lambda.min
    cat("Lambda con menor error de validación cruzada:",bestlam,"\n")
    lasso.pred = predict(cv.out, s=bestlam, newx=x[divB$test,])
    MSE = mean((lasso.pred - y[divB$test])^2)
    cat("Mean Squared Error = ", MSE,"\n")
    bestlam
}

bestlam <- regresion_glmnet()
```

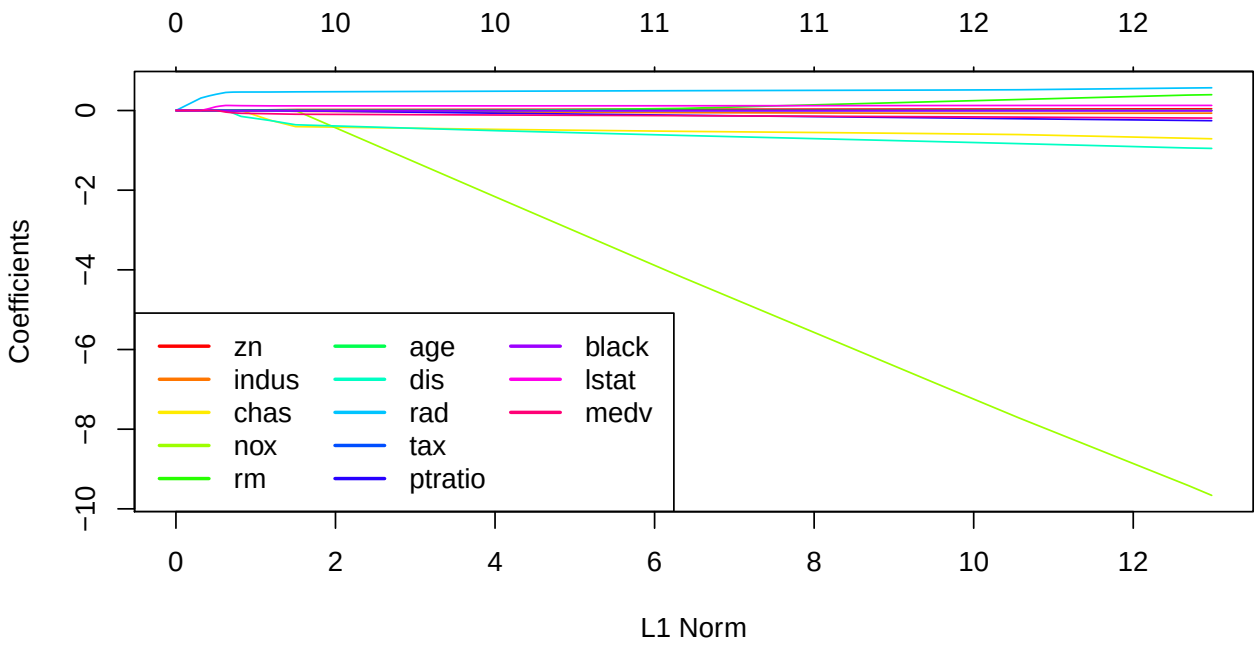



```
## Lambda con menor error de validación cruzada: 0.07011092
## Mean Squared Error =  23.26687
```

Como se ve en la gráfica obtenida, el error obtenido sigue la tendencia del error del modelo, pero éste tiene bastante desviación.

Una vez obtenido el mejor lambda, pasamos a calcular el modelo. Para calcular el modelo probamos valores para λ desde $\lambda = 10^{10}$ hasta $\lambda = 10^{-2}$ para poder cubrir todos los posibles escenarios.

```
grid = 10^seq(10, -2, length=100)
out = glmnet(x=x, y=y, alpha=1, lambda=grid)
plot(out, col=rainbow(n=length(rownames(out$beta))))
legend('bottomleft', rownames(out$beta),
  col=rainbow(n=length(rownames(out$beta))), lwd=2, ncol = 3)
```




```
lasso.coef = predict(out, type="coefficients", s=bestlam)[1:ncol(x),]
lasso.coef[abs(lasso.coef) > 0.5]
```

```
## (Intercept)        chas         nox         dis         rad
##  11.4071032  -0.5556811  -5.7502619  -0.7167740   0.5067982
```

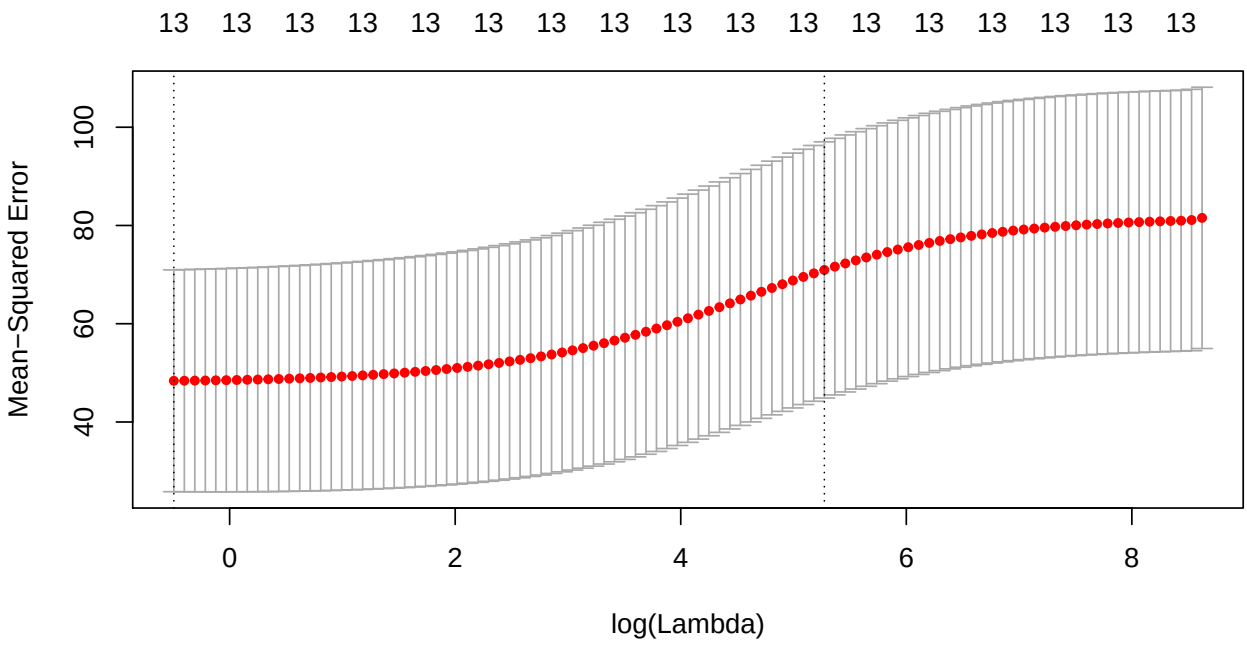
Al principio teníamos 13 variables predictoras (`zn`, `indus`, `chas`, `nox`, `rm`, `age`, `dis`, `rad`, `tax`, `ptratio`, `black`, `lstat` y `medv`), pero tras obtener el modelo de regresión Lasso, hemos pasado a tener 4: `chas`, `nox`, `dis` y `rad`

Como se ve en la gráfica, el valor que entra en primer lugar en el modelo es `rad`, que justo al principio se eleva del 0. Después, lo hacen algunas variables más, pero debido a que tienen un peso muy bajo no han sido seleccionadas. La última en entrar, pero que más peso tiene, es `nox` y en la gráfica se ve como se separa del resto de las variables. `dis` y `chas` presentan una tendencia muy parecida, aunque al final `dis` acaba teniendo un mayor peso.

2.2 Ajustando un modelo de Regresión Ridge

Como dije antes, para poder ajustar un modelo de regresión ridge, debemos usar la función `glmnet` con el parámetro $\alpha = 0$. Al igual que antes, usaremos validación cruzada para obtener el valor de λ.

```
bestlam_ridge <- regresion_glmnet(a=0)
```




```
## Lambda con menor error de validación cruzada: 0.6097891
## Mean Squared Error =  23.50477
```

Al igual que antes, la desviación del error del modelo es muy grande, pero el error obtenido sigue la tendencia del modelo.

Ahora vamos a pasar a calcular el error residual del modelo, para ello, vamos a obtener el modelo usando los datos de entrenamiento para poder luego predecir usando los datos de test.

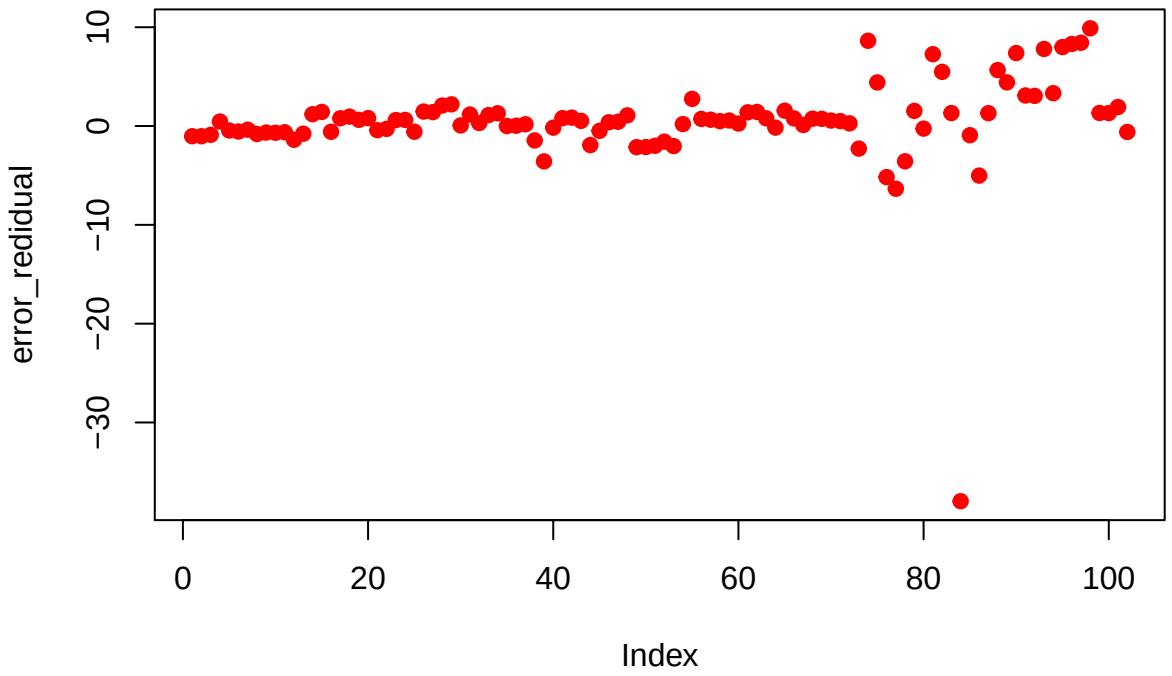
Para calcular el *error residual* (no cuadrático), restamos el valor predecido, menos el real: $y_{fit} - y$.

```
x = cbind(chas, nox, dis, rad)
out = glmnet(x=x[divB$train,], y=y[divB$train], alpha=0, lambda=grid)

get_residual_error <- function(pred, real) {
    pred - real
}

ridge.pred <- predict(out, s=bestlam, newx=x[divB$test,])

error_redidual <- get_residual_error(ridge.pred, crim[divB$test])

mean(error_redidual)
```

```
## [1] 0.4922679
```

En este caso, vemos que, en media, el error del modelo es $0,5$. Éste error es bastante aceptable, aunque si hacemos una gráfica con los valores de error obtenidos vemos que muchos se salen de esta tendencia:

```
plot(error_redidual, pch=19, col="red")
```



Como se ve en la gráfica, los datos están "subajustados" debido que los errores obtenidos en algunos de ellos es demasiado grande, muestra de que el modelo se aleja bastante de la tendencia que los datos siguen. Esto se aprecia sobre todo en los últimos datos de la gráfica.

2.3 Clasificando los datos en función de su mediana con un modelo SVM

Para clasificar los datos, he usado la misma función que he desarrollado antes:

```
crim01 <- etiquetas(datos=Boston$crim)

Boston = data.frame(Boston, as.factor(crim01))
names(Boston)[ncol(Boston)] = "crim01"
```

Una vez clasificados, pasamos a ajustar el modelo con SVM. En primer lugar lo hacemos con un modelo lineal. Para ello, usamos el parámetro `kernel="linear"` en la función `svm`. Para decidir el valor que asignamos al parámetro `cost` usamos la función `tune`

```
support_vector_machine <- function(k="linear") {
    tune_svm = tune(svm, crim01 ~ ., data=Boston, kernel=k,
                    ranges=list(cost=c(0.001,0.01,0.1,1,5)))
    bestmod = tune_svm$best.model
    ypred = predict(bestmod, Boston[divB$test,])
    print(table(predict=ypred, truth=Boston$crim01[divB$test]))
    cat("Tamaño total del conjunto de test:",length(divB$test),"\n")
    cat("Error por validación cruzada de 10 particiones del modelo:", tune_svm$best.performance)
}
support_vector_machine()
```

```
##         truth
## predict  0  1
##       0 46  6
##       1  2 48
## Tamaño total del conjunto de test: 102
## Error por validación cruzada de 10 particiones del modelo: 0.07501961
```

En este caso, `coste=5` tiene el menor error de validación cruzada $0,075$. Además, en la matriz de confusión vemos que el modelo clasifica correctamente 94 casos de los 102 que hay en total en el conjunto de test, es decir, tiene un $7,84\,\%$ de fallo. Vamos a probar ahora un kernel polinomial para ver si conseguimos mejorar el error obtenido.

```
support_vector_machine(k="polynomial")
```

```
##         truth
## predict  0  1
##       0 47  7
##       1  1 47
## Tamaño total del conjunto de test: 102
## Error por validación cruzada de 10 particiones del modelo: 0.09862745
```

Usando el kernel polinomial vemos que el modelo tiene $0,099$ error de validación cruzada y clasifica correctamente 94 de los 102 casos, por tanto, obtenemos un peor resultado: un $7,84\,\%$ de error, al igual que con el modelo lineal. En este caso, a pesar de obtener un igual error de test, el error de validación cruzada del modelo lineal es mejor. Vamos a probar ahora un kernel radial para ver si conseguimos mejorar el error obtenido por el kernel lineal:

```
support_vector_machine(k="radial")
```

```
##         truth
## predict  0  1
##       0 47  6
##       1  1 48
## Tamaño total del conjunto de test: 102
## Error por validación cruzada de 10 particiones del modelo: 0.07705882
```

Usando el kernel radial vemos que el modelo clasifica correctamente 95 de los 102 casos, obteniendo un error del $6,86\,\%$. Por validación cruzada, vemos que el error obtenido es del $0,077$. Por tanto, el mejor modelo obtenido hasta el momento es el radial.

Por último, vamos a probar el kernel *sigmoid* para ver si conseguimos mejorar el modelo lineal:

```
support_vector_machine(k="sigmoid")
```

```
##         truth
## predict  0  1
##       0 43 15
##       1  5 39
## Tamaño total del conjunto de test: 102
## Error por validación cruzada de 10 particiones del modelo: 0.1777647
```

Con este kernel el modelo clasifica correctamente 82 de los 102 casos y el error obtenido sube al $19,61\,\%$. Este kernel ha dado el peor resultado de todos en test y en entrenamiento, ya que su su error de validación cruzada es $0,17$

En resumen, el mejor modelo obtenido ha sido con el kernel lineal que nos daba un error del $8\,\%$.

2.4 Estimando el error de test y train por validación cruzada de 5 particiones

Para estimar el error de test y training del modelo SVM obtenido, podemos usar la función `tune` al igual que hicimos en el caso del KNN.

```
tune_svm = tune(svm, crim01 ~ ., data=Boston[divB$train,], kernel="radial",
                    cost=5, tunecontrol = tune.control(cross=5))
cat("Error por validación cruzada de 5 particiones del modelo:", tune_svm$best.performance)
```

```
## Error por validación cruzada de 5 particiones del modelo: 0.06932099
```

obtenemos un error de train del $6,9\,\%$. Ahora al modelo obtenido con validación cruzada de 5 particiones le calculamos el error de test:

```
ypred = predict(tune_svm$best.model, Boston[divB$test,])
table(predict=ypred, truth=Boston$crim01[divB$test])
```

```
##        truth
## predict  0  1
##       0 46  7
##       1  2 47
```

```
cat("Tamaño total del conjunto de test:",length(divB$test),"\n")
```

```
## Tamaño total del conjunto de test: 102
```

Teniendo en cuenta que el modelo clasifica correctamente 93 de los 102 casos totales, el modelo tiene un error de test del $8,82\,\%$.

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 de 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 de 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*.**

3.1 Dividiendo los datos en train y test

Para hacer el ejercicio anterior, dividí los datos tal y como se especifica en el enunciado: $80\,\%$ de train y $20\,\%$ de test. Por tanto, voy a mantener la misma división para este ejercicio, eliminando la columna `crim01` que añadí.

```
Boston$crim01 = NULL
```

3.2 Ajustando un modelo de regresión usando *bagging*

Para poder usar *bagging*, necesitamos usar la función `randomForest` de `R`, ya que *bagging* es un caso especial de *random forest* con $m = p$, donde m es el número de predictores del modelo y p es el número de predictores totales. Esto es indicado en la función con el parámetro `mtry`, al que debemos darle como valor el número de atributos de `Boston`, menos 1, `medv`.

```
library(randomForest)
bag.boston = randomForest(medv ~ ., data = Boston, subset=divB$train,
                          mtry=ncol(Boston)-1)
bag.boston
```

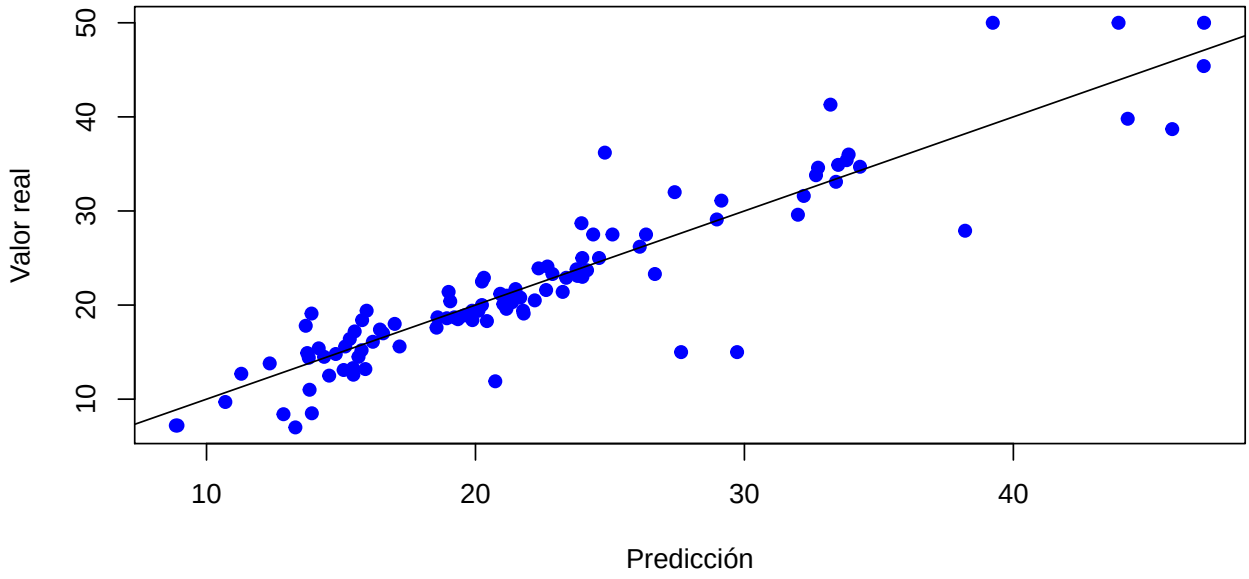
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = ncol(Boston) -      1, subset = divB$train)
##                Type of random forest: regression
```

```
##                       Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 9.864764
##                     % Var explained: 88.29
```

Con *bagging* obtenemos un RSS[1] de 9,86 en los datos de entrenamiento. Ahora vamos a ver el rendimiento del modelo en los datos de test:

```
error_test <- function(model=bag.boston,...) {
    yhat = predict(model, newdata=Boston[divB$test,],...)
    plot(yhat, Boston$medv[divB$test],pch=19, col="blue",xlab="Predicción",ylab="Valor real")
    abline(0,1)
    mean((yhat - Boston$medv[divB$test])^2)
}
```

```
error_test()
```




```
## [1] 13.37389
```

El MSE[2] del modelo es 13,37 en los datos de test, como es normal es algo más alto que el error de entrenamiento pero, como se aprecia en la gráfica, la predicción obtenida se acerca al valor real en algunos casos. En otros, el valor predecido no tiene mucho que ver con el real. Por tanto, a pesar de que el modelo obtenido es aceptable también es cierto que se puede mejorar.

3.3 Ajustando un modelo de regresión *Random Forest*

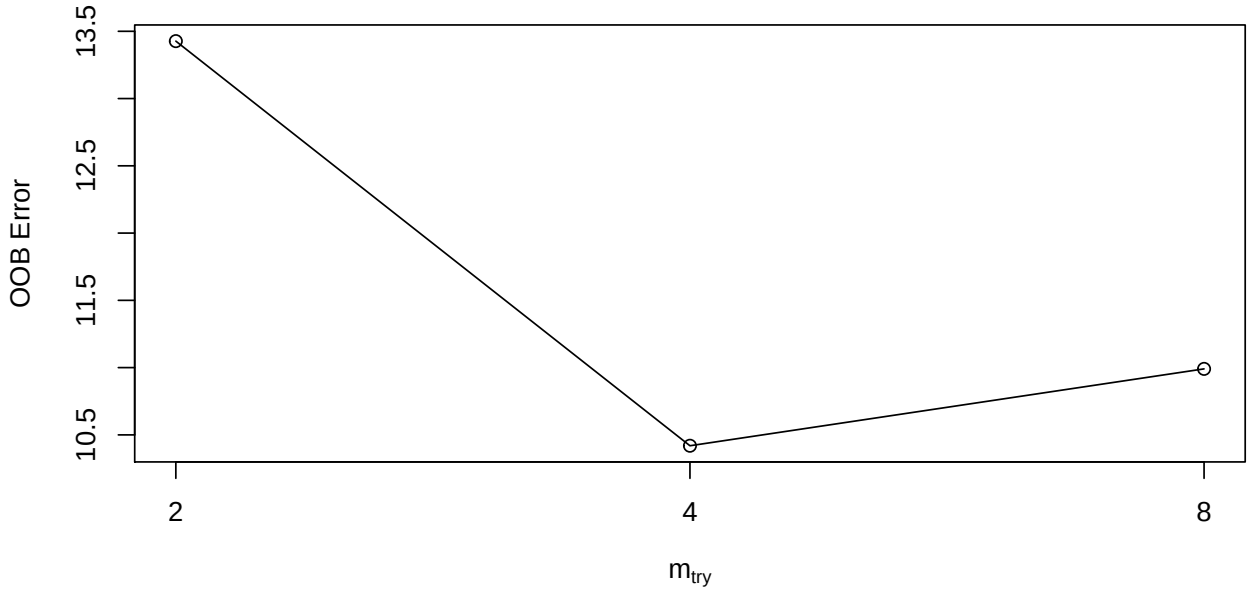
Al igual que hicimos en los ejercicios anteriores, podemos optimizar el número de predictores a usar mediante la función `tuneRF`. Esta función tiene por defecto el flag `trace`, que permite ver el progreso del algoritmo y además, también hemos activado el flag `doBest` para que la función nos devuelva el mejor modelo encontrado directamente.

```
modelo_rf = tuneRF(x=subset(Boston, select=-medv), y=Boston$medv, doBest=T, subset=divB$train)
```

```
## mtry = 4  OOB error = 10.41924
## Searching left ...
```

[1] Residual Sum of Squares
[2] Mean Squared Error

```
## mtry = 2     OOB error = 13.42682
## -0.2886562 0.05
## Searching right ...
## mtry = 8     OOB error = 10.99057
## -0.05483363 0.05
```

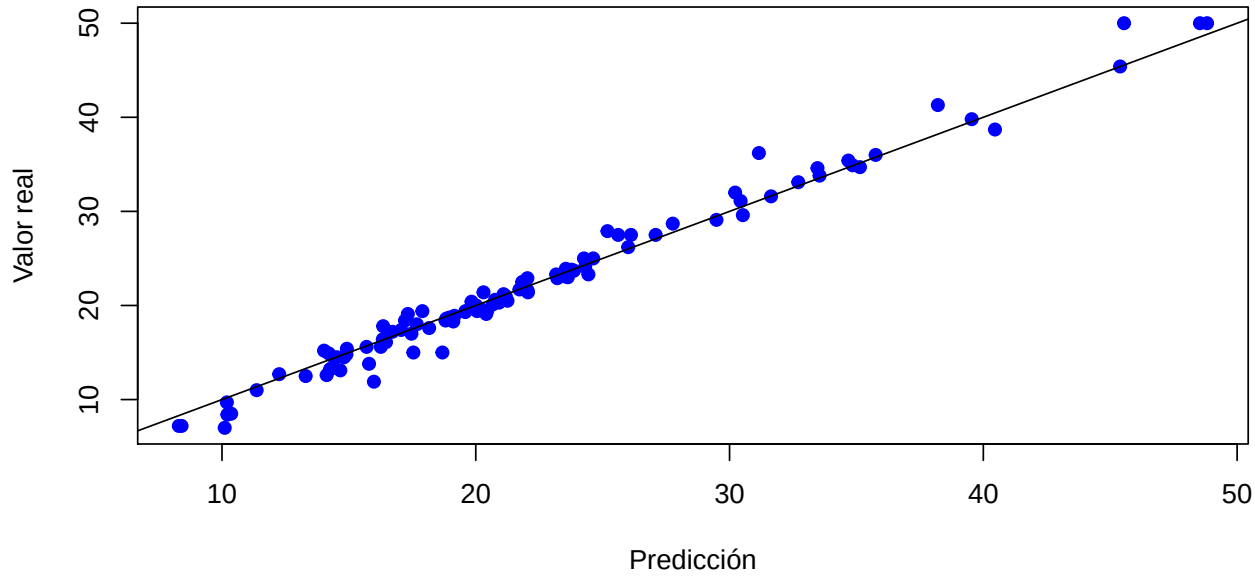


El m recomendado para un problema de regresión es $p/3$, que sería 4 en este caso, justo el m con el que hemos obtenido el mejor modelo.

```
modelo_rf
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = res[which.min(res[, 2]), 1],      subset = ..1)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 10.10899
##                     % Var explained: 88.03
```

```
error_test(model=modelo_rf)
```

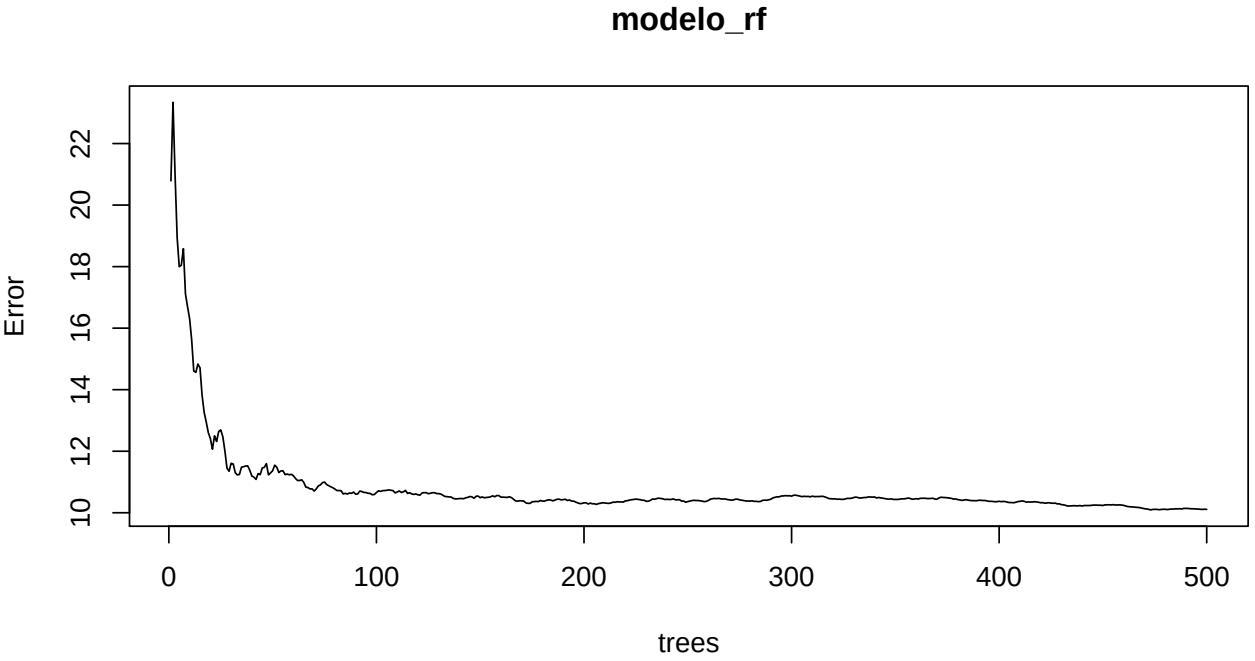



```
## [1] 1.721576
```

En comparación con *bagging*, el RSS obtenido con el conjunto de train es mayor (10, 11 contra 9, 86) pero en cuanto al MSE en el conjunto de test, es muchísimo mejor éste modelo que el anterior (1, 72 contra 13, 37). En el anterior, el valor predecido y el real se alejaban bastante, hasta el punto en el que la gráfica que obtenemos tiene bastantes más puntos dispersos. Si con el modelo anterior dije que era mejorable, considero que con este modelo hemos conseguido esa mejora.

En cuanto al número óptimo de árboles, si representamos el modelo obtenido en una gráfica vemos la relación entre el error obtenido y el número de árboles:

```
plot(modelo_rf)
```

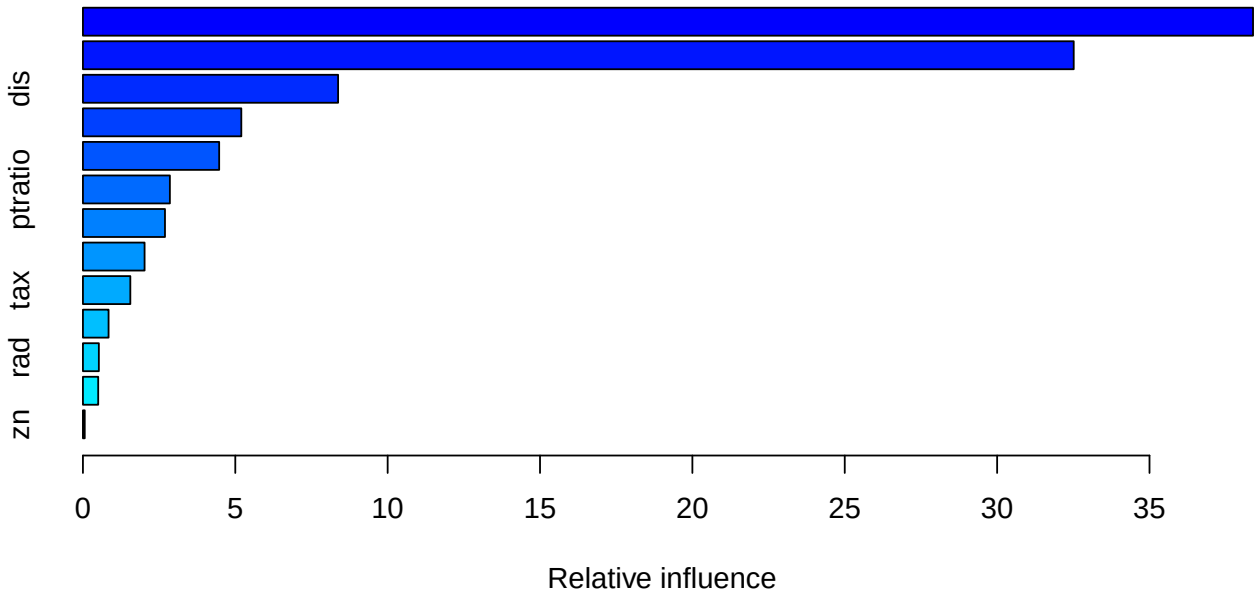


Como se ve en la gráfica, cuanto mayor es el número de árboles menor es el error obtenido. Ahora bien, si aumentamos demasiado éste número podemos llegar a sobreajustar el modelo. Por tanto, considero que el valor por defecto que se usa en `R`, 500 árboles, obtiene un buen resultado.

3.4 Ajustando un modelo de regresión *Boosting*

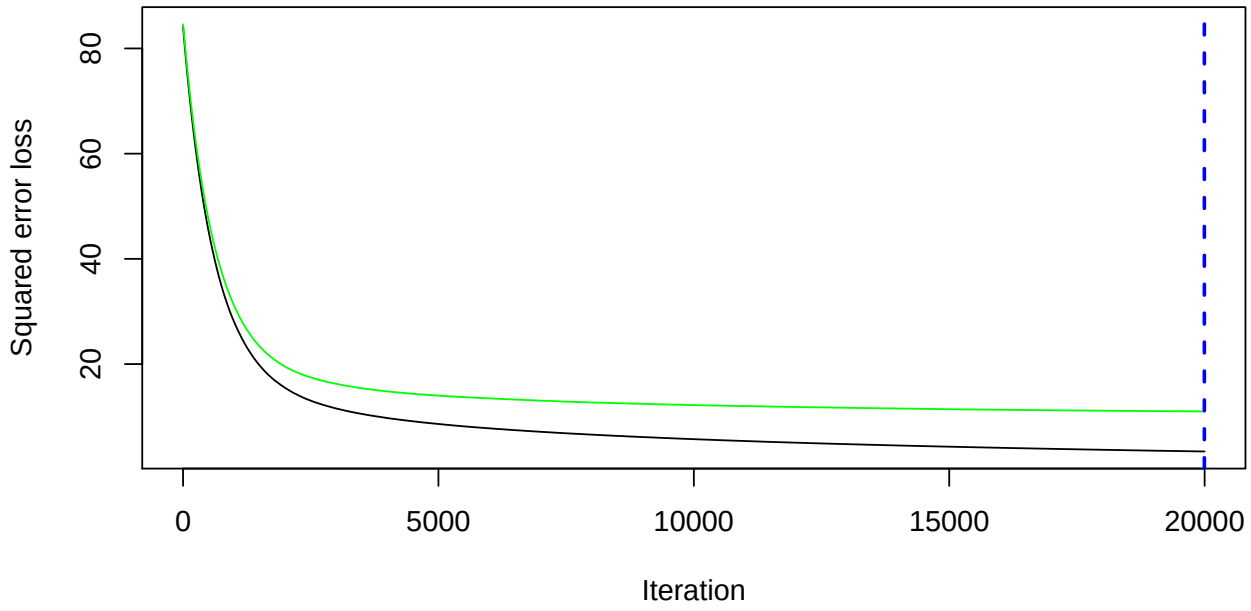
Para poder hacer un modelo con *Boosting*, tenemos que usar la función `gbm` del paquete `gbm`. Debido a que estamos ajustando un modelo de regresión, usamos el parámetro `distribution="gaussian"`. Usamos los parámetros `n.trees` y `interaction.depth` para poder obtener un mejor modelo. Con la función `gbm.perf` hemos comprobado que `n.trees=20000` es suficiente.

```
library(gbm)

boost.boston = gbm(medv ~ ., data=Boston[divB$train,], distribution="gaussian",
                   n.trees=20000, cv.folds=10, interaction.depth=floor(sqrt(ncol(Boston)-1)))

summary(boost.boston)
```




```
##             var     rel.inf
## lstat     lstat 38.39778856
## rm           rm 32.51367744
## dis         dis  8.37329184
## nox         nox  5.19730766
## crim       crim  4.47065521
## ptratio ptratio  2.85302238
## age         age  2.69267988
## black     black  2.02270724
## tax         tax  1.55520087
## chas       chas  0.84158923
## rad         rad  0.52231784
## indus     indus  0.50079982
## zn           zn  0.05896203
```

```
gbm.perf(object=boost.boston,method="cv")
```

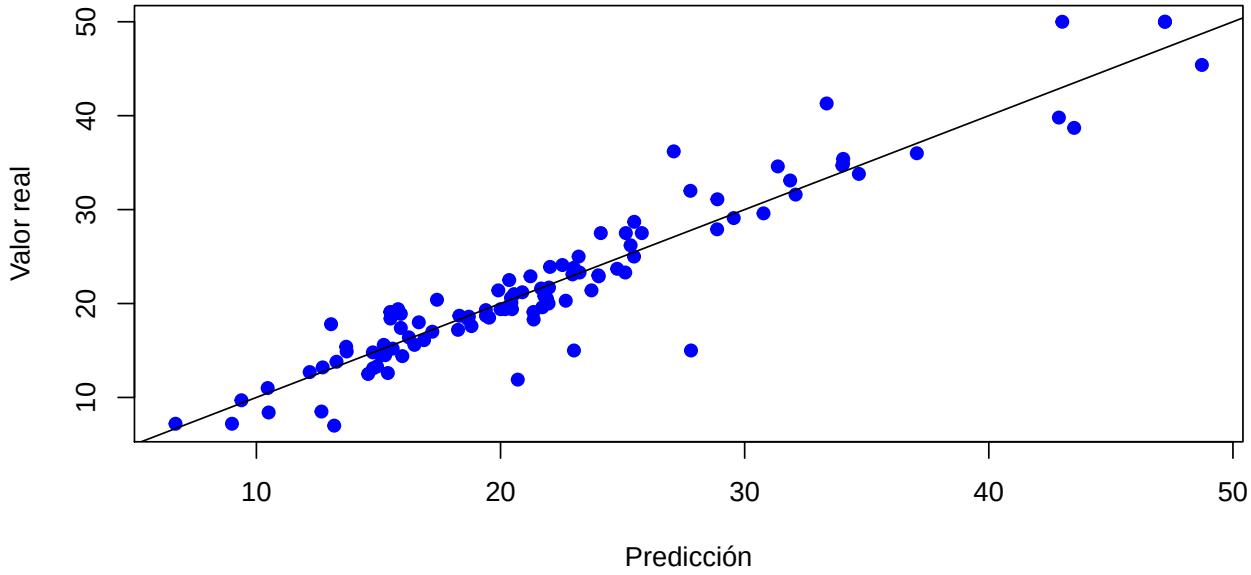



```
## [1] 19995
```

Según el modelo obtenido, las dos variables más influyentes en nuestro modelo son `lstat` y `rm`.

Vamos a calcular el error de test del modelo, para compararlo con *bagging* y *random forest*:

```
error_test(model=boost.boston,n.trees=boost.boston$n.trees)
```




```
## [1] 8.569036
```

El error obtenido es mejor que el de *bagging* pero no supera al de *random forest*. Además, el coste computacional de *boosting* es mucho mayor que el del resto de métodos. Por tanto, en cuanto a relación coste computacional y error de test obtenido, para mí sale ganando *random forest*. En cuanto a *boosting* y *bagging*, es cierto que con *boosting* obtenemos mejor resultado, pero con un coste computacional mayor. Por tanto, la elección entre uno u otro dependerá de la preferencia que tengamos en cuanto a obtener un mejor resultado u obtenerlo en un tiempo menor.

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 como predictoras (usar el 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.**
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)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?**

4.1 Dividiendo los datos en train y test

En el enunciado se especifica que 800 datos deben ser de entrenamiento y el resto para test. Pero la función para dividir datos desarrollada necesita el porcentaje de datos que se usa para entrenamiento, por lo que en la llamada a dicha función calculamos el porcentaje:

```
divOJ = divide_datos(percent_train = 800/nrow(OJ), dataset = OJ)
```

4.2 Ajustando un árbol a los datos de entrenamiento

Debido a que la variable `Purchase` parece tomar valores de clase (`CH` y `MM`), pienso que debemos ajustar un árbol de clasificación. Para ello, usamos la función `tree` de la librería `tree`.

```
library(tree)
attach(OJ)
tree.oj = tree(Purchase ~ ., OJ, subset=divOJ$train)
summary(tree.oj)
```

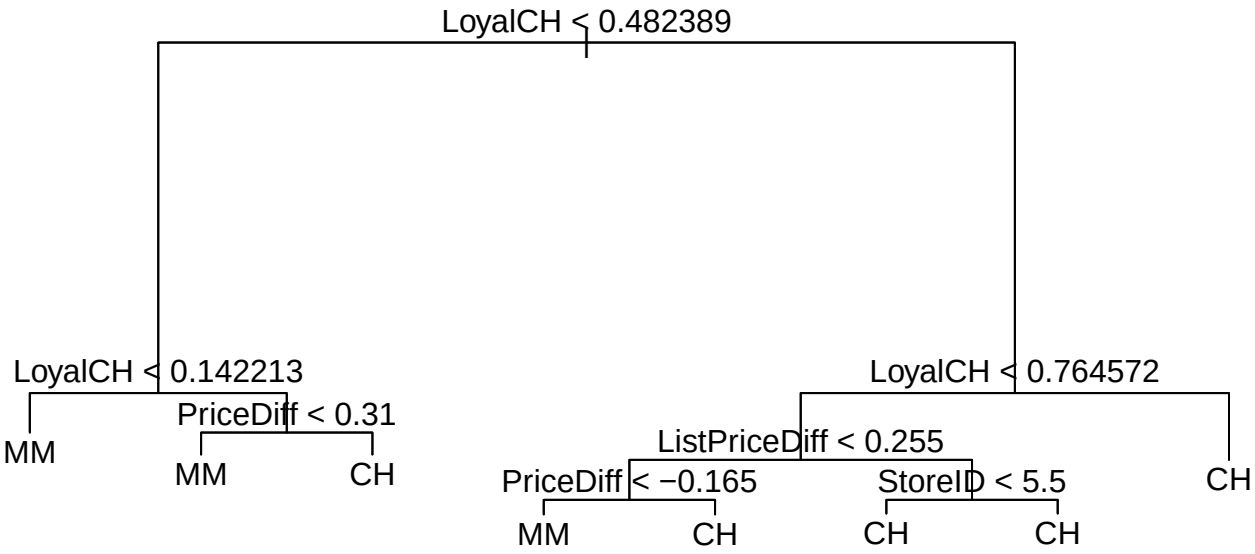
```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ, subset = divOJ$train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff" "StoreID"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7327 = 580.3 / 792
## Misclassification error rate: 0.1562 = 125 / 800
```

El error de entrenamiento del árbol obtenido es del $15\,\%$ y la *deviance*, del $0,73$. Bajo mi opinión, ambos valores son demasiado altos, y por tanto, el árbol obtenido es bastante malo. El número de nodos terminales obtenidos es 8, por tanto creo que hemos obtenido un árbol mediano.

4.3 Representando el árbol

Para representar el árbol junto a las etiquetas de cada nodo debemos usar las dos siguientes funciones:

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



La variable más relacionada con `Purchase` parece ser `LoyalCH`, ya que es la variable que más discrimina en todo el conjunto de entrenamiento. El resto de las variables usadas tienen una relación más débil con `Purchase`, ya que discriminan muchísimo menos que `LoyalCH`. Otra cosa que llama la atención sobre el árbol es que la mayoría de nodos terminales acaban en la clase `CH` (5 nodos contra 3). Por tanto, podemos deducir que hay una mayoría de datos con clase `CH`:

```
length(c(Purchase[divOJ$train])[c(Purchase[divOJ$train]) == 2])
```

```
## [1] 299
```

```
length(c(Purchase[divOJ$train])[c(Purchase[divOJ$train]) == 1])
```

```
## [1] 501
```

Esto puede hacer que cuando tengamos un dato de la clase `MM` obtengamos una clasificación errónea con más probabilidad que cuando el dato sea de la clase `CH`.

4.4 Prediciendo los datos de test

Para ver si realmente el modelo obtenido es bueno o malo, debemos estimar el error de test y no quedarnos sólo con el de entrenamiento. Para ello, predeciremos los datos y generaremos la matriz de confusión:

```
tree.pred = predict(tree.oj, OJ[divOJ$test,], type="class")
t = table(tree.pred, OJ$Purchase[divOJ$test])
print(t)
```

```
##
## tree.pred  CH  MM
##        CH 132  31
##        MM  20  87
```

```
cat("Tamaño del conjunto de test: ", length(divOJ$test), "\n")
```

```
## Tamaño del conjunto de test:  270
```

```
cat("Error de test del modelo: ", (t[2]+t[3])/length(divOJ$test))
```

```
## Error de test del modelo:  0.1888889
```

```
cat("Precisión del test: ", (t[1]+t[4])/length((divOJ$test)))
```

```
## Precisión del test:  0.8111111
```

El modelo ha clasificado correctamente 219 de los 270 casos de test totales y ha clasificado incorrectamente 51. Por tanto, el error de test del modelo es del $18,9\,\%$ y su precisión, del $79,26\,\%$. Esto era de esperar, pues ya dijimos al discutir el modelo obtenido que éste era bastante malo.

4.5 Determinando el tamaño óptimo del árbol

La función `cv.tree` ejecuta una validación cruzada con K particiones para encontrar el tamaño óptimo del árbol, es decir, el número óptimo de nodos terminales. Para que la validación cruzada se haga teniendo en cuenta el número de ejemplos mal clasificados, usamos el parámetro `FUN=prune.misclass`. Tenemos que tener en cuenta que el nombre `dev` se corresponde con el error de validación cruzada:

```
cv.oj = cv.tree(tree.oj, FUN=prune.misclass)
data.frame(cv.oj$size, cv.oj$dev)
```

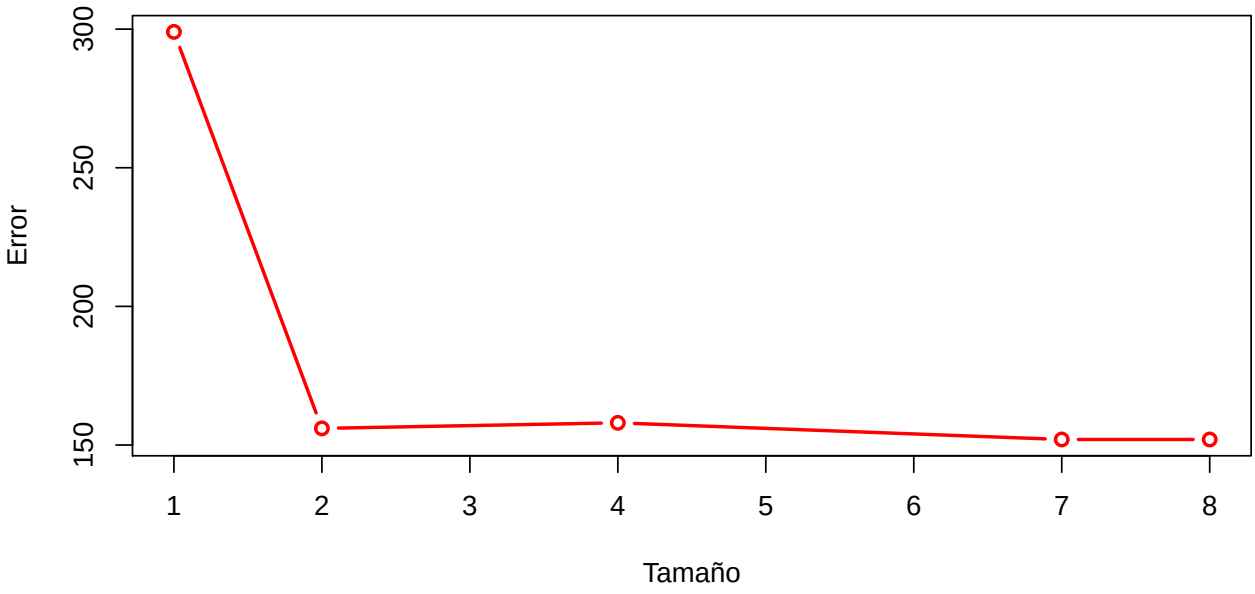
```
##   cv.oj.size cv.oj.dev
## 1          8       152
## 2          7       152
## 3          4       158
## 4          2       156
## 5          1       299
```

El mínimo error de validación cruzada es 152. Dicho error se da tanto en el árbol de tamaño 8 como en el árbol de tamaño 7, por tanto, ambos tamaños son óptimos para el árbol. A la hora de escoger uno de los dos modelos, es mejor escoger el de tamaño 7, pues al ser más simple generaliza más.

4.6 Representando gráficamente el tamaño en función del error de CV

Para representar gŕaficamente el tamaño del árbol en función del error de validación cruzada, ejecutamos el siguiente comando:

```
plot(cv.oj$size, cv.oj$dev, type="b", xlab="Tamaño", ylab="Error", col="red", lwd=2)
```



Tal y como dije antes, tanto el tamaño 7 como el 8 tienen la menor tasa de error de validación cruzada.