Trabajo 3

Luis Suárez Lloréns

4 de junio de 2016

Apartado 1

Antes de empezar a responder a los diferentes puntos del apartado, debemos cargar los datos de la base de datos *Auto*. Estos se encuentran en la librería *ISLR* de R. Cargamos la librería con la siguiente orden.

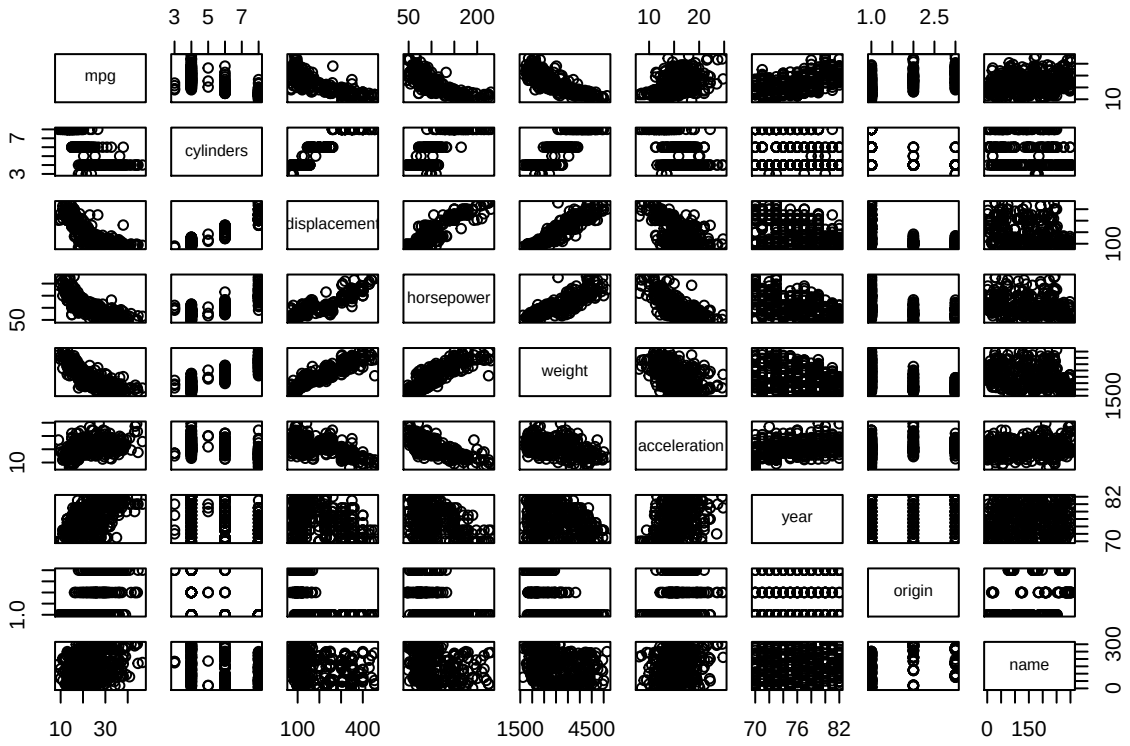
```
library(ISLR)
```

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

a)

Vamos a usar la función *pairs* para tener una visión general de los datos.

```
pairs(Auto)
```

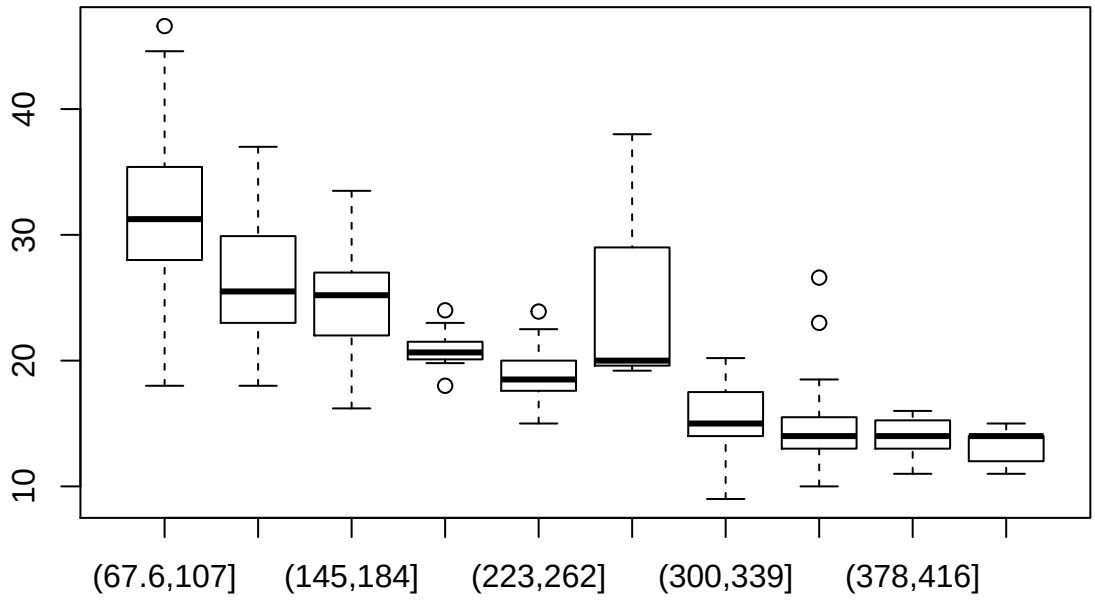


Nos fijamos en la fila de mpg, que es la característica que queremos clasificar. Viendo las gráficas de la fila de mpg, podemos ver que hay 3 que muestran cierta tendencia, que son *displacement*, *horsepower* y *weight*. Las demás muestran nubes de puntos demasiado amplias y difuminadas, que no pueden ser ajustadas.

Vamos a usar la función *boxplot* para representar estas tres gráficas.

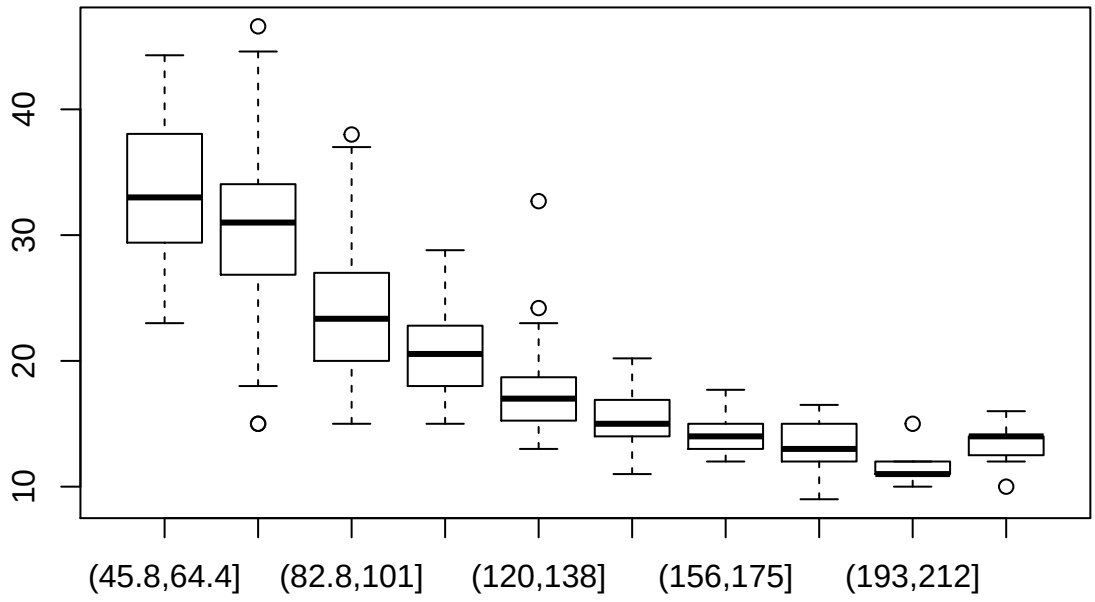
DISPLACEMENT

```
boxplot(mpg~cut(displacement, breaks = 10),data = Auto)
```



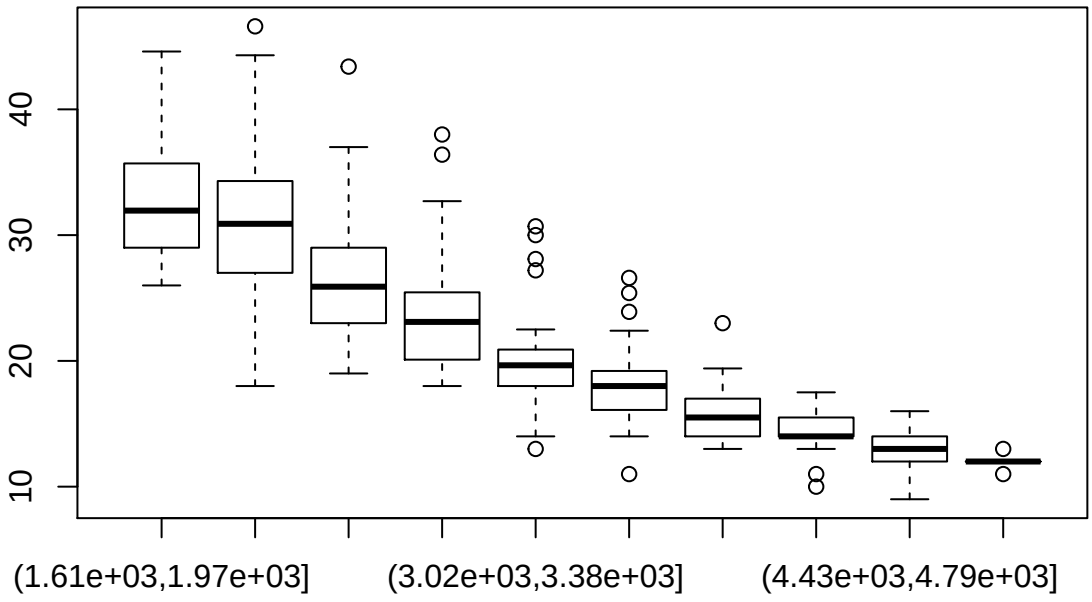
HORSEPOWER

```
boxplot(mpg~cut(horsepower, breaks = 10),data = Auto)
```



WEIGHT

```
boxplot(mpg~cut(weight, breaks = 10),data = Auto)
```



De las anteriores, el dato *displacement* sufre un repunte, en el intervalo (262,300]. Si nos fijamos realmente en los datos, esta sección tiene muy pocos datos, es normal que se pueda ver alterado. Salvo esto, las 3 muestran una tendencia clara. Si hubiera que elegir entre una de las tres, probablemente cogería *horsepower*, pues *weight* tiene muchos outliners y *displacement* tiene ese comportamiento un poco extraño comentado antes.

b)

Por lo dicho anteriormente, seleccionaremos como posibles variables predictoras *displacement*, *horsepower* y *weight*, pues las nubes de puntos parecen reflejar una tendencia clara.

c)

Para separar los datos en entrenamiento y test, vamos a usar un muestreo sin remplazamiento. En nuestro caso, como tenemos muchos datos, el comportamiento de este muestreo aleatorio debería cubrir bien todos los casos. En el caso de que no se tuvieran buenos resultados, sería necesario hacer este muestreo de manera estratificada.

```
idx.train = sample(nrow(Auto),size = nrow(Auto)*0.8)
Auto.train = Auto[idx.train,c("mpg","displacement","horsepower","weight")]
Auto.test = Auto[-idx.train,c("mpg","displacement","horsepower","weight")]
```

d)

Vamos a crear la variable mpg01. Para poder hacer el aprendizaje de modo correcto, la mediana la realizaremos sólo con los datos de entrenamiento, y dicha mediana la usaremos para ambos conjuntos.

```
Auto.train = data.frame(mpg01=sign(Auto.train$mpg>=median(Auto.train$mpg))*2-1, Auto.train)
Auto.test = data.frame(mpg01=sign(Auto.test$mpg>=median(Auto.train$mpg))*2-1, Auto.test)
```

Una vez tenemos los datos formateados de la manera desada, procedemos a aplicar los distintos modelos que se nos piden.

Regresión logística

Para realizar la regresión logística, utilizamos la función *glm*.

```
modelo.RegLog = glm(mpg01 ~ displacement + horsepower + weight, data = Auto.train, start=c(log(mean(Aut
```

Con el modelo ya aprendido, realizamos la predicción del test.

```
prediccion.glm = predict(modelo.RegLog,newdata = Auto.test)
```

Ahora, definimos como error cuando el clasificador nos daría la clase incorrecta. Esto se produce cuando el signo de la predicción y el de *mpg01* son distintos.

```
sum((sign(-prediccion.glm*Auto.test$mpg01)+1)/2)/nrow(Auto.test) * 100
```

```
## [1] 6.329114
```

Obtenemos un error bastante bajo, del 6 por ciento.

K-NN

Hacemos algo similar que lo realizado para la regresión logística, pero con la función *knn*. La única consideración adicional es que, al ser K-NN especialmente sensible a la escala de los datos, vamos a normalizarlos.

```
Auto.train.knn = scale(Auto.train[,c("displacement","horsepower","weight")])
media  = attr(Auto.train.knn, "scaled:center")
escala = attr(Auto.train.knn, "scaled:scale")
Auto.test.knn= scale(Auto.test[,c("displacement","horsepower","weight")],media,escala)

Auto.train.knn = data.frame(mpg01=sign(Auto.train$mpg>=median(Auto.train$mpg))*2-1, Auto.train.knn)
Auto.test.knn = data.frame(mpg01=sign(Auto.test$mpg>=median(Auto.train$mpg))*2-1, Auto.test.knn)
```

Ahora que ya tenemos los datos normalizados, pasamos a aplicar los modelos.

```
library(class)
prediccion.knn = knn(Auto.train.knn[,c("displacement","horsepower","weight")],Auto.test.knn[,c("displac
```

A diferencia de la regresión, los resultados del clasificador K-NN son directamente las clases. Luego para saber si hemos fallado, sólo tenemos que mirar si los resultados son distintos.

```
sum(prediccion.knn != Auto.test.knn$mpg01)/nrow(Auto.test.knn) * 100
```

```
## [1] 6.329114
```

El error que encontramos está entorno al 6 por ciento.

Para intentar conocer el mejor parámetro de la variable *k*, vamos a realizar la prueba anterior cambiando los valores de k.

```
for(k in 1:20){
  prediccion.knn = knn(Auto.train.knn[,c("displacement","horsepower","weight")],Auto.test.knn[,c("displa
  error = sum(prediccion.knn != Auto.test.knn$mpg01)/nrow(Auto.test.knn) * 100
  cat(paste0("K: ",k, " \tError: ",error,"\n"))
}
```

```
## K: 1     Error: 8.86075949367089
## K: 2     Error: 7.59493670886076
## K: 3     Error: 6.32911392405063
## K: 4     Error: 6.32911392405063
## K: 5     Error: 6.32911392405063
## K: 6     Error: 7.59493670886076
## K: 7     Error: 6.32911392405063
## K: 8     Error: 6.32911392405063
## K: 9     Error: 6.32911392405063
## K: 10    Error: 6.32911392405063
## K: 11    Error: 6.32911392405063
## K: 12    Error: 7.59493670886076
## K: 13    Error: 6.32911392405063
## K: 14    Error: 6.32911392405063
## K: 15    Error: 6.32911392405063
## K: 16    Error: 6.32911392405063
## K: 17    Error: 6.32911392405063
## K: 18    Error: 6.32911392405063
## K: 19    Error: 6.32911392405063
## K: 20    Error: 6.32911392405063
```

Podemos ver que hay muchos valores donde obtenemos el mismo error. No tenemos más información, luego no podemos decidirnos entre ellos.

R tiene una funcionalidad que, automáticamente nos devuelve el mejor parámetro *k*. Es la función *tune.knn()* de la librería *e1071*.

```
library(e1071)
tune.values = tune.knn(x = Auto.train.knn[,c("displacement","horsepower","weight")], y = as.factor(Auto
summary(tune.values)
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  k
##  8
##
## - best performance: 0.08981855
##
## - Detailed performance results:
```

```
##     k      error dispersion
## 1   1 0.11199597 0.05859207
## 2   2 0.12822581 0.05928052
## 3   3 0.10262097 0.05233485
## 4   4 0.09616935 0.05510179
## 5   5 0.09616935 0.04839923
## 6   6 0.09314516 0.06541330
## 7   7 0.09616935 0.05471313
## 8   8 0.08981855 0.05832940
## 9   9 0.09616935 0.05678704
## 10 10 0.09606855 0.05484474
## 11 11 0.10252016 0.05451435
## 12 12 0.10877016 0.05498743
## 13 13 0.10574597 0.05714826
## 14 14 0.10564516 0.05283515
## 15 15 0.10564516 0.05283515
## 16 16 0.10564516 0.05283515
## 17 17 0.10564516 0.05283515
## 18 18 0.10887097 0.06082825
## 19 19 0.10887097 0.05714935
## 20 20 0.10574597 0.05508796
```

El método nos indica que el mejor k es 8. Para poder realizar el ejercicio sobre la curva ROC, guardamos la predicción del K-NN, pero obteniendo la probabilidad de pertenencia a la clase, y no sólo la clase.

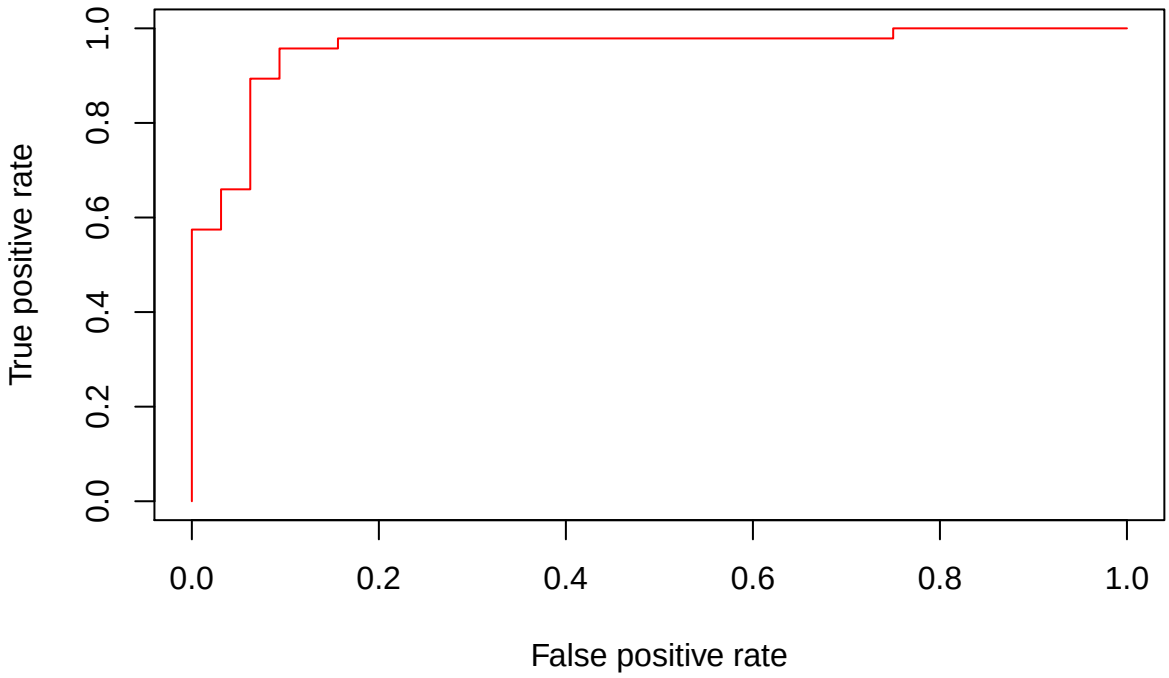
```
prediccion.knn = knn(Auto.train.knn[,c("displacement","horsepower","weight")],Auto.test.knn[,c("displac

prob <- attr(prediccion.knn, "prob")
prob <- 2*ifelse(prediccion.knn == "-1", 1-prob, prob) - 1
```

Curvas ROC

Para poder pintar las *curvas ROC*, tenemos que usar las ordenes del paquete *ROCR prediction* y *performance*.

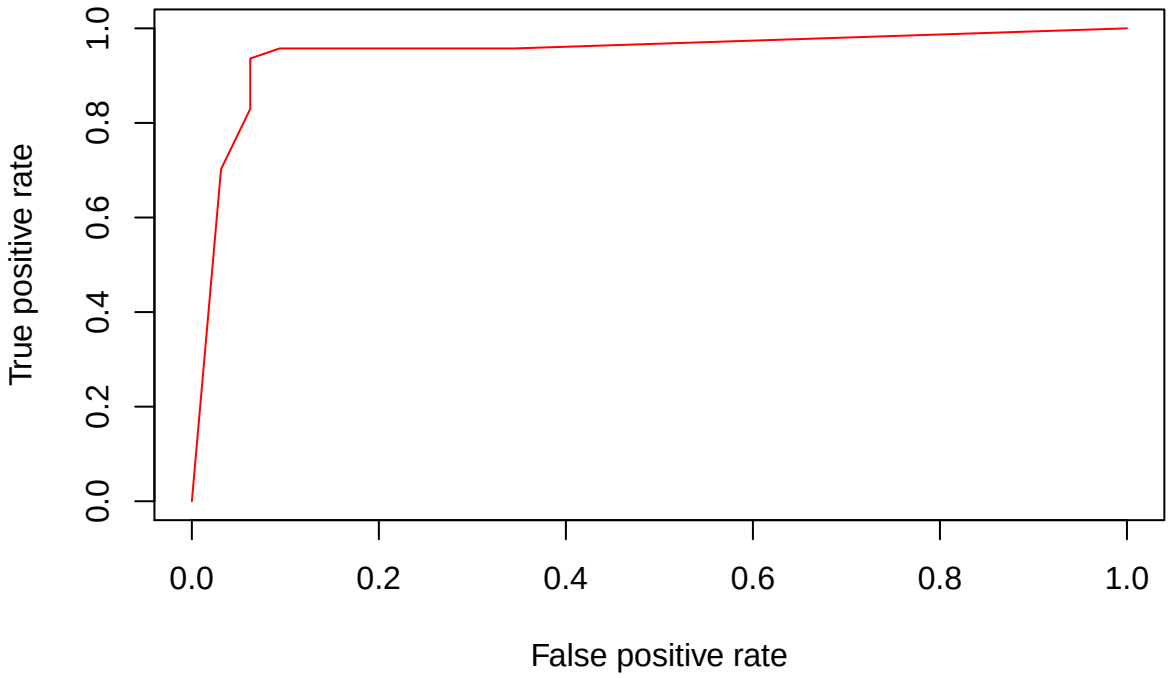
Primero, pintamos la curva de la regresión logística.

```
pred <- prediction(prediccion.glm, Auto.test$mpg01)
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf, col=rainbow(10))
```



Después, pintamos la curva del clasificador K-NN

```
pred <- prediction(prob, Auto.test.knn$mpg01)
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf, col=rainbow(10))
```



Podemos ver que tenemos curvas similares en ambos casos. Eso sí, parece que K-NN se comporta algo mejor, pues la curva parece dominar a la de la regresión.

e) (Bonus)

Para poder estimar los dos modelos usando validación cruzada, tenemos que generar las 5 particiones. Para crear los *folds* o particiones que usaremos para la validación cruzada, podemos usar la función *createFolds* del paquete *caret*.

```
library(caret)
```

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

```
## Loading required package: ggplot2
```

```
folds = createFolds(1:nrow(Auto), k = 5)

error.log = vector("numeric",5)
error.knn = vector("numeric",5)

for(i in 1:5){
  Auto.train = Auto[-folds[[i]],c("mpg","displacement","horsepower","weight")]
  Auto.test = Auto[folds[[i]],c("mpg","displacement","horsepower","weight")]
```

```
  Auto.train = data.frame(mpg01=sign(Auto.train$mpg>=median(Auto.train$mpg))*2-1, Auto.train)
  Auto.test = data.frame(mpg01=sign(Auto.test$mpg>=median(Auto.train$mpg))*2-1, Auto.test)

  Auto.train.knn = scale(Auto.train[,c("displacement","horsepower","weight")])
  media  = attr(Auto.train.knn, "scaled:center")
  escala = attr(Auto.train.knn, "scaled:scale")
  Auto.test.knn= scale(Auto.test[,c("displacement","horsepower","weight")],media,escala)

  Auto.train.knn = data.frame(mpg01=sign(Auto.train$mpg>=median(Auto.train$mpg))*2-1, Auto.train.knn)
  Auto.test.knn = data.frame(mpg01=sign(Auto.test$mpg>=median(Auto.train$mpg))*2-1, Auto.test.knn)

  #Regresion logistica
  modelo.RegLog = glm(mpg01 ~ displacement + horsepower + weight, data = Auto.train, start=c(log(mean(Au
  prediccion.glm = predict(modelo.RegLog,newdata = Auto.test)
  error.log[i] = sum((sign(-prediccion.glm*Auto.test$mpg01)+1)/2)/nrow(Auto.test) * 100
  #Clasificacion K-NN
  prediccion.knn = knn(Auto.train.knn[,c("displacement","horsepower","weight")],Auto.test.knn[,c("displa
  error.knn[i] = sum(prediccion.knn != Auto.test.knn$mpg01)/nrow(Auto.test.knn) * 100
}

cat(paste0("Error medio cometido con regresión logística: ",mean(error.log)))
```

```
## Error medio cometido con regresión logística: 10.9877885827253
```

```
cat(paste0("Error medio cometido con K-NN: ",mean(error.knn)))
```

```
## Error medio cometido con K-NN: 8.70264756340706
```

Podemos por tanto, confirmar lo visto en el apartado anterior. El K-NN funciona ligeramente mejor que la regresión.

Apartado 2

Primero, vamos a cargar los datos, y a realizar una partición de los mismos.

```
library(MASS)
idx.train = sample(nrow(Boston),size = nrow(Boston)*0.8)
Boston.train = Boston[idx.train,]
Boston.test = Boston[-idx.train,]
```

Vamos a usar un ajuste de tipo LASSO.

```
library(glmnet)
```

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

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

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

```
prediccion.lasso <- glmnet(x = as.matrix(Boston.train[,-1]), y = Boston.train[,1])
cv = cv.glmnet(x = as.matrix(Boston.train[,-1]), y = Boston.train[,1], alpha = 1)
```

Ahora, tomamos los coeficientes y vemos los que están por encima de un umbral dado. En este caso hemos escogido 0.5.

```
lasso.coef = predict(prediccion.lasso,type="coefficients", s = cv$lambda.min)[1:14,]
coef = abs(lasso.coef)>0.5
coef
```

```
## (Intercept)          zn       indus        chas         nox          rm
##        TRUE       FALSE       FALSE        TRUE        TRUE       FALSE
##         age         dis         rad         tax     ptratio       black
##       FALSE        TRUE       FALSE       FALSE       FALSE       FALSE
##       lstat        medv
##       FALSE       FALSE
```

```
Boston.train[1,coef]
```

```
##       crim chas   nox    dis
## 81 0.04113    0 0.426 5.4007
```

Por tanto, vemos que las variables que vamos a usar son *chas*, *nox*, *rm*, *dis* y *rad*.

b)

Ahora entrenamos el modelo con *weight-decay* que nos pide el ejercicio con las variables que conseguimos en el apartado anterior.

```
wd = glmnet(x = as.matrix(Boston.train[,coef]), y = Boston.train[,1], alpha = 0)
```

Con el modelo ya entrenado, vamos a predecir los datos del test, y a ver el error cometido.

```
prediction = predict(wd, newx = as.matrix(Boston.test[,coef]),s = cv$lambda.min, type = "response")
error = mean((prediction - Boston.test[,1])^2)
cat(paste0("Error cometido: ",error))
```

```
## Error cometido: 1.07686988824831
```

El error que obtenemos aceptable, pero se podría mejorar. En un principio podríamos pensar que hay un poco de *underfitting*.

c)

Vamos a generar la variable adicional.

```
crim01 = sign(Boston[,1] >= median(Boston[,1]))*2-1
new.crim = crim01[idx.train]
Boston.train.01 = data.frame(Boston.train, new.crim)
new.crim = crim01[-idx.train]
Boston.test.01  = data.frame(Boston.test, new.crim)
```

Ahora, una vez tenemos generados las nuevas variables, vamos a entrenar el SVM con núcleo lineal.

```
model.svm = svm(new.crim ~ chas+nox+rm+dis+rad, data = Boston.train.01, kernel = "linear")
prediction = predict(model.svm, newdata = Boston.test.01)

(sum((sign(-prediction*Boston.test.01$new.crim)+1)/2)/nrow(Boston.test.01)) *100
```

```
## [1] 21.56863
```

Ahora vamos a probar con otros núcleos, para ver si mejoramos el resultado. Primero veamos que pasa con un kernel polinomial.

```
model.svm = svm(new.crim ~ chas+nox+rm+dis+rad, data = Boston.train.01,kernel = "polynomial")
prediction = predict(model.svm, newdata = Boston.test.01)

(sum((sign(-prediction*Boston.test.01$new.crim)+1)/2)/nrow(Boston.test.01)) *100
```

```
## [1] 20.58824
```

Y ahora con un kernel radial.

```
model.svm = svm(new.crim ~ chas+nox+rm+dis+rad, data = Boston.train.01,kernel = "radial")
prediction = predict(model.svm, newdata = Boston.test.01)

(sum((sign(-prediction*Boston.test.01$new.crim)+1)/2)/nrow(Boston.test.01)) *100
```

```
## [1] 14.70588
```

Como podemos ver, el kernel radial obtiene los mejores resultados de error en test.

Bonus

Vamos a utilizar el mismo método de realizar la validación cruzada que en el primer bonus.

```
folds = createFolds(1:nrow(Boston), k = 5)

error.in = vector("numeric",5)
error.test = vector("numeric",5)

for(i in 1:5){
  #Generar particion
  Boston.train = Boston[-folds[[i]],]
  Boston.test = Boston[folds[[i]],]

  crim01 = sign(Boston[,1] >= median(Boston[,1]))*2-1
  new.crim = crim01[-folds[[i]]]
  Boston.train.01 = data.frame(Boston.train, new.crim)
  new.crim = crim01[folds[[i]]]
  Boston.test.01  = data.frame(Boston.test, new.crim)
```

```
  #Calcular el modelo
  model.svm = svm(new.crim ~ chas+nox+rm+dis+rad, data = Boston.train.01,kernel = "radial")

  #Error de test
  prediction = predict(model.svm, newdata = Boston.test.01)
  error.test[1] = (sum((sign(-prediction*Boston.test.01$new.crim)+1)/2)/nrow(Boston.test.01)) *100

  #Error de train
  prediction = predict(model.svm, newdata = Boston.train.01)
  error.test[1] = (sum((sign(-prediction*Boston.train.01$new.crim)+1)/2)/nrow(Boston.train.01)) *100
}

cat(paste0("Error medio en train: ",mean(error.in)))
```

```
## Error medio en train: 0
```

```
cat(paste0("Error medio en test: ",mean(error.test)))
```

```
## Error medio en test: 2.8641975308642
```

Podemos ver que acierta exactamente en el error en la muestra, y que consigue muy buenos resultados en el test.

Apartado 3

a)

Para empezar, vamos a prepara las particiones de entrenamiento(80%) y de test(20%), como ya hemos realizado en otros apartados.

```
library(MASS)
idx.train = sample(nrow(Boston),size = nrow(Boston)*0.8)
Boston.train = Boston[idx.train,]
Boston.test = Boston[-idx.train,]
```

Con los datos ya preparamos, vamos a realizar los ejercicios.

b)

Bagging es un *random forest* donde podemos coger cualquier atributo para la generación de arboles. Por tanto, seleccionamos el parámetro *mtry* que indica el número de atributos a considerar para generar un árbol como el número total de atributos, 13.

```
library(randomForest)
```

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

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

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

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
random.forest = randomForest(medv~crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat, data =
pred = predict(random.forest,Boston.test)
error = sum(abs(Boston.test["medv"]-pred))/nrow(Boston.test)
cat(paste0("El error obtenido es: ",error))
```

```
## El error obtenido es: 2.26002751633987
```

Obtenemos un error bastante bajo, entorno al 2%.

c)

Para un *random forest* general, eliminamos el parámetro *mtry* usado en el apartado anterior. Entonces automáticamente, la función tomará un número menor de atributos que el máximo, teniendo en cuenta el número total de atributos. En resumen, eligiremos entre menos atributos que el total, realizando por tanto un *random forest*.

```
random.forest = randomForest(medv~crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat, data =
pred = predict(random.forest,Boston.test)
error = sum(abs(Boston.test["medv"]-pred))/nrow(Boston.test)
cat(paste0("El error obtenido es: ",error))
```

```
## El error obtenido es: 2.16149616896733
```

El error obtenido es ligeramente inferior al bagging, pero casi no se puede apreciar.

d)

Vamos a ajustar un modelo de Boosting para poder comparar los resultados de los apartados anteriores.

```
library(gbm)
```

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

## Loading required package: survival

##
## Attaching package: 'survival'

## The following object is masked from 'package:caret':
##
##     cluster
```

```
## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.1
```

```
boost = gbm(medv~crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+black+lstat, data = Boston.train, dis
pred = predict(boost,Boston.test,n.trees = 100)
error = sum(abs(Boston.test["medv"]-pred))/nrow(Boston.test)
cat(paste0("El error obtenido es: ",error))
```

```
## El error obtenido es: 6.11451940297085
```

Este error es entorno al 6%. Es decir, que ambos modelos, tanto *bagging* como *random forest* son muy buenas estimaciones, sacando gran ventaja al modelo de *Boosting*. Podemos ver entonces el potencial de generar la decisión tomando multiples opiniones (multiples árboles) obtiene muy buenos resultados, ligeramente mejores para *random forest*.

Apartado 4

a)

Primero, vamos a preparar los datos. Vamos a utilizar el mismo mecanismo que en los demás apartados.

```
idx.train = sample(nrow(OJ),size = 800)
OJ.train = OJ[idx.train,]
OJ.test = OJ[-idx.train,]
```

Vamos a entrenar el árbol de decisión.

```
library(tree)
arbol = tree(Purchase~WeekofPurchase+StoreID+PriceCH+PriceMM+DiscCH+DiscMM+SpecialCH+SpecialMM+LoyalCH+S
```

Con esto, ya habríamos ajustado nuestro árbol con los datos de entrenamiento.

b)

Vamos a ver los resultados de la orden *summary*.

```
summary(arbol)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ WeekofPurchase + StoreID + PriceCH +
##     PriceMM + DiscCH + DiscMM + SpecialCH + SpecialMM + LoyalCH +
##     SalePriceMM + SalePriceCH + PriceDiff + Store7 + PctDiscMM +
##     PctDiscCH + ListPriceDiff + STORE, data = OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7504 = 594.3 / 792
## Misclassification error rate: 0.1612 = 129 / 800
```

Obtenemos una lista de características del árbol que hemos generado. Destacar primero que pese al gran número de variables, sólo usamos 5 para el árbol final. Esto podría ser sorprendente, pues dado la gran cantidad de variables que podría seleccionar, selecciona un grupo muy pequeño. Debido el funcionamiento de los árboles, si usaramos todas las variables, el árbol sería mucho más preciso, pudiendo fácilmente legar el error a 0. Lo que sucede es que se somete al árbol a un proceso de regularización para mejorar la generalización, que nos reduce el árbol hasta lo que obtenemos.

Destacar también el error que nos da, que es entorno a 0.16.

c)

```
plot(arbol, uniform=TRUE,
    main="Árbol de clasificación")
```

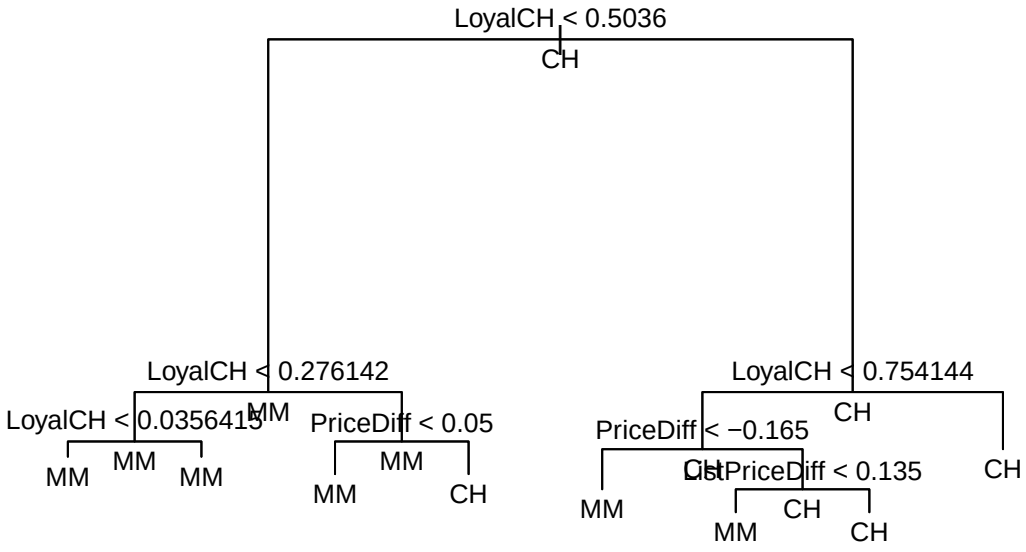
```
## Warning in text.default(x[1L], y[1L], "|", ...): "uniform" is not a
## graphical parameter
```

```
## Warning in plot.xy(xy.coords(x, y), type = type, ...): "uniform" is not a
## graphical parameter
```

```
text(arbol, use.n=TRUE, all=TRUE, cex=.8)
```

```
## Warning in text.default(xy$x[ind], xy$y[ind] + 0.5 * charht, rows[ind], :
## "use.n" is not a graphical parameter
```

```
## Warning in text.default(xy$x[leaves], xy$y[leaves] - 0.5 * charht, labels =
## stat, : "use.n" is not a graphical parameter
```



Podemos ver que el árbol cumple los datos que obteníamos del *summary*. Las primeras elecciones, como sabemos por el funcionamiento de la generación de los árboles, son las que aportan más separación a los datos. Entonces, podemos usar esta representación del árbol para conocer los valores más importantes para clasificar la muestra. En este caso, el factor que claramente es más importante es el *LoyalCH*.

d)

Para poder calcular la matriz de confusión, debemos tomar una decisión sobre el valor donde pasamos de clasificar un punto como *CH* para clasificarlo como *MM*. Por defecto, vamos a tomar la decisión de asignar a los valores superiores de 0.5 a *CH* y los valores menores o iguales a la clase *MM*

```
probabilidad.prediccion = predict(arbol,OJ.test)
OJ.prediction= vector(length = nrow(probabilidad.prediccion))
OJ.prediction[probabilidad.prediccion[,1]>0.5] = "CH"
OJ.prediction[probabilidad.prediccion[,1]<=0.5] = "MM"
confusionMatrix(OJ.prediction,OJ.test$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 155  27
##         MM  20  68
##
##                Accuracy : 0.8259
```

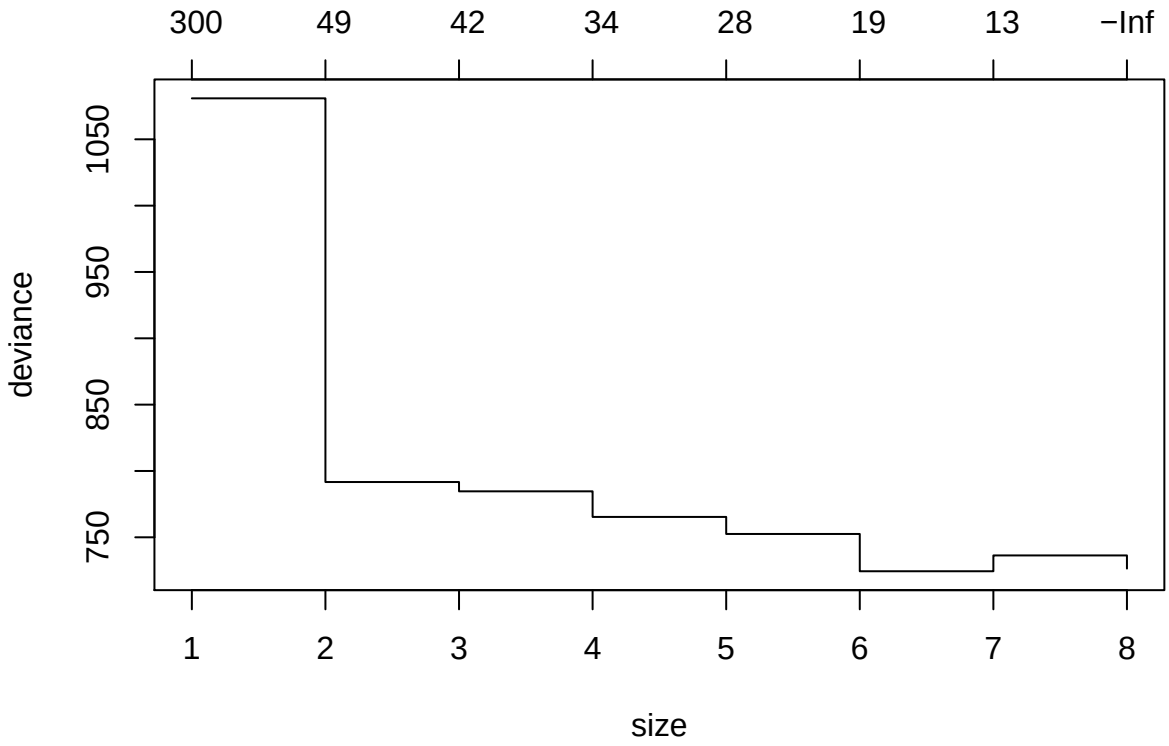
```
##                  95% CI : (0.7753, 0.8692)
##     No Information Rate : 0.6481
##     P-Value [Acc > NIR] : 8.008e-11
##
##                   Kappa : 0.6118
##  Mcnemar's Test P-Value : 0.3815
##
##             Sensitivity : 0.8857
##             Specificity : 0.7158
##          Pos Pred Value : 0.8516
##          Neg Pred Value : 0.7727
##              Prevalence : 0.6481
##          Detection Rate : 0.5741
##    Detection Prevalence : 0.6741
##       Balanced Accuracy : 0.8008
##
##        'Positive' Class : CH
##
```

Obtenemos un error de 0.82. Más o menos obtenemos el mismo número de falsos positivo que negativos. Antes hemos elegido un punto de corte aleatoriamente. Si quisieramos conseguir modificar este comportamiento, podríamos ser más o menos restrictivos al seleccionar un valor como *CH*, subiendo o bajando ese punto de corte. Este tipo de estudio es el que podemos observar en la curva ROC.

e) y bonus

Vamos a realizar un experimento con validación cruzada, utilizando la función *cv.tree()*

```
tr0.cv = cv.tree(arbol)
plot.tree.sequence(tr0.cv)
```



Como podemos ver, el tamaño del árbol óptimo está entorno a 6. Esto nos indica que el árbol que hemos generado durante el ejercicio tiene demasiados nodos.