Práctica 3

Laura Tirado López

31 de mayo de 2016

Ejercicio 1

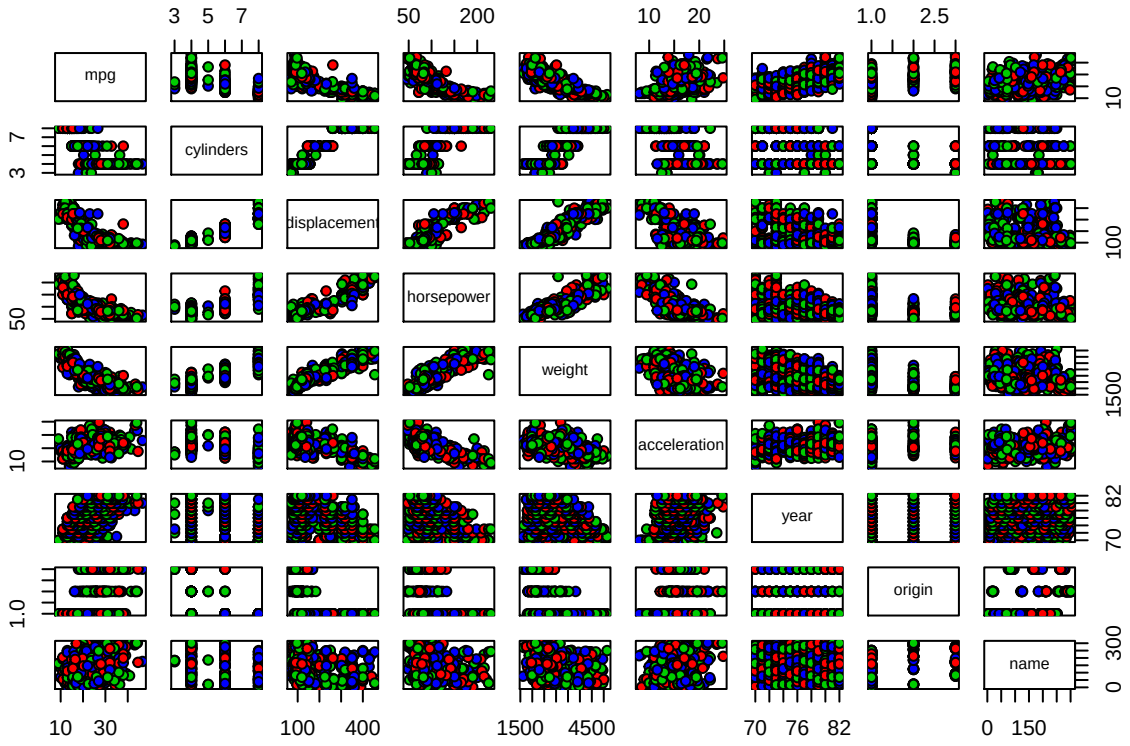
Usar el conjunto de datos Auto que es parte del paquete ISLR. En este ejercicio desarrollaremos un modelo para predecir si un coche tiene un consumo de carburante alto o bajo usando la base de datos Auto. Se considerará alto cuando sea superior a la mediana de la variable mpg y bajo en caso contrario.

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.

```
#library("ISLR", lib.loc="~/R/x86_64-pc-linux-gnu-library/3.0")
library("ISLR", lib.loc="~/R/win-library/3.2")
```

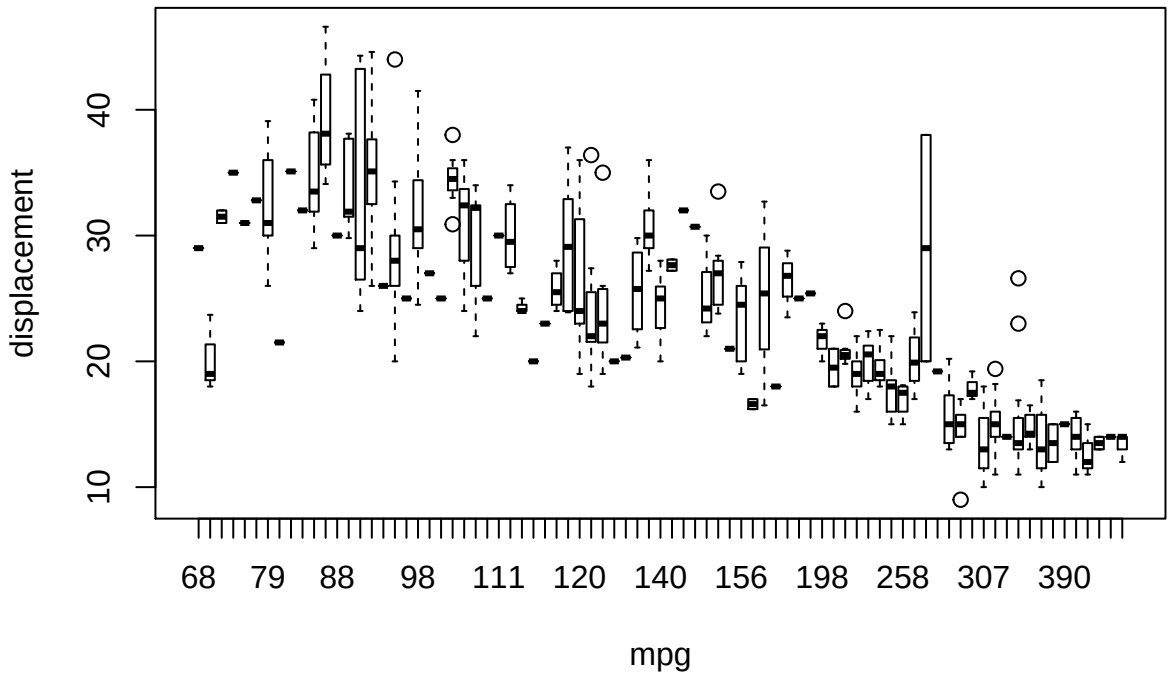
```
## Warning: package 'ISLR' was built under R version 3.2.5
```

```
pairs( ~., data=Auto,pch=21, bg=c("red","green3","blue"))
```

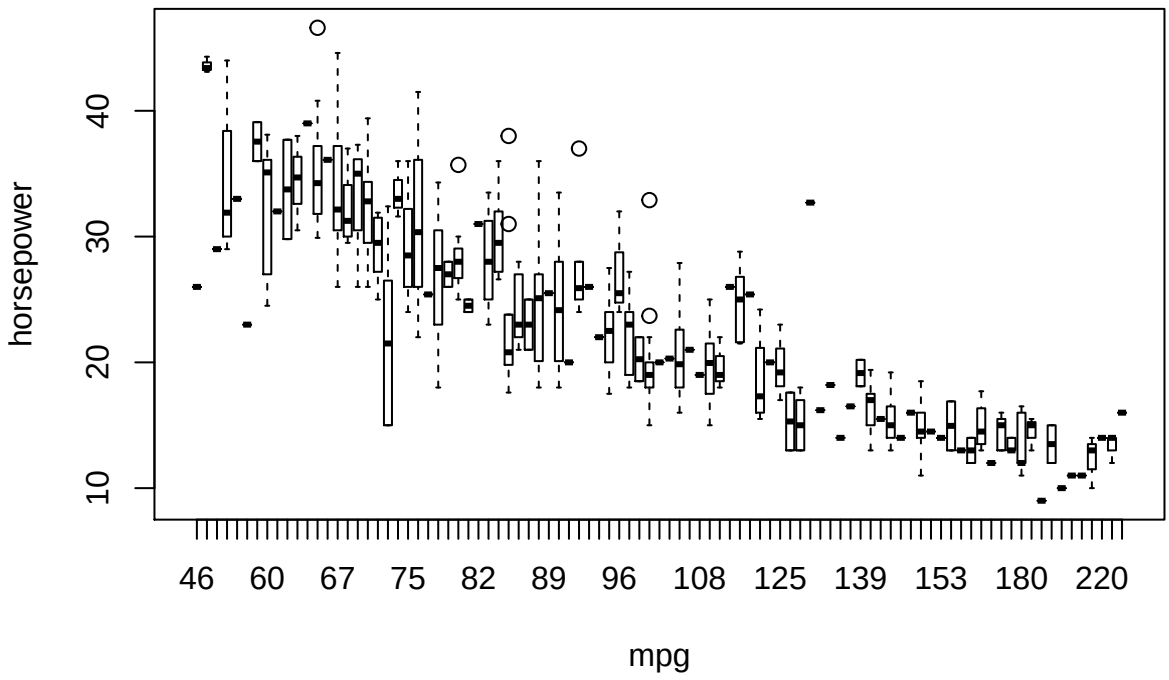


A partir del resultado de la función pairs() podemos ver que mpg tiene dependencia con las características displacement, horsepower y weight.

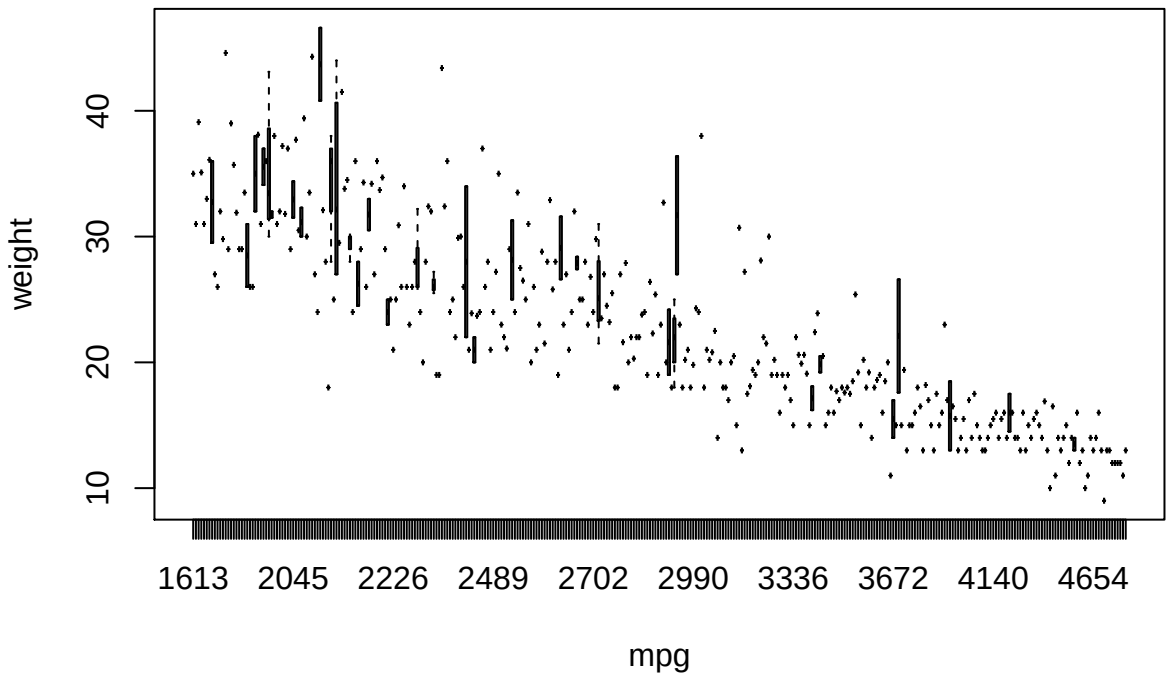
```
boxplot(mpg ~ displacement, data=Auto,xlab="mpg",ylab="displacement")
```




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




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



Con las gráficas generadas por boxplot() podemos ver la dependencia funcional entre las variables displacement, horsepower y weight con la variable mpg.

b) Seleccionar las variables predictoras que considere m?s relevantes.

Las variables predictoras más relevantes son displacement, horsepower y weigth. Para seleccionarlas, las guardamos en una variable.

```
variables <- Auto[,c(1,3,4,5)]
```

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

Para particionar el conjunto generamos aleatoriamente un vector de índices de unos y doses para el número de filas de nuestro conjunto de datos con las probabilidades indicadas en el enunciado. A partir de esos índices, generamos los dos subconjuntos: las filas de índice 1 serán parte del conjunto de entrenamiento y las filas de índice 2 seran parte del conjunto de test.

```
set.seed(1234)
indices <- sample(2,nrow(variables),replace=TRUE,prob=c(0.8,0.2))
training <- variables[indices==1, ]
test <- variables[indices==2, ]
```

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 funci?n 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).

Creamos la variable usando la función sign().

```
mediana <- median(variables[,1])
mpg01_train <- sign(training[,1]-mediana)
mpg01_test <-sign(test[,1]-mediana)
```

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.

Para ajustar el modelo utilizamos la función glm(). Como variables predictoras utilizamos las variables seleccionadas anteriormente. En mi caso elimino la primera columna porque coincide con la variable mpg.

```
#Ajustamos el modelo
m1 <- glm(mpg01_train ~ .,data=training[,c(2:4)])
#Hacemos la predicción
prediccion <- sign(predict(m1, newdata=test[,c(2:4)]))
#Calculamos el error.
error <- (sum(abs(prediccion-mpg01_test))/2)/length(mpg01_test)
error
```

```
## [1] 0.1216216
```

El error de test del modelo lo calculamos midiendo el número de diferencias entre los resultados que nos da la predicción y la variable mpg01 asociada a los datos de test. Podemos ver que el error de test del modelo es 0,1216 (12,16%).

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.

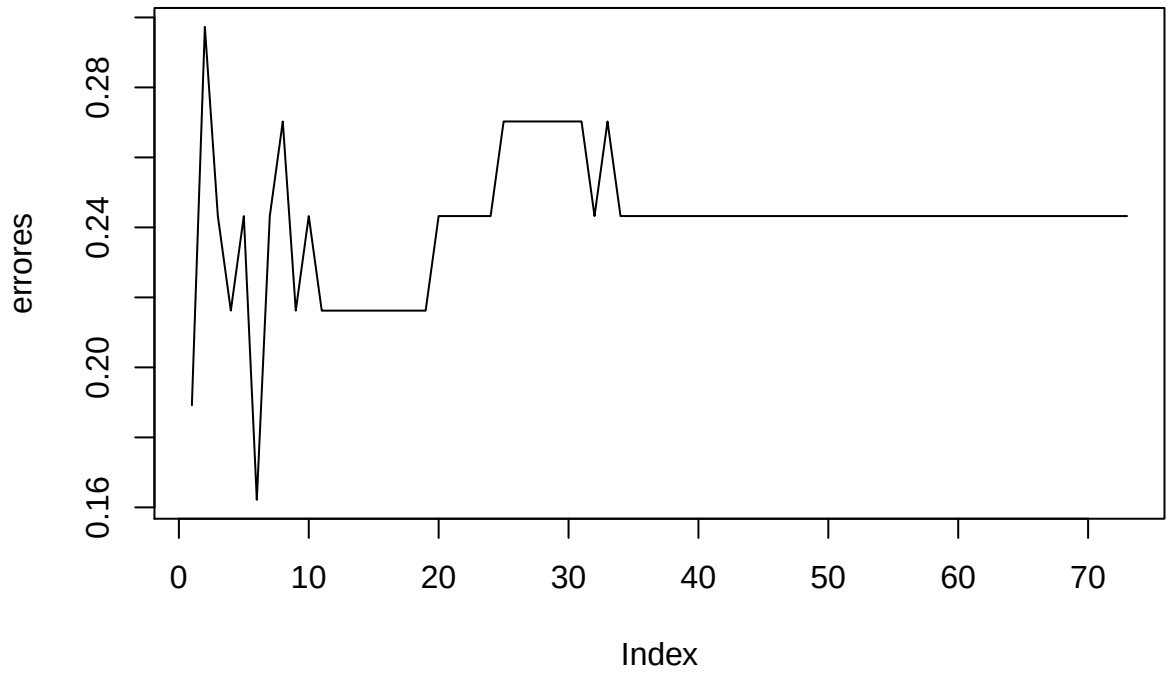
Para ajustar el modelo utilizamos la función knn.

```
library(class)
# Ajustamos el modelo
mod_knn <- knn(training,test,mpg01_train, k=1)
#Calculamos el error
error_knn <- (sum(abs(as.numeric(as.character(mod_knn))-mpg01_test))/2) / length(mpg01_test)
error_knn
```

```
## [1] 0.09459459
```

Con valor k=1, el error de test es 0,0945 (9,45%).

```
errores <- NULL
for (i in 1:73) {
library(class)
  pred_knn <- knn(training,test,mpg01_train, k=i)
  error_knn <- sum(abs(as.numeric(as.character(pred_knn))-mpg01_test)) / length(mpg01_test)
  errores <- c(errores, error_knn)
}
plot(errores, type='l')
```




```
#Buscamos el k con menor error
which.min(errores)
```

```
## [1] 6
```

El valor de k con el que mejor se ajustan los datos es con k=6.

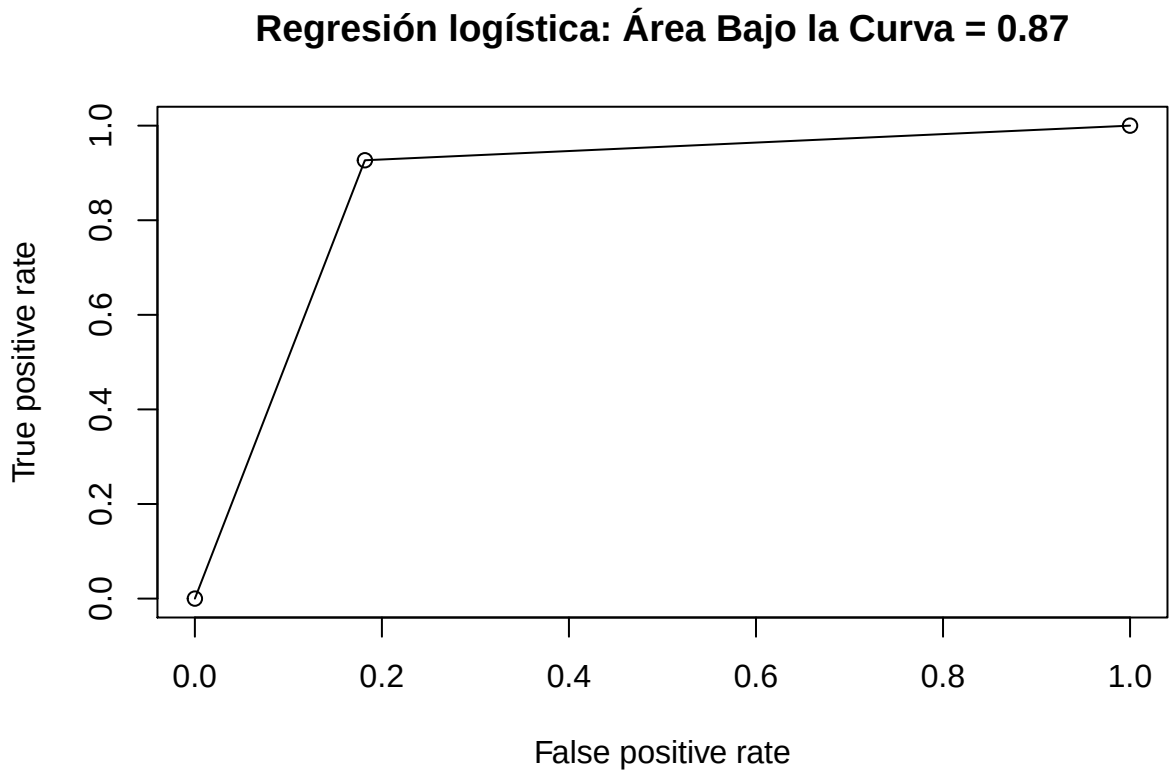
Pintar las curvas ROC (instalar paquete ROCR en R) y comparar y valorar los resultados obtenidos para ambos modelos.

```
#Cargamos el paquete ROCR
library("ROCR", lib.loc="~/R/win-library/3.2")
```

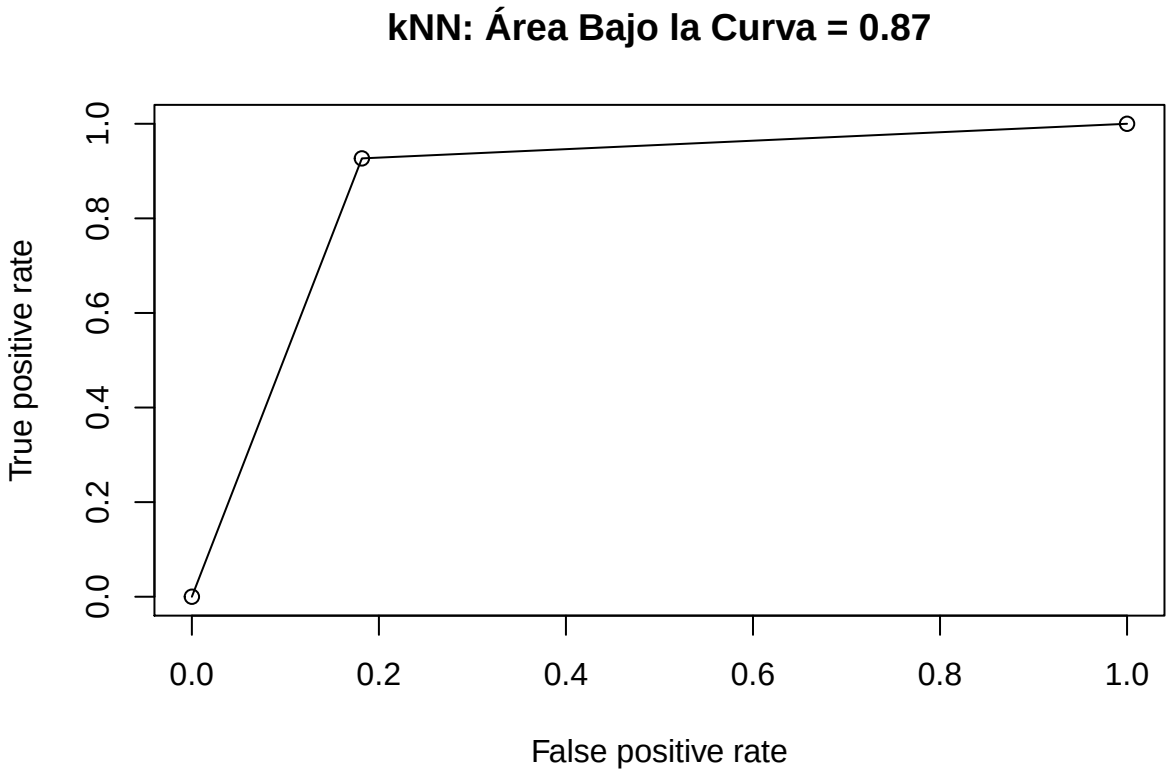
```
## Warning: package 'ROCR' was built under R version 3.2.5
```

```
## Loading required package: gplots

## Warning: package 'gplots' was built under R version 3.2.5

##
## Attaching package: 'gplots'

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

```
predict.rocr  <- prediction(prediccion,mpg01_test)
perf.rocr     <- performance(predict.rocr,"tpr","fpr")
auc <- as.numeric(performance(predict.rocr ,"auc")@y.values)
plot(perf.rocr,type='o', main = paste('Regresión logística: Área Bajo la Curva =',round(auc,2)))
```




```
predict.rocr  <- prediction((as.numeric(as.character(pred_knn))),mpg01_test)
perf.rocr     <- performance(predict.rocr,"tpr","fpr")
auc <- as.numeric(performance(predict.rocr ,"auc")@y.values)
plot(perf.rocr,type='o', main = paste('kNN: Área Bajo la Curva =',round(auc,2)))
```



Para comparar los modelos hemos calculado el área bajo la curva ROC (AUC) de cada modelo. Para el modelo de regresión logística el valor de ésta es 0,87 al igual que para el kNN. Podemos ver que en este caso, ambos modelos son equivalentes, según las curvas de ROC.

Ejercicio 2

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

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

```
#Cargamos los paquetes glmnet y MASS
library("glmnet", lib.loc="~/R/win-library/3.2")
```

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

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

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

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

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

```
library("MASS", lib.loc="C:/Program Files/R/R-3.2.3/library")
#Calculamos el error de predicción GCV
datos <- as.matrix(Boston[c(2:ncol(Boston))])
crim <- as.numeric(Boston$crim >= median(Boston$crim))
lassoreg <- cv.glmnet(datos,crim)
#Ajustamos el modelo al valor de lamba que minimiza el error GCV
coef(lassoreg,s=lassoreg$lambda.min)
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept) -1.0146738096
## zn          -0.0010396562
## indus        0.0006035915
## chas         .
## nox          1.8339924232
## rm           0.0076629460
## age          0.0027736677
## dis          .
## rad          0.0155544062
## tax          .
## ptratio      0.0030455811
## black       -0.0001533190
## lstat        .
## medv         0.0050609945
```

Podemos ver que el subconjunto óptimo de variables predictoras está formado por zn, indus, nox, rm, age, rad, ptratio, black y medv.

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".

```
var_sel <- as.matrix(Boston [,c(2,3,5,6,7,9,11,12,14)])
ridge <- glmnet(var_sel,crim,alpha=0)
#Calculamos el error residual
residuos <- predict(ridge,var_sel,s=0)
error <- mean(residuos)
error
```

```
## [1] 0.5
```

El error residual es 0,5.

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.

Para ajustar los modelos utilizaremos la función svm().

```
#Creamos la variable binaria
mediana <- median(Boston$crim)
crim_01 <- sign(Boston[,1]-mediana)
#Cargamos el paquete e1071
library("e1071", lib.loc="~/R/win-library/3.2")
```

```
## Warning: package 'e1071' was built under R version 3.2.5
```

Vamos a ajustar un modelo lineal, un modelo polinomial, un modelo sigmoidal y un modelo con kernel de base radial:

a) Modelo lineal:

```
#Ajustamos el modelo
modelo <- svm(crim_01 ~., data=var_sel, cost=100, kernel='linear')
#Predecimos y calculamos el número de errores
prediccion <- (predict(modelo, var_sel)>0)*2-1
error <- sum(crim_01!=prediccion) / length(crim_01)
error
```

```
## [1] 0.173913
```

b) Modelo polinomial:

```
# Ajustamos el modelo
modelo <- svm(crim_01 ~ ., data=var_sel, cost=100, kernel='polynomial')
# Predecimos y calculamos el número de errores
prediccion <- (predict(modelo, var_sel)>0)*2-1
error <- sum(crim_01!=prediccion) / length(crim_01)
error
```

```
## [1] 0.08498024
```

c) Modelo sigmoidal:

```
# Ajustamos el modelo
modelo <- svm(crim_01 ~ ., data=var_sel, cost=100, kernel='sigmoid',gamma=1)
# Predecimos y calculamos el número de errores
prediccion <- (predict(modelo, var_sel)>0)*2-1
error <- sum(crim_01!=prediccion) / length(crim_01)
error
```

```
## [1] 0.4822134
```

d) Modelo de base radial:

```
# Ajustamos el modelo
modelo <- svm(crim_01 ~ ., data=var_sel, cost=100, kernel='radial', gamma=1)
# Predecimos y calculamos el número de errores
prediccion <- (predict(modelo, var_sel)>0)*2-1
error <- sum(crim_01!=prediccion) / length(crim_01)
error
```

```
## [1] 0.001976285
```

Comparando todos los modelos el modelo de base radial es el mejor de todos. El modelo polinomial mejor al modelo lineal, el cual tiene un error bastante considerable. Por otra parte, el modelo sigmoidal empeora mucho los resultados, siendo mucho peor que el modelo lineal.

Ejercicio 3

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

```
#Cargamos las librerías
library("randomForest", lib.loc="~/R/win-library/3.2")
```

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

## randomForest 4.6-12

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

```
library("gbm", lib.loc="~/R/win-library/3.2")
```

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

## Loading required package: survival

## Loading required package: lattice

## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.1
```

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

Para dividir la base de datos utilizamos el mismo método que en el apartado c) del ejercicio 1.

```
variables = Boston
indices <- sample(2,nrow(variables),replace=TRUE,prob=c(0.8,0.2))
training <- variables[indices==1, ]
test <- variables[indices==2, ]
```

b) 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.

Para ajustar un modelo de regresión utilizando bagging, usamos la función bagging pasándole como parámetros la variable que queremos predecir (training$medv) y el conjunto de variables de training.

```
library("ipred", lib.loc="~/R/win-library/3.2")
```

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

```
#Ajustamos el modelo
bagfit <- bagging(training$medv ~., data=training[,c(1:13)])
#Calculamos el error de test
bag_residuos <- predict(bagfit,newdata=test[,c(1:13)])
bag_error <- mean(bag_residuos)
bag_error
```

```
## [1] 22.05829
```

El error de test es 21,40086.

c)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 el modelo de regresión utilizamos la función randomForest(). Los parámetros que usamos son la variable de respuesta (training$mdve) y un subconjunto de las características predictoras de $\sqrt{m}$ características, siendo m el número total de caracetrísticas. EStе subconjunto se escoge de manera aleatoria.

```
#Escogemos p características aleatorias
p <- sample(1:ncol(training),sqrt(ncol(training)-1))
#Ajustamos el modelo
forest_fit <- randomForest(training$medv ~., data=training[,p])
#Calculamos el error de test
forest_residuos <- predict(forest_fit,newdata=test[,p])
forest_error <- mean(forest_residuos)
forest_error
```

```
## [1] 22.23666
```

El error de test es 22,09276. Podemos ver que el error de test de baggin es menor que el error de test de "Random Forest", por lo que la desviación es menor con el modelo bagging.

d) 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 ajustar el modelo de regresión utilizamos la función gbm. Los parámetros que usamos son la variable de respuesta (training$mdve) y el conjunto de datos de training, exceptuando la última columna que es la que coincide con nuestra variable de respuesta. Además, le damos el valor "gaussian" al parámetro distribution.

```
#Ajustamos el modelo
boosting_fit <- gbm(training$medv ~., data=training[,c(1:13)], distribution="gaussian")
#Calculamos el error de test
boosting_residuos <- predict(boosting_fit,newdata=test[,c(1:13)],n.trees=100)
error_boosting <- mean(boosting_residuos)
error_boosting
```

```
## [1] 22.48352
```

El error de test es 22,71086. Comparado con los otros errores de test, el del modelo boosting es el mayor, por lo que es en el que hay una mayor desviación respecto a los datos de test.

Como conclusión, en este caso y para los datos empleados, el modelo de bagging es mejor que los modelos "Random Forest" y boosting.

Ejercicio 4

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

```
library("ISLR", lib.loc="~/R/win-library/3.2")
```

a) 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).

```
#Hacemos los conjuntos de entrenamiento y test
variables = OJ
indices <- sample(1:nrow(variables),800,replace=FALSE)
training <- variables[indices, ]
test <- variables[-indices, ]
#Cargamos el paquete tree
library("tree", lib.loc="~/R/win-library/3.2")
```

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

```
#Ajustamos el modelo
treefit <- tree(training$Purchase ~.,data=training[,c(2:18)])
```

b) 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(treefit)
```

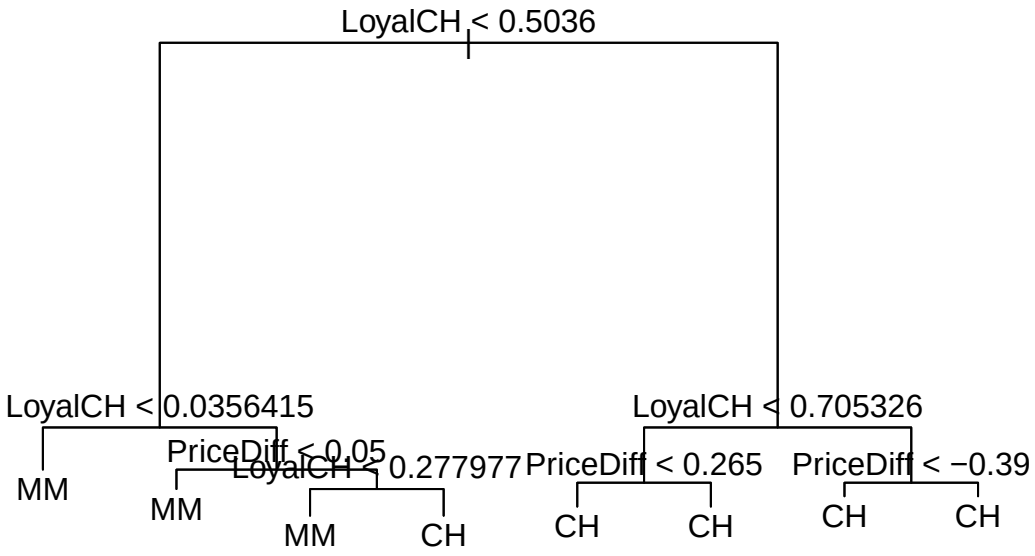
```
##
## Classification tree:
## tree(formula = training$Purchase ~ ., data = training[, c(2:18)])
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7422 = 587.8 / 792
## Misclassification error rate: 0.175 = 140 / 800
```

Como podemos ver la tasa de error viene descrita como Misclassification error rate y es de un 14,62%. El número de nodos terminales del árbol es 9 y se han empleado para su construcción las variables LoyalCH, SpecialCH, PriceDiff y ListPriceDiff.

c) Crear un dibujo del árbol e interpretar los resultados.

Para crear el dibujo del árbol utilizamos las funciones plot y text para añadir los textos al árbol.

```
plot(treefit)
text(treefit)
```



En el caso de que la variable LoyalCH sea menor a 0,48285, el árbol clasifica tres casos como MM y uno cómo CH en función de las variables LoyalCH y SpecialCH. Si es mayor, clasifica dos casos como MM y tres como CH, teniendo en cuenta las variables LoyalCH, ListPriceDiff y PriceDiff.

d) 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?

Predecimos la respuesta de los datos de test con la función predict y creamos la matriz de confusión a partir de los datos de predicción con la función table.

```
treepred <- predict(treefit, newdata=test, type="class")
treetable <- table(test$Purchase,treepred)
treetable
```

```
##     treepred
##       CH  MM
##   CH 157  14
##   MM  37  62
```

De la variable CH hay un total de 172 instancias de las cuales clasificó 139 como CH y 33 como MM. Para la variable MM, de 98 instancias, clasificó 19 como CH y 79 como MM. Podemos ver que puede distinguir más o menos bien cada clase.

La tasa de error del test la calculamos con la fórmula $\frac{FP+FN}{P+N}$ por lo que la tasa de error es $\frac{33+19}{172+98} = \frac{52}{270} = 0,1925$ (19,25%).

La precisión del test la calculamos con la fórmula $\frac{TP+TN}{P+N}$ por lo que la precisión del test es $\frac{139+79}{172+98} = \frac{218}{270} = 0,8074$ (80,74%).

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

```
cv.model <- cv.tree(treefit)
#Determinamos el tamaño óptimo a partir de los parámetros calculados por cv.tree
best.size <- cv.model$size[which(cv.model$dev==min(cv.model$dev))]
best.size
```

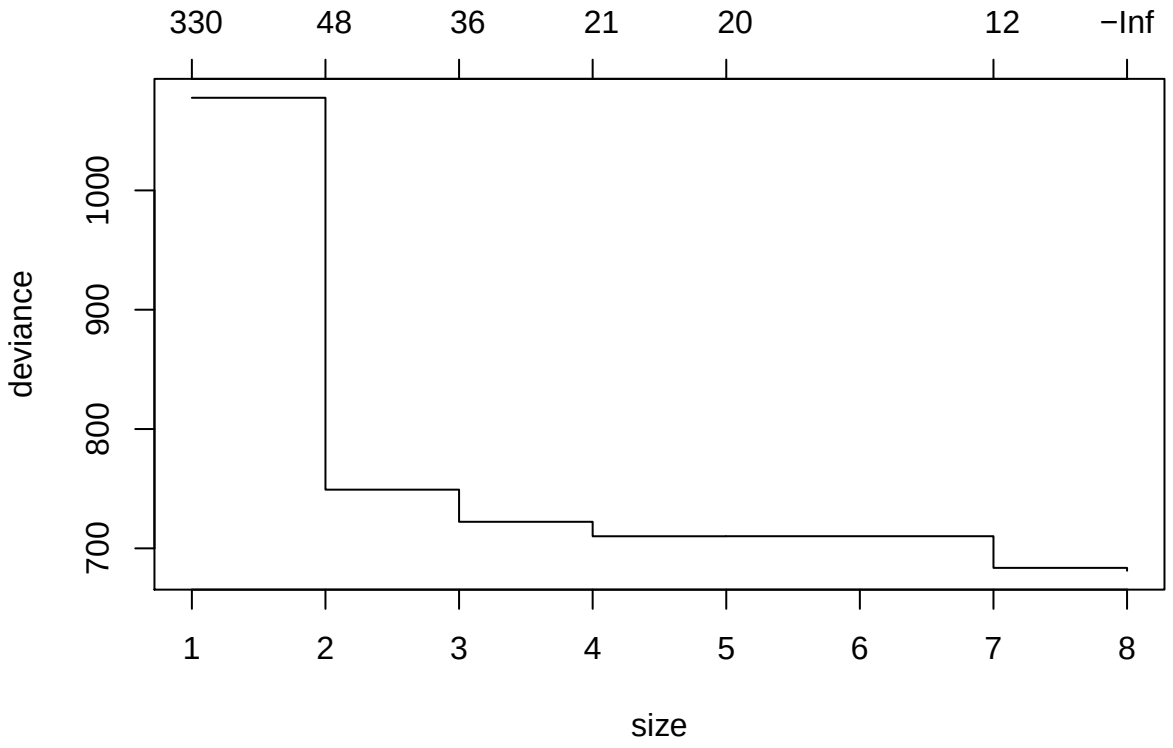
```
## [1] 8
```

El tamaño óptimo del árbol es 8. La función cv.tree() realiza una función de validación cruzada para encontrar la desviación o el número de errores de clasificación y utilizarlo como un parámetro de la función coste-complejidad y encontrar el tamaño óptimo del árbol.

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?

Para generar el gráfico utilizamos la función plot.

```
plot(cv.model)
```



Como podemos ver en la gráfica el tamaño de árbol que corresponde a la tasa más pequeña de error el tamaño=8, como calculamos en el apartado anterior.