Práctica 3. Aprendizaje Automático

José Carlos Martínez Velázquez

28 de Mayo de 2016

Ejercicio 1

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

1. 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.

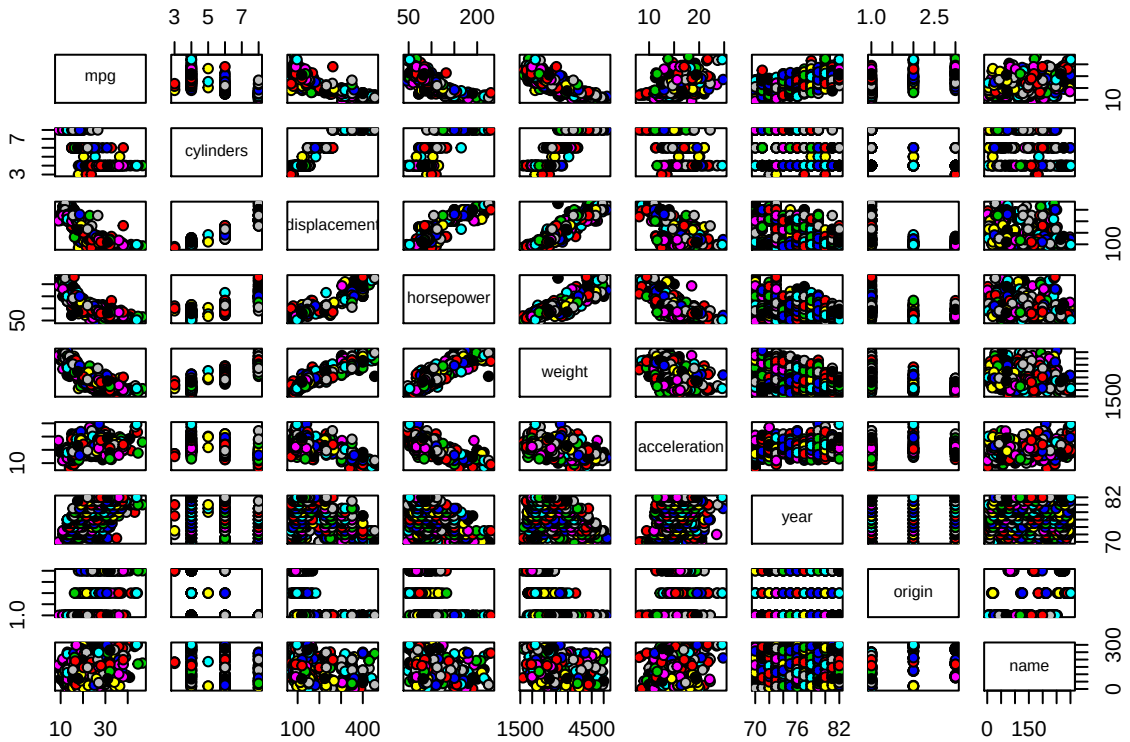
Para utilizar los datasets de ISLR hay que instalar y usar dicho paquete, esto lo haremos con las siguientes órdenes:

```
install.packages("ISLR")
library("ISLR")
attach(Auto)
```

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.

Usando `pairs()` tenemos un gráfico múltiple que nos permite comparar todas con todas las características.

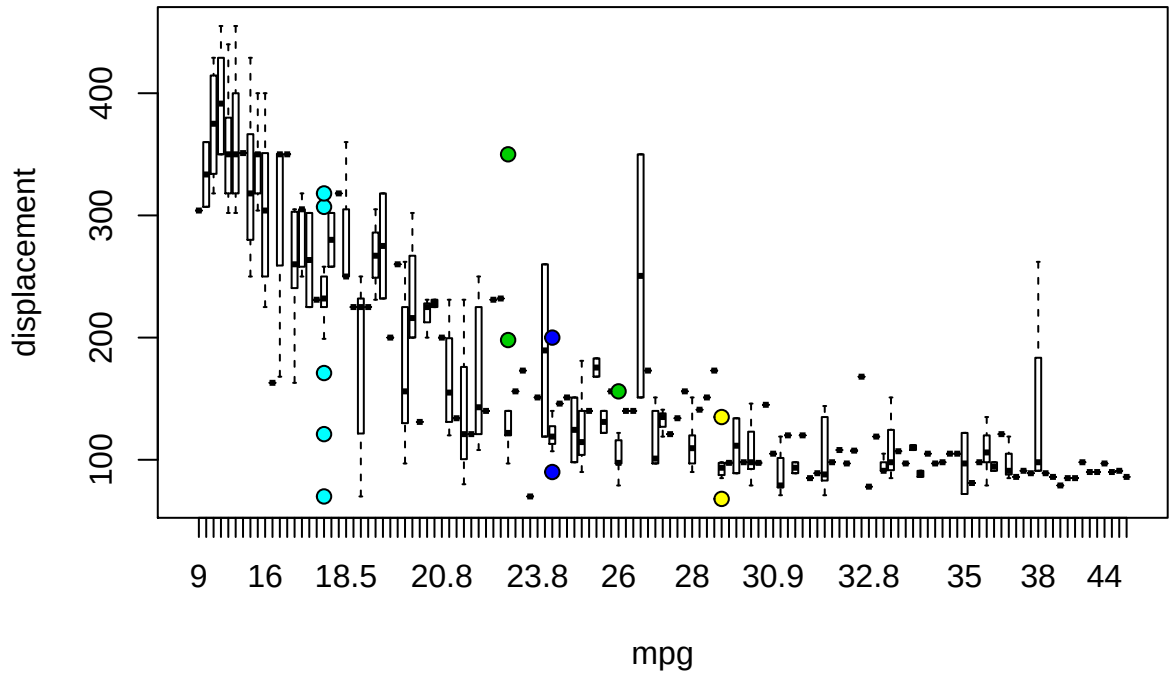
```
pairs(Auto,pch=21, bg=unique(c(unclass(mpg))))
```



Como vemos, parece haber un patrón claro entre la característica `mpg` y las características `displacement` (capacidad del motor), `horsepower` (cantidad de cv) y `weight` (peso).

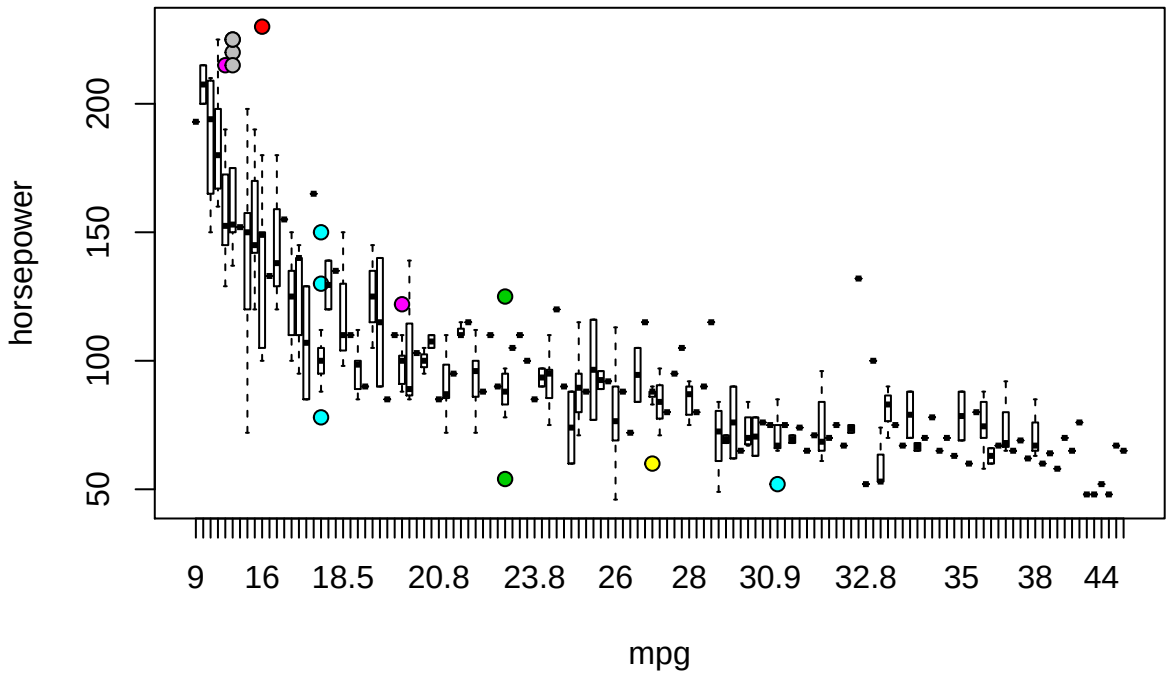
Vamos a ver los rangos (boxplots) de cada variable predictora con la variable que queremos predecir. Comenzamos con displacement:

```
boxplot(displacement ~ mpg,
        data=Auto,
        pch=21,
        bg=unique(c(unclass(mpg))),
        xlab="mpg",ylab="displacement")
```



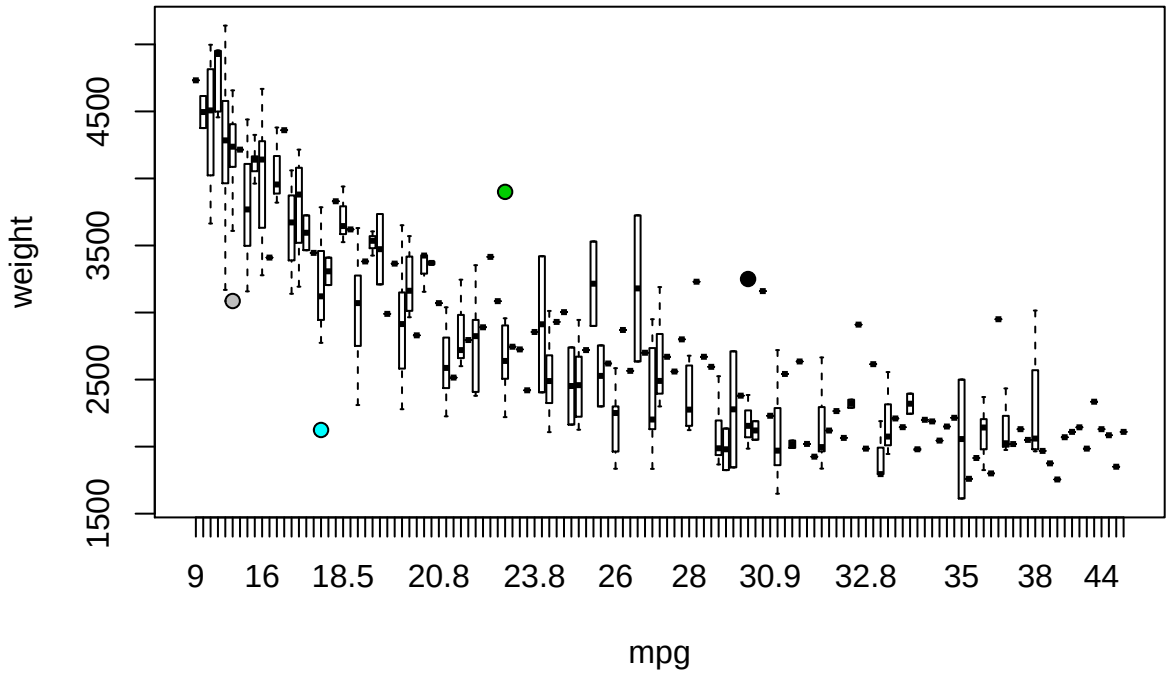
Veamos ahora la relación con horsepower:

```
boxplot(horsepower ~ mpg,
        data=Auto,
        pch=21,
        bg=unique(c(unclass(mpg))),
        xlab="mpg",ylab="horsepower")
```



Por último, con weight.

```
boxplot(weight ~ mpg,
        data=Auto,
        pch=21,
        bg=unique(c(unclass(mpg))),
        xlab="mpg",ylab="weight")
```



Cada caja (rectángulo) en los gráficos nos habĺa del rango de valores que puede tomar la variable predictora para un valor de consumo. De este modo, sabemos que dado un valor de consumo, la variable predictora

toma valores acotados en un determinado intervalo. Aunque los valores en que están medidas las diferentes variables predictoras son de diferente orden, vemos que la relación de todas ellas con `mpg` (consumo) es muy parecida.

b) Seleccionar las variables predictoras que considere más relevantes.

Tras el análisis del apartado anterior, las variables predictores que consideraré más relevantes son `displacement`, `horsepower` y `weight`.

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

El dataset podría estar ordenado, por lo que si partimos el dataset sin más, podríamos estar sesgando los resultados. Lo que haremos será barajar el dataset y seleccionar el 80% de primeros ejemplos para training y el resto, serán incluidos en el conjunto de entrenamiento. Antes de barajar o hacer cualquier cosa que conlleve operaciones aleatorias, conviene fijar una semilla para poder repetir los experimentos, lo haremos con `set.seed()`. Por ahora, nos quedaremos sólo con los índices que hacen referencia a los ejemplos seleccionados, por lo que tendremos un conjunto de índices para entrenamiento y otro conjunto de índices para test.

```
set.seed(199)
complet_Auto<-Auto
#Eligiendo las posiciones de los ejemplos (barajar)
orden<-runif(nrow(complet_Auto))
complet_Auto<-complet_Auto[order(orden),]
#Separación de entrenamiento y test
porcentaje_training<-0.8
nEjemplosTrain<-trunc(nrow(complet_Auto)*porcentaje_training)
i_training<-1:nEjemplosTrain
i_test<-(nEjemplosTrain+1):nrow(complet_Auto)
#Quito el nombre, que es una variable no numérica
complet_Auto<-complet_Auto[,1:ncol(complet_Auto)-1]
```

d) Crear una variable binaria, mpg01, que será igual 1 si la variable mpg contiene un valor por encima de la mediana, y -1 si mpg contiene un valor por debajo de la mediana. La mediana se puede calcular usando la función median(). (Nota: puede resultar útil usar la función data.frames() para unir en un mismo conjunto de datos la nueva variable mpg01 y las otras variables de Auto).

Copiaremos la columna `mpg` de auto y la iremos modificando conforme las condiciones dadas. Una vez hecho, partiremos esta columna en entrenamiento y test, ya que tenemos guardados los índices que corresponden a un conjunto u otro. Incluiremos cada columna en el dataset de entrenamiento o test según correspondan.

```
#Construcción de los valores de la columna mpg01
mpg01<-complet_Auto$mpg
mpg01[mpg01<median(complet_Auto$mpg)]<-(-1)
mpg01[mpg01>=median(complet_Auto$mpg)]<-1
#Quitar la columna mpg y añadir mpg01
complet_Auto<-complet_Auto[,-1]
complet_Auto<-data.frame(mpg01,complet_Auto)
```

Una vez añadida la variable `mpg01`, hay que normalizar (las clases deben ser 0/1). Lo haremos por separado. Esto es porque no es lo mismo considerar los máximos y mínimos globales, es decir, de todo el dataset, que considerar los máximos y mínimos de cada conjunto por separado. Si lo hiciéramos de la primera forma,

podríamos estar "envenenando" el conjunto de test si el máximo o el mínimo (o ambos) se quedasen en el conjunto de entrenamiento una vez partido, pues estaríamos incluyendo datos de entrenamiento en el test.

```
normalizar<-function(data,max_deseado=1,min_deseado=0){
  max<-numeric(ncol(data))
  min<-numeric(ncol(data))
  for(i in 1:ncol(data)){
    max[i]=max(data[,i])
    min[i]=min(data[,i])
  }
  for(i in 1:ncol(data)){
    data[,i]<-((max_deseado-min_deseado)/(max[i]-min[i]))*((data[,i]-min[i])+min_deseado)
  }
  return(data)
}
#Separación, normalización por separado y unión.
conj_training<-complet_Auto[i_training,]
conj_test<-complet_Auto[i_test,]
conj_training<-normalizar(conj_training)
conj_test<-normalizar(conj_test)
complet_Auto<-rbind(conj_training,conj_test)
```

d.1)Ajustar un modelo de regresión Logística a los datos de entrenamiento y predecir mpg01 usando las variables seleccionadas en b). ¿Cuál es el error de test del modelo? Justificar la respuesta.

```
attach(complet_Auto)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     mpg01
## The following objects are masked from Auto:
##
##     acceleration, cylinders, displacement, horsepower, origin,
##     weight, year
```

```
model_rl<-glm(mpg01 ~
              displacement+horsepower+weight,
              data =complet_Auto, subset=i_training, family=binomial("logit"))

pred_rl_ts<-predict(model_rl,conj_test[,c(3,4,5)],type="response")
pred_rl_ts01<-pred_rl_ts
pred_rl_ts01[pred_rl_ts01>0.5]<-1
pred_rl_ts01[pred_rl_ts01<0.5]<-0
error_rl<-length(pred_rl_ts01[pred_rl_ts01!=conj_test[,1]])/nrow(conj_test)
print(paste("Error en test (regresion logistica):",error_rl))
```

```
## [1] "Error en test (regresion logistica): 0.126582278481013"
```

Aunque Regresión logística habla de probabilidades, hemos supuesto que una probabilidad de más del 50% quiere decir que el ejemplo es de la clase por la que se decanta dicha probabilidad, entonces estamos en un escenario de clasificación. Con estas condiciones podemos calcular un porcentaje de error en test basado en la diferencia de las predicciones.

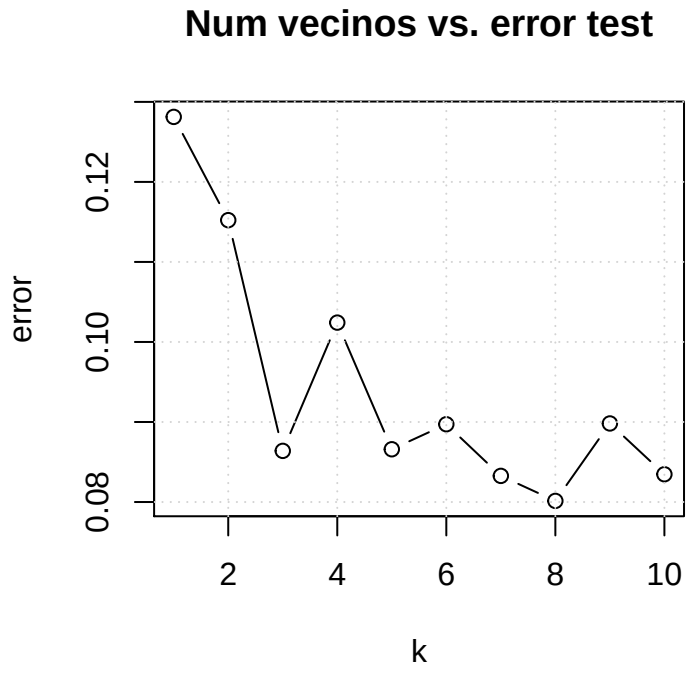
d.2)Ajustar un modelo K-NN a los datos de entrenamiento y predecir mpg01 usando solamente las variables seleccionadas en b). ¿Cuál es el error de test en el modelo? ¿Cuál es el valor de K que mejor ajusta los datos? Justificar la respuesta. (Usar el paquete class de R)

Instalamos el paquete class de R y fijamos una semilla para poder repetir los experimentos:

```
install.packages("class")
install.packages("e1071")
library(class)
library(e1071)
```

Lo primero que haremos será obtener el mejor número de vecinos (k) para el k-NN. Lo haremos con la función `tune.knn()`. Fijaremos una semilla para que los experimentos sean deterministas.

```
set.seed(199)
fit_knn<-tune.knn(x=conj_training[,c(3,4,5)], y=as.factor(conj_training[,1]),
                  k = 1:10,tunecontrol=tune.control(sampling = "cross"), cross=10)
plot(fit_knn,main="Num vecinos vs. error test")
grid()
```




```
best_k<-as.numeric(fit_knn$best.parameters)
print(paste("Numero optimo de vecinos:",best_k))
```

```
## [1] "Numero optimo de vecinos: 8"
```

Como es lógico, nos quedaremos con el número de vecinos `k` que más reduzca el error. Conocido el valor de `k` pasamos a construir el clasificador k-NN.

```
set.seed(199)
train.labels <- conj_training[,1]
```

```
pred_knn<-knn(train=conj_training[,c(3,4,5)],
              test=conj_test[,c(3,4,5)],
              cl=train.labels,
              k = best_k,
              prob=TRUE)
prob_knn<-attr(pred_knn,"prob")

#Modifico probabilidades del siguiente modo:
  #Prob de la clase 0 = p -> prob. global = 1 - p
  #Prob de la clase 1 = p -> prob. global = p
prob_knn<-ifelse(pred_knn==0,1-prob_knn,prob_knn)
m_conf_knn<-table(pred_knn,conj_test[,1])
error_knn<-1-sum(diag(m_conf_knn))/sum(m_conf_knn)
print(paste("Error cometido en test (K-nn): ",error_knn))
```

```
## [1] "Error cometido en test (K-nn):  0.126582278481013"
```

d.3)Pintar las curvas ROC (instalar paquete ROCR en R) y comparar y valorar los resultados obtenidos para ambos modelos.

Hay que instalar y usar el paquete `ROCR`.

```
install.packages("ROCR")
library(ROCR)
```

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

```
## KernSmooth 2.23 loaded
## Copyright M. P. Wand 1997-2009
```

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

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

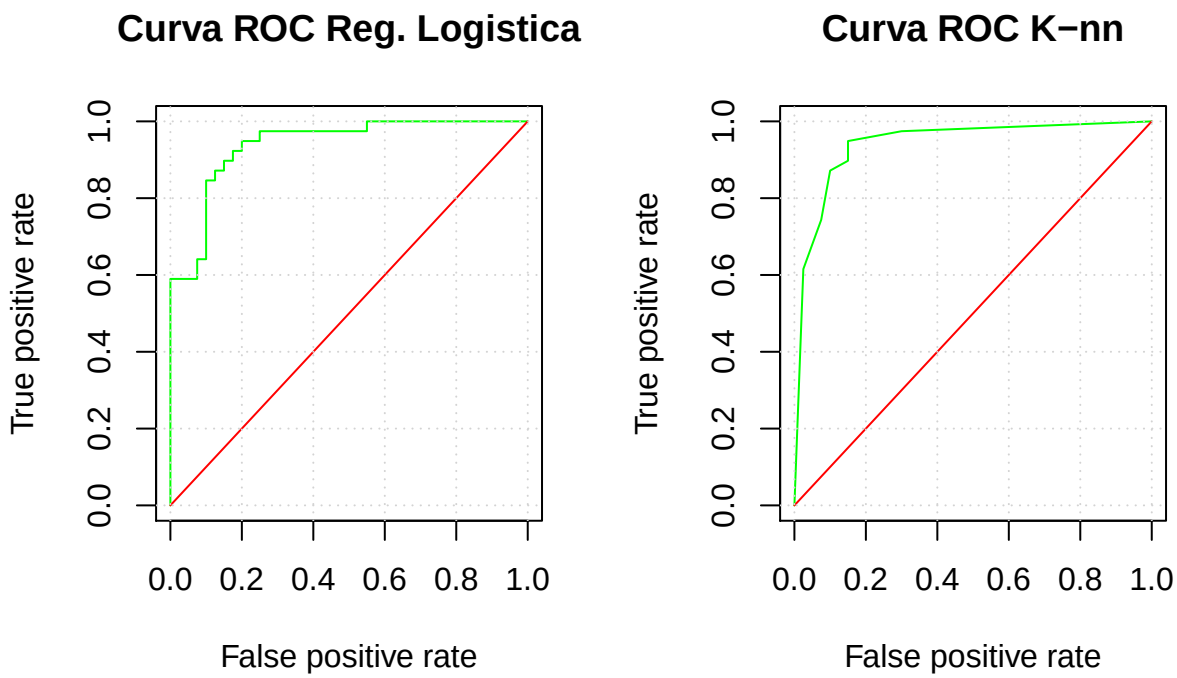
Procedemos a pintar las curvas ROC:

```
par(mfrow=c(1,2))

linexy<-function(x){x}

#CURVA ROC REG LOGISTICA
p_roc_rl <- prediction(pred_rl_ts,conj_test[,1])
perf_rl <- performance(p_roc_rl, "tpr", "fpr")

area_rl<-performance(p_roc_rl,"auc")@y.values[[1]]

plot(perf_rl,main="Curva ROC Reg. Logistica",col="green")
curve(linexy,col="red",add=T)
```

```r
grid()

print(paste("Area curva ROC Reg.Logistica",area_rl))
```

```
## [1] "Area curva ROC Reg.Logistica 0.938461538461538"
```

```r
#CURVA ROC KNN

p_roc_knn <- prediction(prob_knn,conj_test[,1])
perf_knn <- performance(p_roc_knn, "tpr", "fpr")

area_knn<-performance(p_roc_knn,"auc")@y.values[[1]]

plot(perf_knn,main="Curva ROC K-nn",col="green")
curve(linexy,col="red",add=T)
grid()
```




```r
print(paste("Area curva ROC k-NN",area_rl))
```

```
## [1] "Area curva ROC k-NN 0.938461538461538"
```

```r
par(mfrow=c(1,1))
```

Vemos cómo k-NN optimizado tiene una tasa de falsos positivos menor. Lo que siempre se prefiere es que no haya ni falsos positivos ni falsos negativos, pero es es inevitable. Las curvas ROC nos dicen que Reg. Logística tiene una tasa de falsos positivos mayor que k-NN optimizado. Del mismo modo, la tasa de aciertos es mayor en k-NN optimizado. Sin duda alguna, el clasificador k-NN optimizado obtiene mejores resultados que Reg. Logística en este problema. Otro dato que apoya el resultado es el área bajo la curva, que en curvas ROC, cuanto más grande, mejor. Vemos cómo en el caso de k-NN optimizado el área bajo la curva es mayor y por lo tanto, obtiene mejores registros de acierto.

Bonus-1. 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.

Hagamos las 5 particiones con el uso de `cut()`, que asigna a cada índice a qué grupo va a pertenecer: $Grupo_1, ..., Grupo_5$ de forma secuencial. Como los datos están barajados, no habrá ningún tipo de alteración en los datos de cada grupo.

```
nparticiones<-5
parts <- cut(seq(1,nrow(complet_Auto)),breaks=nparticiones,labels=FALSE)
```

Estimemos ahora el error de test de regresión logística

```
errors_cv_rl<-numeric(nparticiones)
for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(complet_Auto)),testIndexes)

  conj_test_cv<-complet_Auto[testIndexes,]
  conj_training_cv<-complet_Auto[trainIndexes,]

  #Limpiar valores envenenados (NA)
  conj_training_cv<-conj_training_cv[-which(is.na(conj_training_cv)),]

  model_rl<-glm(mpg01 ~
                displacement+horsepower+weight,
                data =complet_Auto, subset=trainIndexes, family=binomial("logit"))

  pred_rl_ts<-predict(model_rl,conj_test_cv[,c(3,4,5)],type="response")
  pred_rl_ts[pred_rl_ts>=0.5]<-1
  pred_rl_ts[pred_rl_ts<0.5]<-0
  errors_cv_rl[i]<-length(pred_rl_ts[pred_rl_ts!=conj_test_cv[,1]])/length(conj_test_cv[,1])
}
print(paste("Error en test validacion cruzada 5cv Reg. Logistica:",mean(errors_cv_rl)))
```

```
## [1] "Error en test validacion cruzada 5cv Reg. Logistica: 0.102207075624797"
```

Vamos ahora con validación cruzada 5cv en k-NN.

```
set.seed(199)
errors_cv_knn<-numeric(nparticiones)
for(i in i:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(complet_Auto)),testIndexes)

  conj_test_cv<-complet_Auto[testIndexes,]
  conj_training_cv<-complet_Auto[trainIndexes,]

  #Limpiar valores envenenados (NA)
  conj_training_cv<-conj_training_cv[-which(is.na(conj_training_cv)),]

  fit_knn<-tune.knn(x=conj_training[,c(3,4,5)], y=as.factor(conj_training[,1]),
                    k = 1:10,tunecontrol=tune.control(sampling = "cross"), cross=10)
  best_k<-as.numeric(fit_knn$best.parameters)
```

```
  train.labels <- conj_training[,1]
  pred_knn<-knn(train=conj_training[,c(3,4,5)],
                test=conj_test[,c(3,4,5)],
                cl=train.labels,
                k = best_k,
                prob=TRUE)

  prob_knn<-attr(pred_knn,"prob")
  prob_knn<-ifelse(pred_knn==0,1-prob_knn,prob_knn)
  m_conf_knn<-table(pred_knn,conj_test[,1])
  errors_cv_knn[i]<-1-sum(diag(m_conf_knn))/sum(m_conf_knn)
}
print(paste("Error de validacion cruzada 5cv k-NN:",mean(errors_cv_knn)))
```

```
## [1] "Error de validacion cruzada 5cv k-NN: 0.0227848101265823"
```

Vemos cómo el error de validación cruzada de k-NN es más bajo que el de regresión logística, por lo que podríamos decir que k-NN en este caso es mejor que regresión logística. En el apartado anterior, viendo las curvas ROC no hubiéramos dicho de forma tan precisa que hubiera una diferencia de error del 8%, pues las curvas no eran tan diferentes. El k-NN tiene una tasa de falsos positivos mayor, pero por contra acierta más (mayor número de verdaderos positivos). Reg. logística, aunque aciera un menos, no da tantos falsos positivos. El clasificador k-NN sale vencedor en este caso porque la diferencia en los falsos positivos no gana a la diferencia de aciertos, que es mayor en k-NN.

Bonus-2. 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.

Hemos visto el error de test de regresión logística, vamos a probar qué tal va el modelo de regresión lineal. Vamos a volver a las condiciones del principio, donde partíamos el dataset, y normalizabamos por separado, quitando la variable nombre. Recuperaremos la variable `mpg` en lugar de la variable `mpg01`. La partición será exactamente la misma.

```
set.seed(199)
#Copiar Auto y barajar
complet_Auto<-Auto
complet_Auto<-complet_Auto[order(orden),]
#Quito el nombre
complet_Auto<-complet_Auto[,1:ncol(complet_Auto)-1]
#Parto el dataset en dos conjuntos de entrenamiento y test
conj_training<-complet_Auto[i_training,]
conj_test<-complet_Auto[i_test,]
#Normalizo con la variable mpg en lugar de mpg01
conj_training<-normalizar(conj_training)
conj_test<-normalizar(conj_test)
```

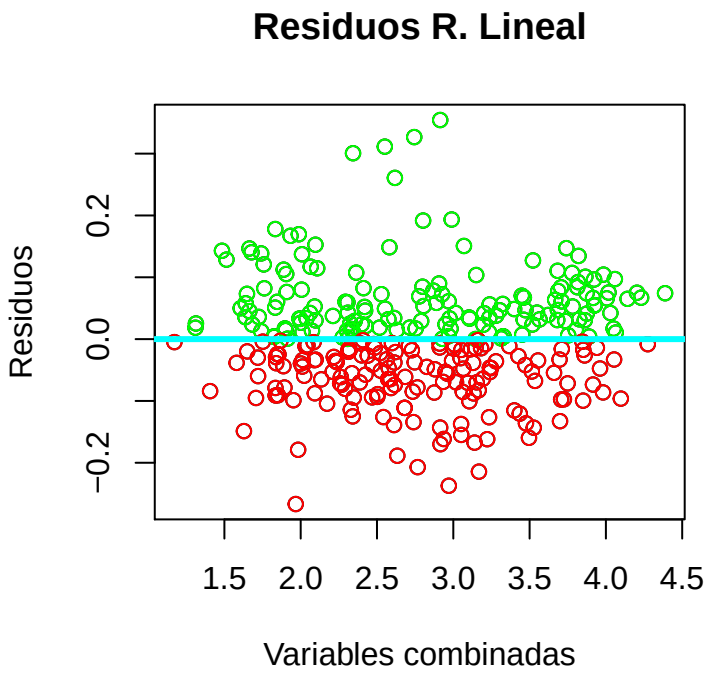
Ajusto el modelo de regresión lineal

```
attach(complet_Auto)
```

```
## The following objects are masked from complet_Auto (position 7):
```

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

```
mod_rlin <- lm(mpg~.,
               data =conj_training)

residuos<-mod_rlin$residuals

x<-conj_training[,2]+conj_training[,3]+conj_training[,4]+
   conj_training[,5]+conj_training[,6]+conj_training[,7]+
   conj_training[,8]

plot(x,
     residuos,
     ylab="Residuos", xlab="Variables combinadas",
     main="Residuos R. Lineal")

points(x[residuos>0],residuos[residuos>0], col="green")

points(x[residuos<0],residuos[residuos<0], col="red")

abline(h=0, col="cyan",lwd=3)
```




```
print(paste("Residuos por encima de 0:",length(residuos[residuos>0])))
```

```
## [1] "Residuos por encima de 0: 156"
```

```r
print(paste("Residuos por debajo de 0:",length(residuos[residuos<0])))
```

```
## [1] "Residuos por debajo de 0: 157"
```

Como vemos, en cuanto a los residuos, el modelo de regresión lineal tiene sentido, pues los residuos que quedan por encima y por debajo de 0 son 156 y 157 ejemplos respectivamente, lo cual está bastante equilibrado.

Vamos ahora a calcular su error en test con validación cruzada.

```r
nparticiones<-5
parts <- cut(seq(1,nrow(complet_Auto)),breaks=nparticiones,labels=FALSE)

errors_cv_rlin<-numeric(nparticiones)
for(i in 1:nparticiones){

  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(complet_Auto)),testIndexes)

  conj_test_cv<-complet_Auto[testIndexes,]
  conj_training_cv<-complet_Auto[trainIndexes,]

  #Limpiar valores envenenados (NA)
  conj_training_cv<-conj_training_cv[-which(is.na(conj_training_cv)),]

  mod_rlin <- lm(mpg~.,
                 data =conj_training_cv)

  pred_rlin_ts<-predict(mod_rlin,conj_test_cv[,2:ncol(conj_test_cv)],type="response")

  errors_cv_rlin[i]<-mean((conj_test_cv[,1]-pred_rlin_ts)*(conj_test_cv[,1]-pred_rlin_ts))
}

print(paste("Error en test (regresion lineal):",mean(errors_cv_rlin)))
```

Tengamos en cuenta que el error calculado no es clasificatorio, sino de regresión, por lo que, el error se calcula con mínimos cuadrados. Si ahora ajustamos el modelo con todos los datos (aunque no se pueda testear), tendríamos un modelo con un error teóricamente inferior a este.

```r
mod_rlin <- lm(mpg~.,data =complet_Auto)
summary(mod_rlin)
```

Aquí, lo único a lo que podemos encomendarnos es al test estadístico. Si miramos el p-valor, vemos cómo se rechaza la hipótesis nula de que las variables no tengan relaciones entre sí y que el modelo sea un mal predictor, es decir, que lo que predice el modelo es válido con un factor de confianza del 95%.

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.

Para usar la base de datos `Boston`, es necesario utilizar la biblioteca MASS.

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

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.

Para usar R-LASSO hay que usar el paquete `glmnet` de R.

```
install.packages("glmnet")
library(glmnet)
```

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

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

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

Regresión LASSO (Least Absolute Shrinkage and Selection Operator) es un método de regresión lineal con selección de variables. La ventaja de R-LASSO es que es un técnica de regresión lineal ya regularizada. R-LASSO pretende, al igual que R-Ridge, penalizar variables que no dicen nada acerca de la salida.

Las expresiones de ambos tipos de regresión se pueden deducir de forma sencilla. Partimos de la expresión de penalización

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda P_\alpha$$

Donde $P_\alpha=\sum_{j=1}^{p}\left((1-\alpha)\beta_j^2+\alpha\left|\beta_j\right|\right)$ es el término de penalización. Cuando $\alpha=0$, estamos ante R-Ridge, y cuando $\alpha=1$ estamos ante R-LASSO. Entonces, la expresión de R-LASSO queda así:

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}\left|\beta_j\right|$$

La parte multiplicada por lambda en R-LASSO se llama norma ℓ_1, que es su forma de penalizar.

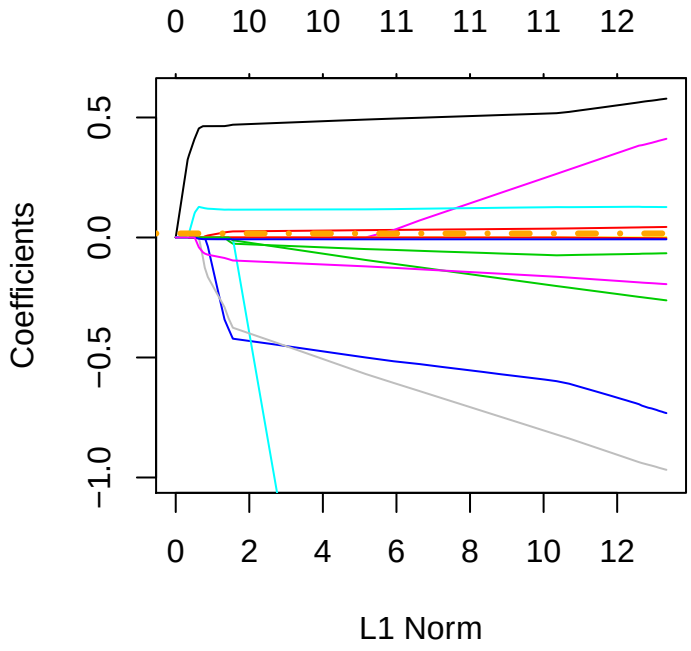
Lo que se pretende es:

$$minimizar_\beta\left\{\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\}\quad sujeto\ a\ \sum_{j=1}^{p}\left|\beta_j\right|\leq s$$

Donde s es una constante que indica lo restrictivos que somos para obtener los coeficientes. Si s es muy grande, apenas seremos restrictivos, por contra, si s es pequeña, la permisividad será mínima.

El otro parámetro es λ. Vamos a estimar el mejor λ mediante validación cruzada. El mejor λ es aquél que penaliza minimizando el error de validación cruzada **(cvm)**.

```
set.seed(199)
x <- as.matrix(Boston[,2:length(Boston)])
y <- as.matrix(Boston[,1])

#Elegimos el mejor lambda por validación cruzada de 10 particiones
cv<-cv.glmnet(x,y,alpha=1,nfolds=10)
plot(cv$glmnet.fit, "norm",ylim=c(-1,0.6),col=c(2:length(Boston)))
best_lambda<-cv$lambda.min
abline(h=best_lambda,col="orange",lty=4,lwd=3)
```




```
print(paste("El mejor lambda es:",best_lambda))
```

```
## [1] "El mejor lambda es: 0.0168006963694516"
```

El gráfico nos muestra los valores que toma lambda por cada variable predictora. Nos interesan aquéllas que estén por encima de un determinado umbral. La línea naranja es el umbral definido, el mejor λ. Vemos cómo sólo cuatro líneas superan el umbral, es decir, sólo cuatro variables son significativas en este modelo. Vamos a ver cuáles son, ajustando el modelo con este umbral. A medida que aumenta la norma L1 (eje x), estamos dotando de precisión al modelo, es decir, vemos si las variables son sustanciales o no. Cuando la norma L1 es pequeña, todas las variables nos dicen lo mismo y estaríamos ante un estimador nulo (o aleatorio).

```
set.seed(199)
#Ajuste de modelo alpha=1 indica lasso; alpha=0 indica Ridge
mdl_lasso <- glmnet(x, y,lambda =best_lambda , family="gaussian",alpha=1)

pred_lasso<-predict(mdl_lasso,type="coef",s=best_lambda)
best_vars<-pred_lasso[,1][pred_lasso[,1]>best_lambda]
```

```
## [1] "Variables seleccionadas:"
```

```
## [1] "(Intercept)" "zn"          "rm"          "rad"         "lstat"
```

En la documentación de R podemos ver qué indica cada variable que ha resultado seleccionada. Dichas variables son las siguientes:

- zn: proporción de zonas residenciales por cada 25.000 pies cuadrados.
- rm: número medio de habitaciones por vivienda.
- rad: índice de acceso a autopistas radiales.
- lstat: porcentaje de estatus más bajo de la población (porcentaje de clase baja).

Con los conocimientos que todos tenemos sobre el tema, parece que tiene sentido que estas variables resulten útiles para determinar si la tasa de criminalidad en un distrito es alta o no, pues, en términos generales, hacen referencia al desarrollo de la zona y el estatus/nivel de alfabetización de sus ciudadanos.

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".

En el apartado anterior vimos cómo obtener la expresión de penalización de R-LASSO con el valor de α. Si ahora $\alpha = 0$, entonces la expresión de R-Ridge es:

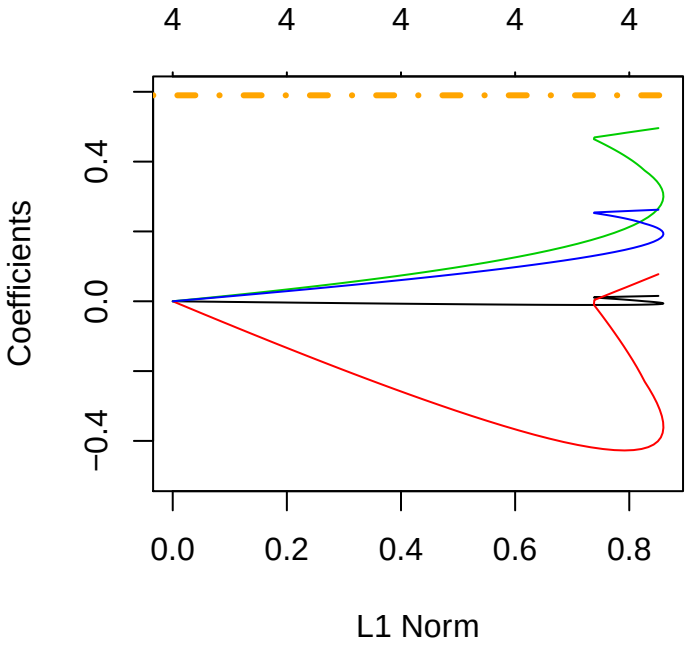
$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

Lo que se pretende en R-Ridge es:

$$minimizar_\beta\left\{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\} \quad sujeto\ a\ \sum_{j=1}^{p}\beta_j^2 \leq s$$

Vamos a obtener el mejor valor de λ y ver qué pasa en R-Ridge.

```
set.seed(199)
x <- as.matrix(Boston[,c(2,6,9,13)])
y <- as.matrix(Boston[,1])

#Elegimos el mejor lambda por validación cruzada de 10 particiones
cvfit<-cv.glmnet(x,y,alpha=0,nfolds=10)
plot(cvfit$glmnet.fit, "norm",ylim=c(-0.5,0.6))
best_lambda<-cvfit$lambda.min
abline(h=best_lambda,col="orange",lty=4,lwd=3)
```



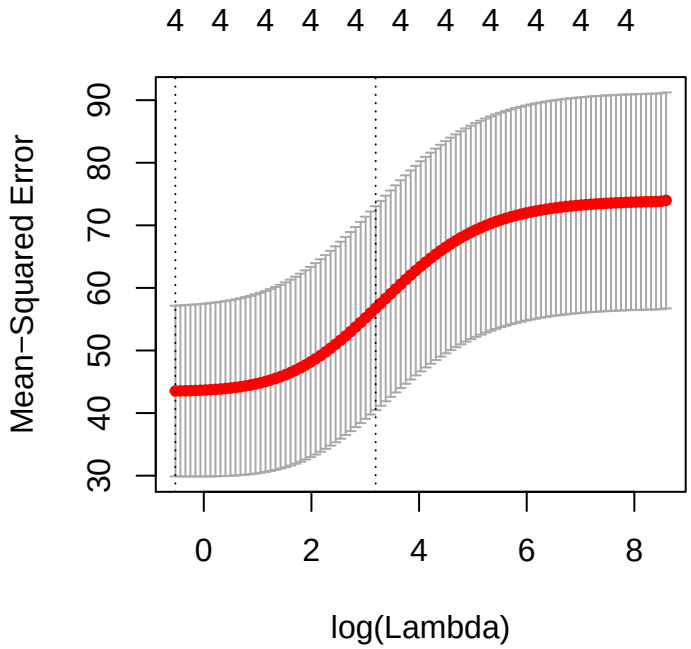

```
print(paste("El mejor lambda es:",best_lambda))
```

```
## [1] "El mejor lambda es: 0.589904662214765"
```

Visualmente es lógico que haya underfitting, pues sólo tenemos cuatro variables y ninguna supera el umbral. Entonces, las predicciones serán a ciegas (aleatorias) ya que no tenemos nada que explique la variable de salida.

Veamos evolución de los grados de libertad del modelo y el porcentaje de desviación explicados según los distintos valores de λ.

```
plot(cvfit)
```



```r
glmnet(x,y,family="gaussian",alpha=0)
```

```
##
## Call:  glmnet(x = x, y = y, family = "gaussian", alpha = 0)
##
##        Df      %Dev    Lambda
##   [1,]  4 1.388e-36 5375.0000
##   [2,]  4 2.399e-03 4897.0000
##   [3,]  4 2.632e-03 4462.0000
##   [4,]  4 2.887e-03 4066.0000
##   [5,]  4 3.166e-03 3705.0000
##   [6,]  4 3.473e-03 3376.0000
##   [7,]  4 3.808e-03 3076.0000
##   [8,]  4 4.176e-03 2803.0000
##   [9,]  4 4.580e-03 2554.0000
##  [10,]  4 5.022e-03 2327.0000
##  [11,]  4 5.505e-03 2120.0000
##  [12,]  4 6.035e-03 1932.0000
##  [13,]  4 6.616e-03 1760.0000
##  [14,]  4 7.251e-03 1604.0000
##  [15,]  4 7.946e-03 1461.0000
##  [16,]  4 8.706e-03 1331.0000
##  [17,]  4 9.538e-03 1213.0000
##  [18,]  4 1.045e-02 1105.0000
##  [19,]  4 1.144e-02 1007.0000
##  [20,]  4 1.253e-02  917.7000
##  [21,]  4 1.371e-02  836.2000
##  [22,]  4 1.501e-02  761.9000
##  [23,]  4 1.642e-02  694.2000
##  [24,]  4 1.796e-02  632.5000
##  [25,]  4 1.964e-02  576.3000
##  [26,]  4 2.147e-02  525.1000
##  [27,]  4 2.346e-02  478.5000
##  [28,]  4 2.562e-02  436.0000
##  [29,]  4 2.797e-02  397.3000
##  [30,]  4 3.052e-02  362.0000
##  [31,]  4 3.329e-02  329.8000
##  [32,]  4 3.628e-02  300.5000
##  [33,]  4 3.952e-02  273.8000
##  [34,]  4 4.302e-02  249.5000
##  [35,]  4 4.680e-02  227.3000
##  [36,]  4 5.087e-02  207.1000
##  [37,]  4 5.525e-02  188.7000
##  [38,]  4 5.995e-02  172.0000
##  [39,]  4 6.499e-02  156.7000
##  [40,]  4 7.038e-02  142.8000
##  [41,]  4 7.612e-02  130.1000
##  [42,]  4 8.224e-02  118.5000
##  [43,]  4 8.874e-02  108.0000
##  [44,]  4 9.563e-02   98.4000
##  [45,]  4 1.029e-01   89.6600
##  [46,]  4 1.106e-01   81.7000
##  [47,]  4 1.186e-01   74.4400
```

```
##  [48,]  4 1.270e-01   67.8200
##  [49,]  4 1.358e-01   61.8000
##  [50,]  4 1.449e-01   56.3100
##  [51,]  4 1.543e-01   51.3100
##  [52,]  4 1.641e-01   46.7500
##  [53,]  4 1.740e-01   42.6000
##  [54,]  4 1.843e-01   38.8100
##  [55,]  4 1.947e-01   35.3600
##  [56,]  4 2.052e-01   32.2200
##  [57,]  4 2.158e-01   29.3600
##  [58,]  4 2.265e-01   26.7500
##  [59,]  4 2.372e-01   24.3700
##  [60,]  4 2.479e-01   22.2100
##  [61,]  4 2.584e-01   20.2400
##  [62,]  4 2.688e-01   18.4400
##  [63,]  4 2.790e-01   16.8000
##  [64,]  4 2.890e-01   15.3100
##  [65,]  4 2.987e-01   13.9500
##  [66,]  4 3.081e-01   12.7100
##  [67,]  4 3.172e-01   11.5800
##  [68,]  4 3.259e-01   10.5500
##  [69,]  4 3.341e-01    9.6140
##  [70,]  4 3.420e-01    8.7600
##  [71,]  4 3.495e-01    7.9820
##  [72,]  4 3.565e-01    7.2730
##  [73,]  4 3.630e-01    6.6270
##  [74,]  4 3.691e-01    6.0380
##  [75,]  4 3.748e-01    5.5010
##  [76,]  4 3.800e-01    5.0130
##  [77,]  4 3.849e-01    4.5670
##  [78,]  4 3.892e-01    4.1620
##  [79,]  4 3.933e-01    3.7920
##  [80,]  4 3.969e-01    3.4550
##  [81,]  4 4.001e-01    3.1480
##  [82,]  4 4.031e-01    2.8680
##  [83,]  4 4.057e-01    2.6140
##  [84,]  4 4.080e-01    2.3810
##  [85,]  4 4.101e-01    2.1700
##  [86,]  4 4.119e-01    1.9770
##  [87,]  4 4.135e-01    1.8010
##  [88,]  4 4.149e-01    1.6410
##  [89,]  4 4.162e-01    1.4960
##  [90,]  4 4.172e-01    1.3630
##  [91,]  4 4.181e-01    1.2420
##  [92,]  4 4.189e-01    1.1310
##  [93,]  4 4.196e-01    1.0310
##  [94,]  4 4.202e-01    0.9393
##  [95,]  4 4.207e-01    0.8559
##  [96,]  4 4.212e-01    0.7798
##  [97,]  4 4.215e-01    0.7105
##  [98,]  4 4.218e-01    0.6474
##  [99,]  4 4.221e-01    0.5899
## [100,]  4 4.223e-01    0.5375
```

Vamos a interpretar lo que tenemos aqui. Tenemos tres columnas, donde la primera representa los grados de libertad (Df) del modelo. La segunda columna es el tanto por ciento de desviación de los residuos que son explicados y la última columna es el valor de λ. De abajo a arriba, cuando λ es incrementada, los grados de libertad deberían bajar, aproximandose a un modelo nulo. Del mismo modo, según sube λ, el tanto por ciento de desviación de los residuos explicada va a 0. La última columna debería ser aquélla en el que se explicara el mayor porcentaje de desviación y los grados de libertad deberían ser los máximos. Vemos cómo no sólo los grados de libertad siempre son los mismo sea cual sea el valor de λ, sino que además, para el mínimo valor de λ, el porcentaje de desviación explicado es muy próximo a cero. Esto es un claro síntoma de subajuste o "underfitting". En el gráfico vemos cómo nos dice que el menor valor de lambda es aquél que minimiza el error, pero está por encima del 40%, por lo que, casi estamos ante un estimador aleatorio.

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.

Vamos a preparar el dataset para este apartado. Reseteamos, creamos la nueva variable, la sustituimos y partimos en entrenamiento y test.

```
#Introducir la variable
complet_Boston<-Boston
crim01<-complet_Boston$crim
crim01[crim01>=median(complet_Boston$crim)]<-1
crim01[crim01<median(complet_Boston$crim)]<--1
complet_Boston<-cbind(crim01,complet_Boston[,-1])

#Barajar
orden<-runif(nrow(complet_Boston))
complet_Boston<-complet_Boston[order(orden),]
#Separación de entrenamiento y test
porcentaje_training<-0.8
nEjemplosTrain<-trunc(nrow(complet_Boston)*porcentaje_training)
i_training<-1:nEjemplosTrain
i_test<-(nEjemplosTrain+1):nrow(complet_Boston)



conj_training<-complet_Boston[i_training,]
conj_test<-complet_Boston[i_test,]
```

Una vez hecho, procedemos a crear el modelo de regresión lineal. Dado que regresión no devuelve clases (-1 ó 1) habrá que transformar cada predicción a la clase a la que más se acerque (por ejemplo una predicción de 0.3 será catalogado como 1 en lugar de -1), teniendo así predicciones clasificatorias.

```
attach(complet_Boston)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     crim01
```

```
mod_lineal <- lm(crim01~
                 zn+rm+rad+lstat,
                 data=conj_training)
pred_lin<-predict(mod_lineal,newdata=conj_test)
pred_lin[pred_lin>0]<-1
pred_lin[pred_lin<0]<--1
error_lin<-length(pred_lin[pred_lin!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test en el modelo lineal: ", error_lin))
```

```
## [1] "Error de test en el modelo lineal:  0.156862745098039"
```

Este es el error del modelo lineal simple, vamos a intentar mejorarlo con SVM. Probemos con los distintos tipos de kernel que trae SVM: lineal, polinómico, de base radial y sigmoidal:

Kernel Lineal:

```
set.seed(199)
mod_svm <- svm(crim01 ~
             zn+rm+rad+lstat,
             data=conj_training, type="C-classification",kernel="linear")

pred_svm <- predict(mod_svm, conj_test)
error_svm<-length(pred_svm[pred_svm!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test svm kernel lineal: ", error_svm))
```

```
## [1] "Error de test svm kernel lineal:  0.166666666666667"
```

No mejora el error del modelo lineal, descartado.

Kernel polinómico:

```
set.seed(199)
mod_svm <- svm(crim01 ~
             zn+rm+rad+lstat,
             data=conj_training, type="C-classification",kernel="polynomial")

pred_svm <- predict(mod_svm, conj_test)
error_svm<-length(pred_svm[pred_svm!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test svm kernel polinomico: ", error_svm))
```

```
## [1] "Error de test svm kernel polinomico:  0.156862745098039"
```

Es mejor que el anterior, pero el error es exactamente el mismo que en el modelo lineal.

Kernel de base radial:

```
set.seed(199)
mod_svm <- svm(crim01 ~
             zn+rm+rad+lstat,
             data=conj_training, type="C-classification",kernel="radial")

pred_svm <- predict(mod_svm, conj_test)
error_svm<-length(pred_svm[pred_svm!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test svm kernel base radial: ", error_svm))
```

```
## [1] "Error de test svm kernel base radial:  0.147058823529412"
```

Mejora a todos los anteriores, incluso al modelo lineal simple.

Kernel sigmoidal:

```
set.seed(199)
mod_svm <- svm(crim01 ~
             zn+rm+rad+lstat,
             data=conj_training, type="C-classification",kernel="sigmoid")

pred_svm <- predict(mod_svm, conj_test)
error_svm<-length(pred_svm[pred_svm!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test svm kernel sigmoidal: ", error_svm))
```

```
## [1] "Error de test svm kernel sigmoidal:  0.254901960784314"
```

El error se dispara, queda descartado.

Vemos cómo el mejor de los kernels es el de base radial, que también mejora al modelo lineal. Sabiendo esto, vamos a obtener los parámetros necesarios para ajustar el modelo óptimo de svm que se pueda con `tune`:

```
tuneResult <- tune(
    svm, crim01 ~ zn+rm+rad+lstat,
    data = conj_training
    )
tuneResult$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = crim01 ~ zn + rm + rad + lstat,
##     data = conj_training)
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.25
##     epsilon:  0.1
##
##
## Number of Support Vectors:  217
```

El optimizador nos sugiere que utilicemos epsilon-regresión, por lo que, para transformar a un problema de clasificación habrá que elegir la clase que esté más cerca de la predicción. Hecho esto, podemos calcular el error como si de un problema de clasificación se tratase.

Ajustamos el modelo con estos parámetros:

```
mod_svm <- svm(crim01 ~
             zn+rm+rad+lstat,
             data=conj_training,kernel="radial", type="eps-regression",
```

```
            epsilon=0.1,cost=1,gamma=0.25)

pred_svm <- predict(mod_svm, conj_test)
pred_svm[pred_svm>0]<-1
pred_svm[pred_svm<0]<--1
error_svm<-length(pred_svm[pred_svm!=conj_test[,1]])/nrow(conj_test)
print(paste("Error de test svm kernel base radial: ", error_svm))
```

```
## [1] "Error de test svm kernel base radial:  0.127450980392157"
```

Usando kernel de base radial optimizado, hemos conseguido un error menor que el del modelo lineal, mejorando alrededor de un 3%.

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

La función `svm` nos lo pone muy sencillo con el parámetro `cross`. Ajustaremos de nuevo el modelo con este parámetro con valor 5. Con ello obtendremos el valor de error de entrenamiento. Para obtener el de test haremos un bucle como lo hemos venido haciendo hasta ahora.

```
#VALIDACION CRUZADA TEST
nparticiones<-5
parts <- cut(seq(1,nrow(complet_Boston)),breaks=nparticiones,labels=FALSE)
errors_cv_svm<-numeric(nparticiones)
for(i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(complet_Boston)),testIndexes)

  conj_test_cv<-complet_Boston[testIndexes,]
  conj_training_cv<-complet_Boston[trainIndexes,]

  mod_svm_cv <- svm(crim01 ~
              zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat+medv,
              data=conj_training_cv,kernel="radial", type="eps-regression",
              gamma=0.07692308)

  pred_svm <- predict(mod_svm_cv, conj_test_cv)
  errors_cv_svm[i]<-mean((pred_svm-conj_test_cv[,1])*(pred_svm-conj_test_cv[,1]))
}

#VALIDACION CRUZADA ENTRENAMIENTO
mod_svm_cv <- svm(crim01 ~
              zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat+medv,
              data=conj_training,kernel="radial", type="eps-regression",
              gamma=0.07692308,cross=5)

print(paste("Error en entrenamiento 5cv svm kernel radial:",mean(mod_svm_cv$MSE)))
```

```
## [1] "Error en entrenamiento 5cv svm kernel radial: 0.360286035457305"
```

```
print(paste("Error en test 5cv svm kernel radial:",mean(errors_cv_svm)))
```

```
## [1] "Error en test 5cv svm kernel radial: 0.338665485327632"
```

Los errores de entrenamiento y test por validación cruzada son similares, lo que no debería ser alarmante.

Ejercicio 3

Usar el conjunto de datos Boston y las librerías randomForest y gbm de R.

Vamos a instalar y usar las bibliotecas para comenzar con el ejercicio.

```
install.packages("randomForest")
install.packages("gbm")
install.packages("ipred")
library(randomForest)
library(gbm)
```

```
## randomForest 4.6-12

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

## Loading required package: survival

## Loading required package: splines

## Loading required package: lattice

## Loading required package: parallel

## Loaded gbm 2.1.1
```

1. Dividir la base de datos en dos conjuntos de entrenamiento (80%) y test (20%).

Lo hacemos como hemos estado haciendo hasta ahora:

```
set.seed(199)
complet_Boston<-Boston
#Barajar
orden<-runif(nrow(complet_Boston))
complet_Boston<-complet_Boston[order(orden),]

#Separación de entrenamiento y test
porcentaje_training<-0.8
nEjemplosTrain<-trunc(nrow(complet_Boston)*porcentaje_training)
i_training<-1:nEjemplosTrain
i_test<-(nEjemplosTrain+1):nrow(complet_Boston)

#Asignación a entrenamiento/test
conj_training<-complet_Boston[i_training,]
conj_test<-complet_Boston[i_test,]
attach(complet_Boston)
```

```
## The following objects are masked from complet_Boston (position 9):
##
##     age, black, chas, dis, indus, lstat, medv, nox, ptratio, rad,
##     rm, tax, zn
```

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.

Random Forest es un caso particular de bagging. La única diferencia es el número de variables predictoras elegidas para cada árbol. Sea N el número de casos de prueba y P el número de variables predictoras, se eligen m variables predictoras para crear cada árbol. Si m=P, es decir, se escogen todas las variables en cada árbol, estamos ante bagging. Repetiremos el experimento 10 veces con distintos números de árboles entre 100 y 1000 para estimar el mejor número de árboles. El número de árboles será evaluado de 100 en 100.

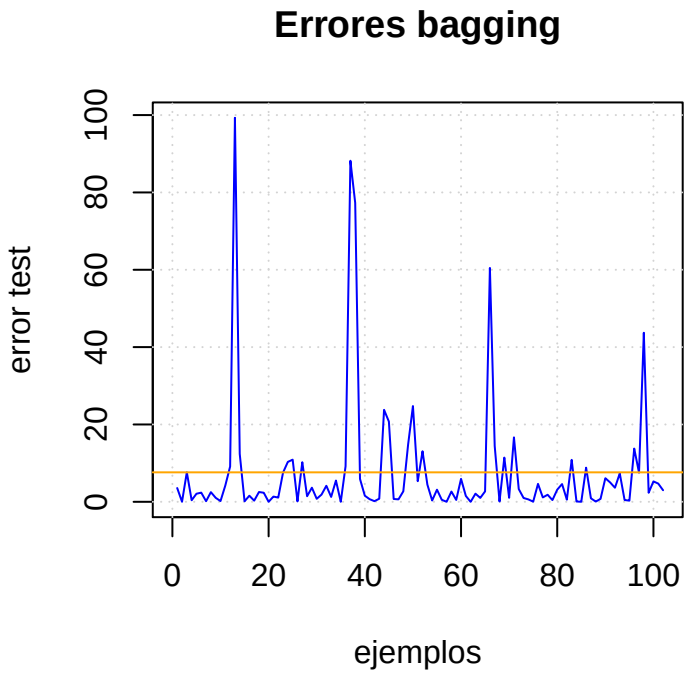
```
narboles_bagg<-numeric(10)
errores_narboles_bagg<-numeric(10)
qerrors_bagg<-list(10)

for(i in 1:10){
mod_bagg <- randomForest(medv ~ crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat,
                    data=conj_training, importance=TRUE, mrty=ncol(conj_training)-1,ntree=100*i)
pred_bagg <- predict(mod_bagg,conj_test)
qerrors_bagg[[i]]<-(pred_bagg-conj_test[,ncol(conj_test)])*(pred_bagg-conj_test[,ncol(conj_test)])
errores_narboles_bagg[i]<-mean(qerrors_bagg[[i]])
narboles_bagg[i]<-mod_bagg$ntree
}

best_index_bagg=which(errores_narboles_bagg==min(errores_narboles_bagg))
print(paste("Mejor error test bagging: ",errores_narboles_bagg[best_index_bagg]))
```

```
## [1] "Mejor error test bagging:  7.61991371541827"
```

```
print(paste("Mejor numero de arboles: ",narboles_bagg[best_index_bagg]))
```

```
## [1] "Mejor numero de arboles:  100"
```

```
error_test_bagg<-mean(errores_narboles_bagg[best_index_bagg])
plot(NA,
     xlim=c(0,length(qerrors_bagg[[best_index_bagg]])),
     ylim=c(min(qerrors_bagg[[best_index_bagg]]),max(qerrors_bagg[[best_index_bagg]])),
     main="Errores bagging",
     xlab="ejemplos",
     ylab="error test")
grid()
lines(1:length(qerrors_bagg[[best_index_bagg]]),qerrors_bagg[[best_index_bagg]],col="blue")
abline(h=error_test_bagg,col="orange")
```



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.

Para ajustar un modelo de random forest puro, el número de variables a seleccionar en cada árbol será un tercio de las variables predictoras, ya que estamos en un problema de regresión. Si estuviéramos en clasificación, lo mejor es elegir $\sqrt{M}$ variables para cada árbol. Al igual que en el apartado anterior, estimaremos el mejor número de árboles repitiendo el experimento 10 veces con un número distinto de árboles.

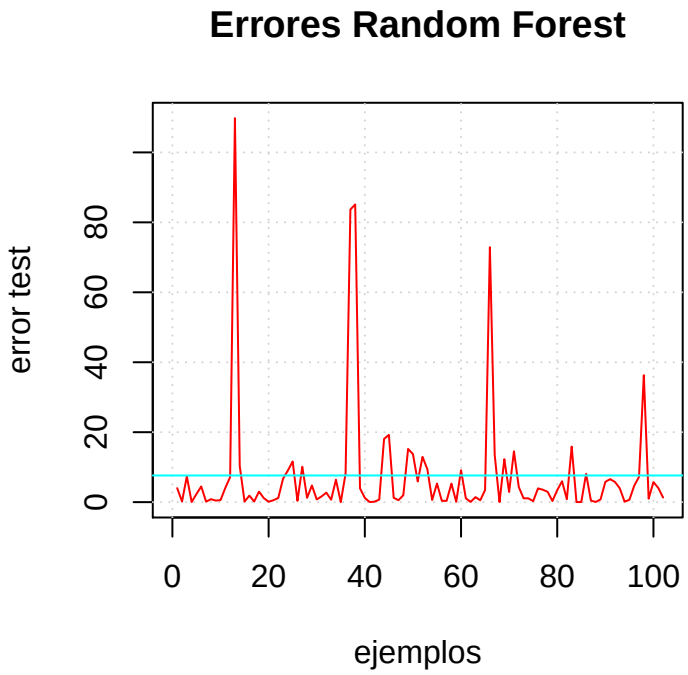
```
narboles_rf<-numeric(10)
errores_narboles_rf<-numeric(10)
qerrors_rf<-list(10)

for(i in 1:10){
mod_rf <- randomForest(medv ~ crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat,
                    data=conj_training, importance=TRUE, mrty=(ncol(conj_training)-1)/3)

pred_rf <- predict(mod_rf,conj_test)

qerrors_rf[[i]]<-(pred_rf-conj_test[,ncol(conj_test)])*(pred_rf-conj_test[,ncol(conj_test)])
errores_narboles_rf[i]<-mean(qerrors_rf[[i]])
narboles_rf[i]<-mod_rf$ntree
}

best_index_rf=which(errores_narboles_rf==min(errores_narboles_rf))
print(paste("Mejor error test rf: ",errores_narboles_rf[best_index_rf]))
```

```
## [1] "Mejor error test rf:  7.6224876665737"
```

```
print(paste("Mejor numero de arboles rf: ",narboles_rf[best_index_rf]))
```

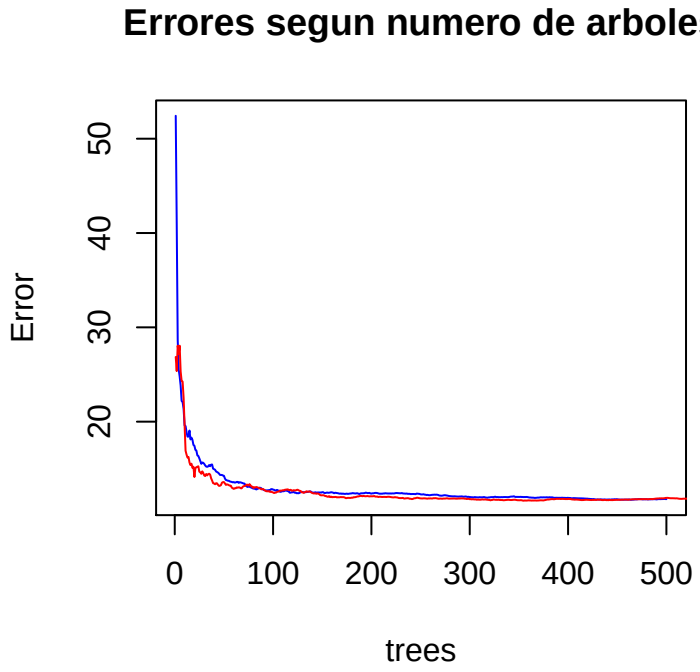
```
## [1] "Mejor numero de arboles rf:  500"
```

```
error_test_rf<-mean(errores_narboles_rf[best_index_rf])

plot(NA,
     xlim=c(0,length(qerrors_rf[[best_index_rf]])),
     ylim=c(min(qerrors_rf[[best_index_rf]]),max(qerrors_rf[[best_index_rf]])),
     main="Errores Random Forest",
     xlab="ejemplos",
     ylab="error test")
grid()
lines(1:length(qerrors_rf[[best_index_rf]]),qerrors_rf[[best_index_rf]],col="red")
abline(h=error_test_rf,col="cyan")
```



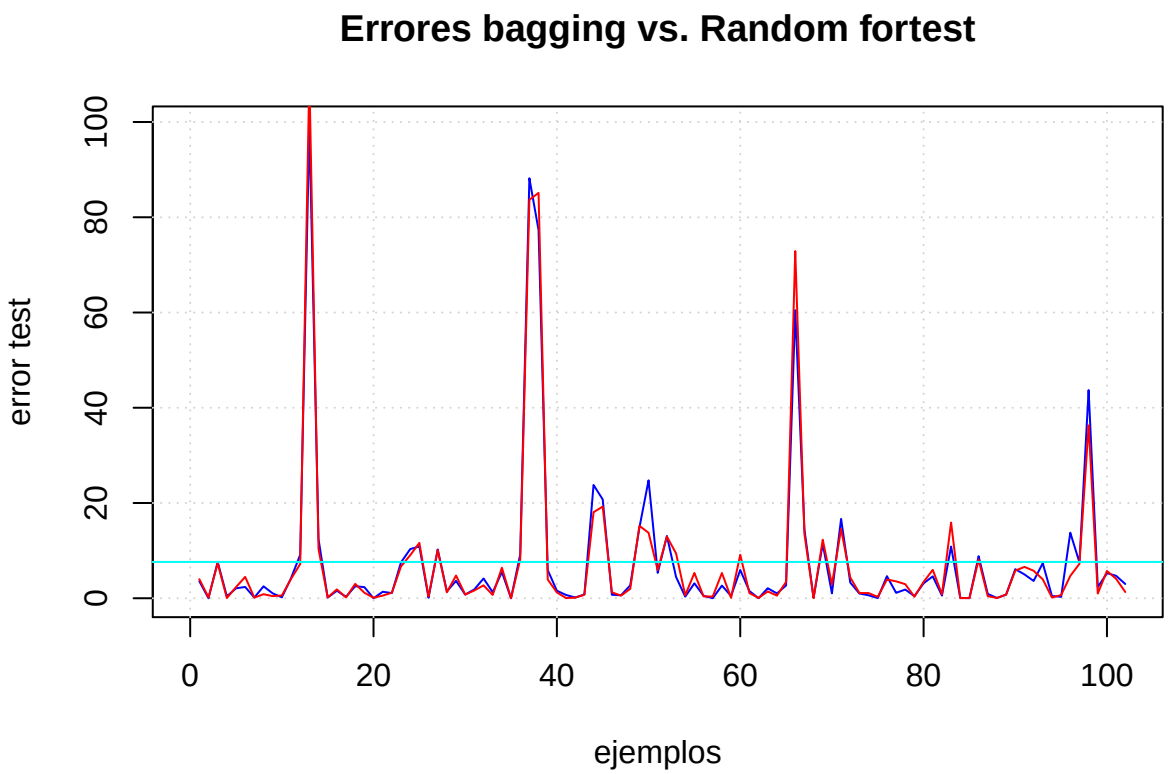
Veamos de forma general el comportamiento de los modelos bagging y random forest. En rojo, random forest, en azul, bagging:

```
plot(mod_rf,col="blue", main="Errores segun numero de arboles")
plot(mod_bagg,add=T,col="red")
```



Comparemos los errores cometidos por ambos modelos sobre un mismo gráfico:

```
plot(NA,
     xlim=c(0,length(qerrors_bagg[[best_index_bagg]])),
     ylim=c(min(qerrors_bagg[[best_index_bagg]]),max(qerrors_bagg[[best_index_bagg]])),
     main="Errores bagging vs. Random fortest",
     xlab="ejemplos",
     ylab="error test")
grid()
lines(1:length(qerrors_bagg[[best_index_bagg]]),qerrors_bagg[[best_index_bagg]],col="blue")
abline(h=error_test_bagg,col="orange")

lines(1:length(qerrors_rf[[best_index_rf]]),qerrors_rf[[best_index_rf]],col="red")
abline(h=error_test_rf,col="cyan")
```



Vemos como ambos modelos son muy parecidos, y las diferencias no son sustanciales.

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.

Para igualar las condiciones, vamos a limitar el número de árboles a 500, que era el valor estimado de bagging y random forest. Además ajustaremos el parámetro de profundidad de los árboles a un valor $2 \leq d \leq 3$ y el parámetro de amortiguación $\lambda = 0.01$. Elegiremos el mejor número de árboles repitiendo el experimento 10 veces con distintos números de árboles.

```
narboles_boost<-numeric(10)
errores_narboles_boost<-numeric(10)
qerrors_boost<-list(10)

for(i in 1:10){

mod_boost <- gbm(medv ~
             crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat,
             data=conj_training,
             distribution = 'gaussian',
             n.trees = 100*i,
             shrinkage = 0.01,
             interaction.depth = 3)

pred_boost<-predict(mod_boost,conj_test,n.trees=100*i)
qerrors_boost[[i]]<-(pred_boost-conj_test[,ncol(conj_test)])*(pred_boost-conj_test[,ncol(conj_test)])
errores_narboles_boost[i]<-mean(qerrors_boost[[i]])
narboles_boost[i]<-(100*i)
}
```

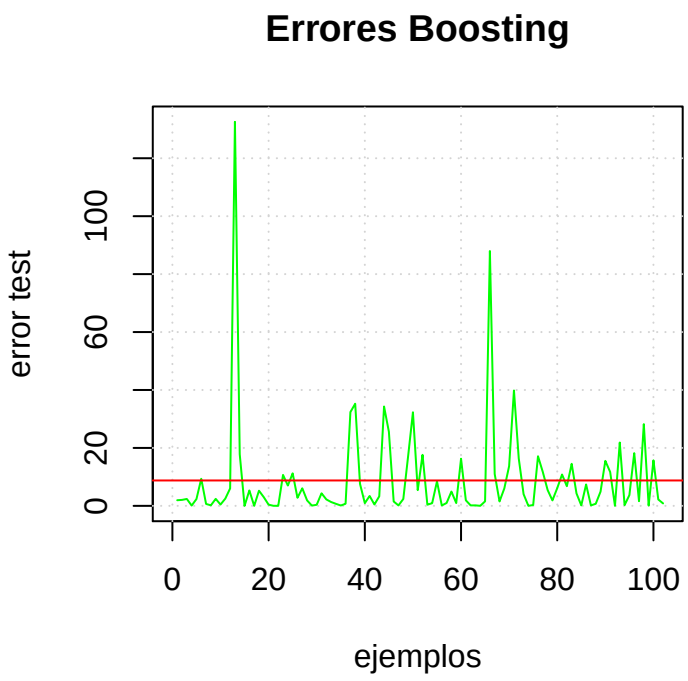
```
best_index_boost=which(errores_narboles_boost==min(errores_narboles_boost))
print(paste("Mejor error test boosting: ",errores_narboles_boost[best_index_boost]))
```

```
## [1] "Mejor error test boosting:  8.76290773191823"
```

```
print(paste("Mejor numero de arboles boosting: ",narboles_boost[best_index_boost]))
```

```
## [1] "Mejor numero de arboles boosting:  900"
```

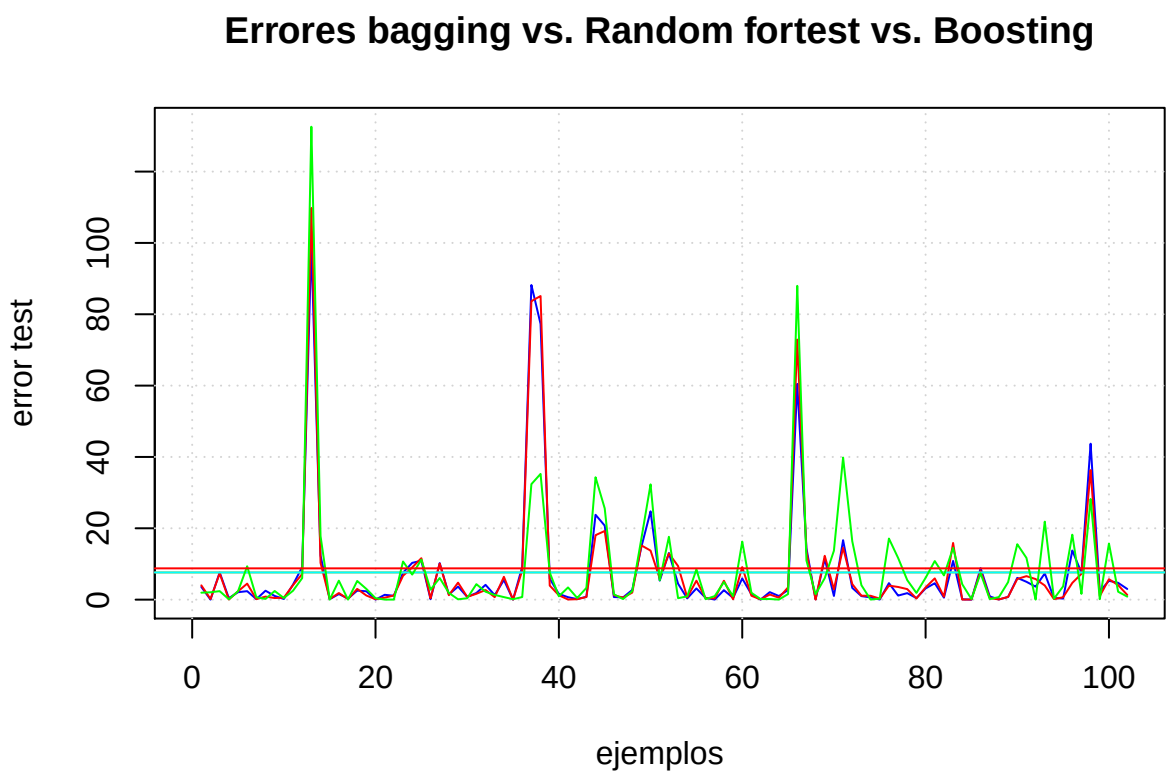
```
error_test_boost<-mean(errores_narboles_boost[best_index_boost])

plot(NA,
     xlim=c(0,length(qerrors_boost[[best_index_boost]])),
     ylim=c(min(qerrors_boost[[best_index_boost]]),max(qerrors_boost[[best_index_boost]])),
     main="Errores Boosting",
     xlab="ejemplos",
     ylab="error test")
grid()
lines(1:length(qerrors_boost[[best_index_boost]]),qerrors_boost[[best_index_boost]],col="green")
abline(h=error_test_boost,col="red")
```



Comparemos todos los errores en un solo gráfico.

```
plot(NA,
     xlim=c(0,length(qerrors_bagg[[best_index_bagg]])),
     ylim=c(min(
               qerrors_bagg[[best_index_bagg]],
               qerrors_rf[[best_index_rf]],
               qerrors_boost[[best_index_boost]]
```

```
          ),max(
              qerrors_bagg[[best_index_bagg]],
              qerrors_rf[[best_index_rf]],
              qerrors_boost[[best_index_boost]]
          )),
     main="Errores bagging vs. Random fortest vs. Boosting",
     xlab="ejemplos",
     ylab="error test")
grid()
lines(1:length(qerrors_bagg[[best_index_bagg]]),qerrors_bagg[[best_index_bagg]],col="blue")
abline(h=error_test_bagg,col="orange")

lines(1:length(qerrors_rf[[best_index_rf]]),qerrors_rf[[best_index_rf]],col="red")
abline(h=error_test_rf,col="cyan")

lines(1:length(qerrors_boost[[best_index_boost]]),qerrors_boost[[best_index_boost]],col="green")
abline(h=error_test_boost,col="red")
```



Obteniendo el mejor número de árboles para cada modelo, vemos cómo el error de todos los modelos es similar. No obstante, el número de árboles habla a favor de bagging, pues obtiene un error similar al de sus competidores con el menor número de árboles, por ello, en este caso me decantaría por bagging.

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).

Instalamos y usamos la biblioteca tree.

```
install.packages("tree")
library(tree)
```

Vamos a partir el dataset y crear el modelo de árbol.

```
set.seed(199)
complet_OJ<-OJ
orden<-runif(nrow(complet_OJ))
complet_OJ<-complet_OJ[order(orden),]

conj_training<-complet_OJ[1:800,]
conj_test<-complet_OJ[801:length(complet_OJ),]

attach(complet_OJ)

arbol<-tree(Purchase~., data = conj_training)
```

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.

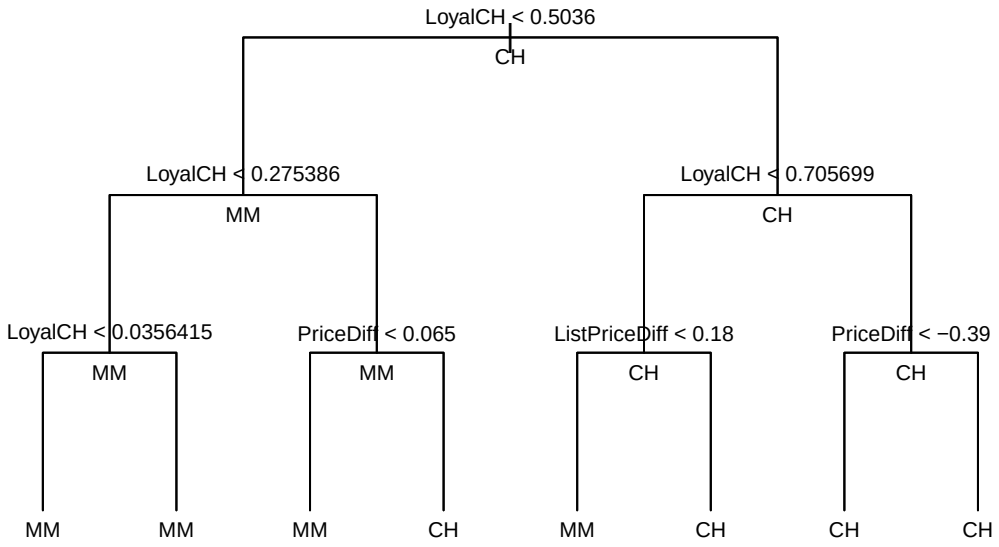
```
summary(arbol)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = conj_training)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7275 = 576.1 / 792
## Misclassification error rate: 0.1612 = 129 / 800
```

Vemos cómo el árbol obtenido sólo ha tenido en cuenta las variables "LoyalCH", "PriceDiff" y "ListPriceDiff", que parecen suficientes para separar todos los ejemplos en clases inequívocas. El error en training está sobre el 16%, lo cual es un tanto elevado en comparación con otros métodos vistos. El número de nodos terminales es 8, es decir, hay 8 grupos de ejemplos que pertenecen a una sola clase al final del árbol.

3. Crear un dibujo del árbol e interpretar los resultados

```
plot(arbol, type="uniform")
text(arbol, all=TRUE, cex=0.7)
```



Expliquemos qué indica el árbol:

- La variable que más separa es "LoyalCH". Si el valor de "LoyalCH" está entre 0.275386 y 0.705699 estamos ante un ejemplo de la clase "Purchase = CH".
  - Si "LoyalCH" es menor que 0.275386 estamos ante un ejemplo de la clase "Purchase = MM". Si no, hay que preguntar por la variable "PriceDiff".
    - Si "PriceDiff" es menor que 0.065, la clase es "Purchase = MM", si no, la clase es "Purchase = CH".
  - Si "LoyalCH" es mayor que 0.705699, hay que preguntar por la variable "ListPriceDiff" o "PriceDiff".
    - Si "ListPriceDiff" es menor que 0.18 entonces estamos ante un ejemplo de la clase "Purchase = CH" en caso contrario la clase es "Purchase = MM".
    - Si "PriceDiff" es menor que -0.39 estamos ante un ejemplo de la clase "Purchase = CH". En realidad, ocurra lo que ocurra aquí, la clase sería "Purchase = CH".

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?

```
set.seed(199)
pred_tree<-predict(arbol,conj_test,type="class")
mc_tree <- table(pred_tree, conj_test[,1])
mc_tree
```

```
##
## pred_tree  CH  MM
##        CH 430  80
##        MM  46 228
```

La matriz de confusión nos está diciendo que hay 414 aciertos de la clase "Purchase = CH" y 249 aciertos de la clase "Purchase = MM". Hay 67 ejemplos de los cuales se predijo que eran "Purchase = MM" y en realidad eran de la clase "Purchase = CH". Del mismo modo hay 54 ejemplos que se predijeron "Purchase = CH" y en realidad eran de la clase "Purchase = MM".

En total vemos que se ha acertado en $414 + 249 = 663$ ejemplos, mientras que se ha fallado en $67 + 54 = 121$. Calculamos el porcentaje de error con la matriz de confusión:

```
print(paste("Error de test modelo arbol:",1-(sum(diag(mc_tree))/sum(mc_tree))))
```

```
## [1] "Error de test modelo arbol: 0.160714285714286"
```

El error resultante es un poco elevado.

La precisión para cada clase se calcula como los aciertos de la clase entre la suma de la fila predicha. Entonces:

```
print(paste("Precision para CH:",mc_tree[1,1]/sum(mc_tree[1,])))
```

```
## [1] "Precision para CH: 0.843137254901961"
```

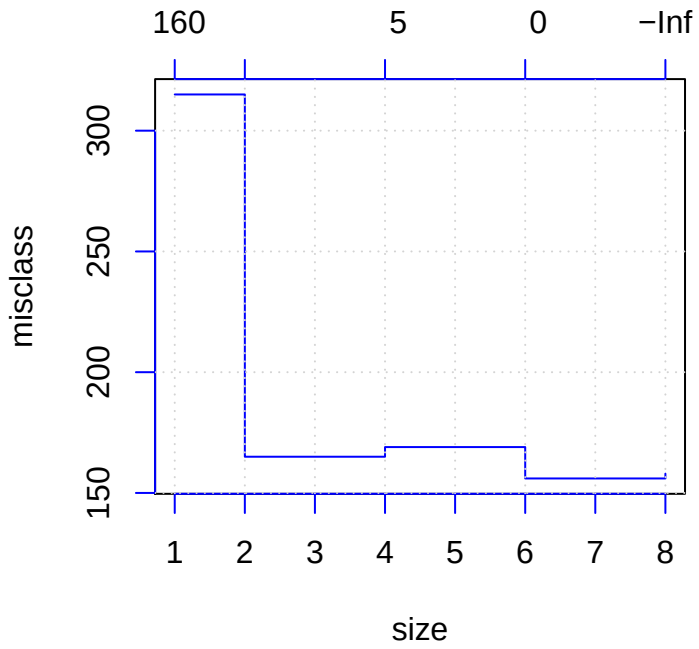
```
print(paste("Precision para MM:",mc_tree[2,2]/sum(mc_tree[2,])))
```

```
## [1] "Precision para MM: 0.832116788321168"
```

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

La función `cv.tree()` aplica validación cruzada a un modelo de arbol, para estimar su error para distintos tamaños de árbol. Con tamaño hago referencia al número de nodos. El método de medida de error que usaremos es el número de puntos mal clasificados, ya que estamos ante un problema de clasificación.

```
set.seed(199)
ajuste_cv<-cv.tree(arbol,K=10,method="misclass")
plot(ajuste_cv, col="blue")
grid()
```




```
ajuste_cv$size
```

```
## [1] 8 6 4 2 1
```

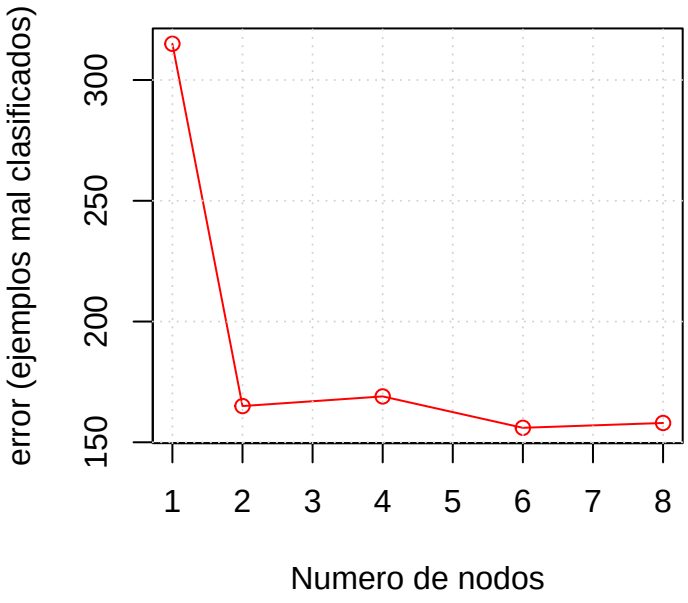
```
ajuste_cv$dev
```

```
## [1] 158 156 169 165 315
```

Según el gráfico, vemos cómo el tamaño óptimo del árbol es 6, pues es el que reduce el número de puntos mal clasificados, cerca de 150, lo que supondría alrededor del 18% de error, que sigue siendo elevado. Además si imprimimos los diferentes tamaños de árbol con sus respectivos errores, vemos que el más pequeño es 6, como dicta el gráfico.

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?

```
error<-as.matrix(c(ajuste_cv$dev))

plot(ajuste_cv$size,error,
     xlab="Numero de nodos",
     ylab="error (ejemplos mal clasificados)",
     type="o",
     col="red")
grid()
```




```
min_error_nnodos<-ajuste_cv$size[which(error==min(error))]

print(paste("El numero de nodos para el que se minimiza el error es: ",min_error_nnodos))
```

```
## [1] "El numero de nodos para el que se minimiza el error es:  6"
```

Como vemos, obtenemos el mismo resultado que ya sabíamos con el gráfico proporcionado por `cv.tree()` en el apartado anterior. Ahora podemos dibujar el árbol con el número óptimo de nodos terminales que ya sabemos, que es 6.

```
set.seed(199)
parbol<-prune.tree(arbol,best = min_error_nnodos)
plot(parbol, type="uniform")
text(parbol, all=TRUE, cex=0.7)
```