Trabajo 3 | Programación

Antonio Miguel Morillo Chica

22/05/2018

1. Ajustes de Modelos Lineales (Clasificación Optical Recognition of Handwritten Digits)

1.1. Comprender el problema a resolver.

El problema que vamos a tratar se basa en la clasificación de números escritos. Optical Recognition of Handwritten Digits es una base de datos que posee una batería de ejemplos divididos en train y test. Cada ejemplo está constituido por 64 caracteristicas que representan el mapa de bits de los números. El problema trata de clasificar los ejemplos.

1.2. Preprocesado los datos: por ejemplo categorización, normalización, etc.

Para el preprocesado de los datos usaremos la función preProces() reduciremos mucho el número de características, además quitaremos la que sean muy proximas a 0 y sean despreciables.

1.3. Selección de clases de funciones a usar.

Las funciones que usaremos para clasificar será las lineales.

1.4. Definición de los conjuntos de training, validación y test usados en su caso.

Los conjuntos train y test están divididos en dos ficheros distintos con distinto número de ejemplos. - Train: 3823 ejemplos con 64 caracteristicas. - Test : 1797 ejemplos con 64 caracteristicas. Las caracteristicas son una matriz de entrada de 8x8 donde cada elemento es un entero de 0..16. Esta entrada organizada así para reducir la dimensionalidad y la invarianza a pequeñas distorsiones.

Para leer los datos:

```r
# Leemos las matrices.
train <- read.csv("./datos/optdigits_tra.csv", head =FALSE)
test  <- read.csv("./datos/optdigits_tes.csv", head =FALSE)

# Guardamos sus clases
train.clases = train[,dim(train)[2]]
test.clases = test[, dim(test)[2]]

# Cambiamos el tipo de dato de las clases y les ponemos nombre
# a las filas y columnas.
train[,dim(train)[2]] <- as.factor(train[,dim(train)[2]])
test[,dim(test)[2]] <- as.factor(test[,dim(test)[2]])
colnames(train) <- c(paste("P.",1:64),"Y")
colnames(test) <- c(paste("P.",1:64),"Y")
```

1.5. Discutir la necesidad de regularización y en su caso la función usada para ello.

La regularización será discutible cuando veamos los errores en el Eout ya que no sabremos hasta entonces si existe sobreajuste en los datos, de todas formas trataremos los datos en el preProcesado, que lo haremos de la siguiente forma:

```r
# Dimensión antes del preProcesado:
cat("Dim Train Original: ", dim(train), "\n")
```

```
## Dim Train Original:  3823 65
```

```r
cat("Dim Test  Original: ", dim(test),  "\n")
```

```
## Dim Test  Original:  1797 65
```

```r
# Reducimos el número de características eliminando las
# que son muy proximas a 0 en train y test:
nzv1 <- nearZeroVar(train)
nzv2 <- nearZeroVar(test)
train <- train[,-nzv1]
test  <- test [,-nzv2]

cat("Dim Train Sin 0: ", dim(train), "\n")
```

```
## Dim Train Sin 0:  3823 49
```

```r
cat("Dim Test  Sin 0: ", dim(test),  "\n")
```

```
## Dim Test  Sin 0:  1797 49
```

```r
# Demasiados atributos para este problema?
# Aplicamos la dimensionalidad con PCA que es un filtro de
# características no supervisado es sensible al escalado y
# valores grandes. Usaré YeoJohson, centrado y escalado

# Hago la transformacion en train para poner los datos con:
# media 0 y desviación 1 para la regresión logistica.
objTrans = preProcess(train, method = c("center", "scale", "pca"))
# Obtengo el nuevo conjunto de datos
train = predict(objTrans, train)
test = predict(objTrans, test)

cat("Dim Train Tras PCA: ", dim(train), "\n")
```

```
## Dim Train Tras PCA:  3823 32
```

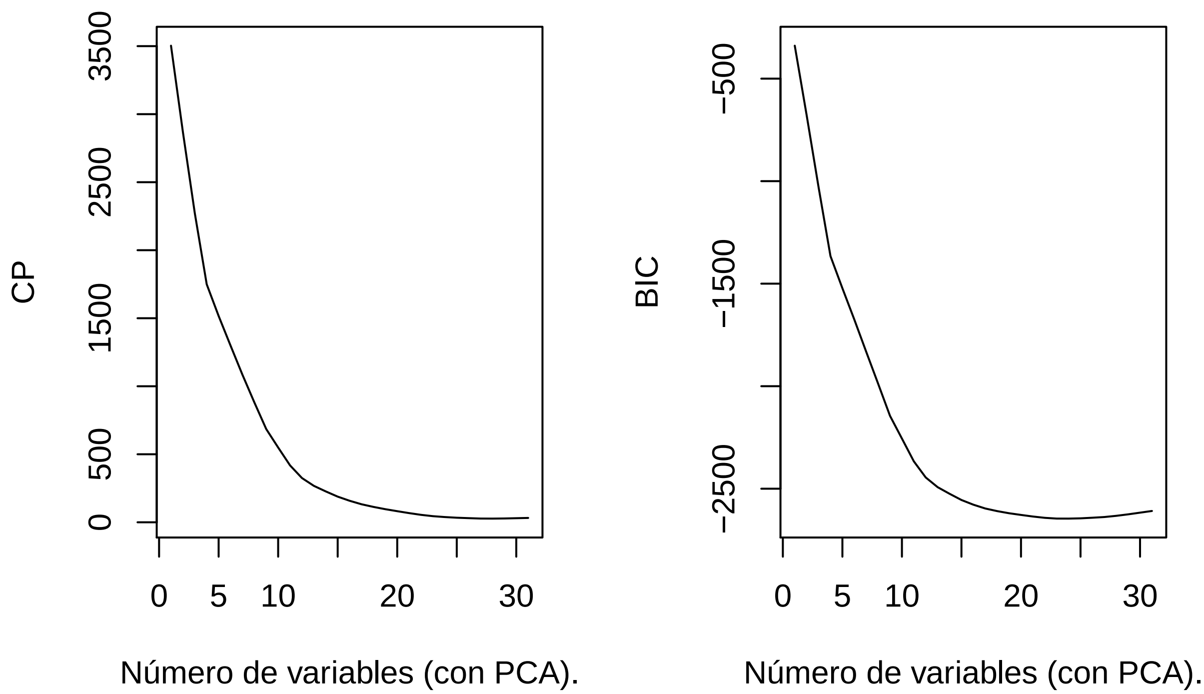
```r
cat("Dim Test  Sin  PCA: ", dim(test),  "\n")
```

```
## Dim Test  Sin  PCA:  1797 32
```

1.6. Definir los modelos a usar y estimar sus parámetros e hyperparámetros.

Los modelos que camos a usar los obtendremos de Regresion Logistica, Arboles de Regresión y SVM. Lo primero que con lo que vamos a seguir es a seleccionar el modelo con los datos optenidos anteriormente con el método greedy:

```r
# Dentro del PCA ahora vamos a ver cuales son las
# caracteristicas más relevantes según el train
subsetPCA = regsubsets(train$Y ~ . , data = train,
                       nvmax = dim(train)[2], method = "forward")
summaryPCA = summary(subsetPCA)

par(mfrow = c(1,2))

plot(summaryPCA$cp, xlab = "Número de variables (con PCA).",
     ylab = "CP", type = "l")
plot(summaryPCA$bic, xlab = "Número de variables (con PCA).",
     ylab = "BIC", type = "l")
```




```r
cat("Mejor número de características - CP (con PCA):",
    which.min(summaryPCA$cp), "\n")
```

```
## Mejor número de características - CP (con PCA): 28
```

```r
cat("Mejor número de características - BIC (con PCA):",
    which.min(summaryPCA$bic), "\n")
```

```
## Mejor número de características - BIC (con PCA): 24
```

Una vez sabemos los mejores datos que debemos usar elegimos los mejores datos para crear los modelos, en nuestro caso con 28 caracteristicas.

```r
# Usamos el nuevo train obtenido:
train = train[, c(1, as.vector(which(summaryPCA$outmat[28,]
                                     == "*")) + 1)]
test = test[, c(1, as.vector(which(summaryPCA$outmat[28,]
                                   == "*")) + 1)]
```

- Modelo con Regresión Logística

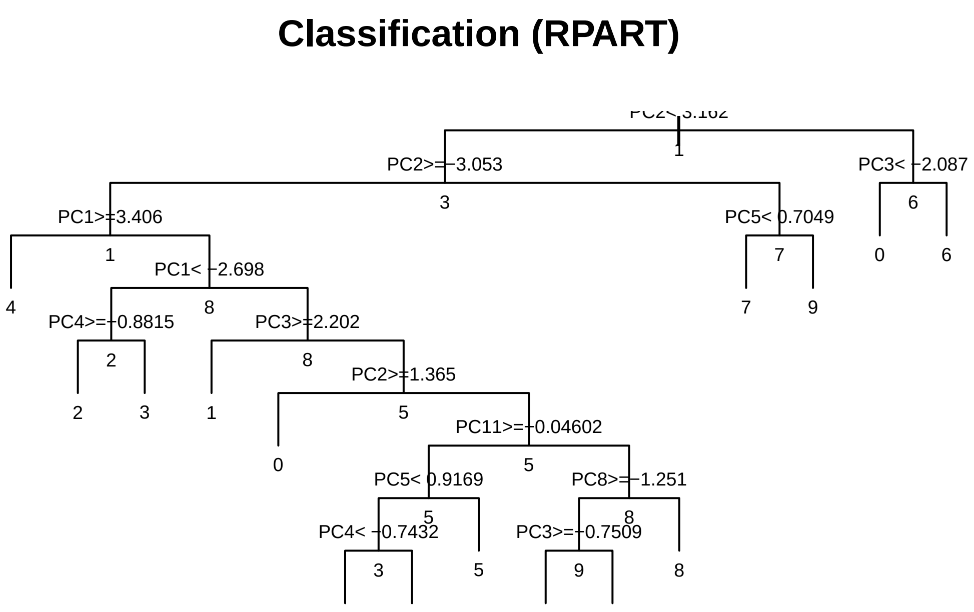
```
########################
## 1. Modelo Regresion Logistica
######################################################################
modelo.rgl <- multinom(train$Y ~ ., data=train)
```

```
## # weights:  300 (261 variable)
## initial  value 8802.782811
## iter  10 value 937.129500
## iter  20 value 627.555036
## iter  30 value 448.928221
## iter  40 value 365.043578
## iter  50 value 325.821227
## iter  60 value 290.954318
## iter  70 value 274.829687
## iter  80 value 248.294793
## iter  90 value 233.218363
## iter 100 value 225.675708
## final  value 225.675708
## stopped after 100 iterations
```

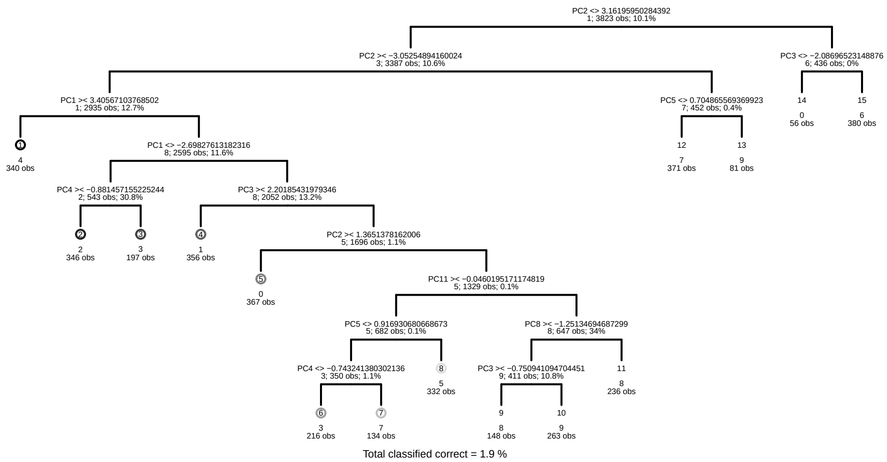
```
prediccion.rgl <- predict(modelo.rgl, newdata = train, type = "class")
error.rgl <- (sum(train$Y != prediccion.rgl)/nrow(train))
```

- Modelo RPART

```
########################
## 2. Modelo RPART
######################################################################
modelo.RPART <- rpart(train$Y ~ ., method="class",data = train)

# Pintamos los arboles
plot(modelo.RPART, uniform = TRUE, main = "Classification (RPART)")
text(modelo.RPART, all = TRUE, cex = 0.60)
```




```
draw.tree(modelo.RPART, cex = 0.25, nodeinfo = TRUE, col = gray(0:8/8))
```




```r
# Predecimos los datos sobre el test y los clasificamos.
prediccion.RPART <- predict(modelo.RPART, newdata = train, type = "class")
error.RPART <- (sum(train$Y != prediccion.RPART)/nrow(train))
```

- Modelo SVM:

```r
########################
## 3. Modelo con SVM
######################################################################
modelo.SVM <- svm(train$Y ~ ., method="class",data = train)
summary(modelo.SVM)
```

```
##
## Call:
## svm(formula = train$Y ~ ., data = train, method = "class")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.03571429
##
## Number of Support Vectors:  1688
##
##  ( 93 163 187 122 163 185 204 176 231 164 )
##
##
## Number of Classes:  10
##
## Levels:
##  0 1 2 3 4 5 6 7 8 9
```

```r
# Predecimos los datos sobre el train y los clasificamos.
prediccion.SVM <- predict(modelo.SVM, newdata = train, type = "class")

error.SVM <- (sum(train$Y != prediccion.SVM)/nrow(train))
```

Vamos a mostrar los resultados obtenidos

```r
cat("E_in  RGL: ", error.rgl, "\n")
```

```
## E_in  RGL:  0.01804865
```

```r
cat("Clasi RGL: ", (1-error.rgl)*100, "%\n\n")
```

```
## Clasi RGL:  98.19513 %
```

```r
cat("E_in  RPART: ", error.RPART, "\n")
```

```
## E_in  RPART:  0.2286163
```

```r
cat("Clasi RPART: ", (1-error.RPART)*100, "%\n\n")
```

```
## Clasi RPART:  77.13837 %
```

```r
cat("Error SVM: ", error.SVM, "\n")
```

```
## Error SVM:  0.002877321
```

```r
cat("Clasi SVM: ", (1-error.SVM)*100, "%\n")
```

```
## Clasi SVM:  99.71227 %
```

1.7. Selección y ajuste modelo final.

El modelo no lineal final será el basado en SVM ya que el error Ein es muy bajo. Cuando calculemos el Eout veremos si necesitamos ajustar más el modelo o si hemos hecho sobreaprendizaje, aún así como modelo lineal final usariamos regresión logística:

1.8. Estimacion del error Eout del modelo lo más ajustada posible.

```r
# Predecimos los datos sobre el test y los clasificamos.
prediccion.SVM_test <- predict(modelo.SVM, newdata = test, type = "class")

error.SVM <- (sum(test$Y != prediccion.SVM_test)/nrow(test))
cat("Eout SVM: ", error.SVM, "\n")
```

```
## Eout SVM:  0.02448525
```

```r
cat("Clasi: ", (1-error.SVM)*100, "%\n")
```

```
## Clasi:  97.55147 %
```

```r
# Tambiem probamos los de la RGL porque se basa en el ajuste de modelos
# lineales como la regresion logistica.

prediccion.RGL_test <- predict(modelo.rgl, newdata = test, type = "class")

error.RGL <- (sum(test$Y != prediccion.RGL_test)/nrow(test))
cat("Eout RGL: ", error.RGL, "\n")
```

```
## Eout RGL:  0.06455203
```

```r
cat("Clasi: ", (1-error.RGL)*100, "%\n")
```

```
## Clasi:  93.5448 %
```

Como podemos ver el error en la clasificación ha sido despreciable, no hemos sobreaprendido por lo que no realizaremos nungún ajuste más en el modelo.

1.10. Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.

La calidad del modelo usado es muy buena, obtenemos valores de clasificación muy altos tanto con el conjunto test como train por lo que la ideonidad de este modelo para este problema es muy buena aunque el modelo no lineal es mejor que el lineal.

---

2. Ajustes de Modelos Lineales (Regresion Airfoil Self-Noise)

2.1. Comprender el problema a resolver.

El problema sobre Self-Noise consiste en un problema de regresión donde deberemos de estimar ruido producido sobre un material ante unas determinadas condiciones del entorno. Este problema es un problema de Regresión.

2.2. Preprocesado los datos: por ejemplo categorización, normalización, etc.

Vamos leer los datos y abuscar la vatiables altamente correlacionadas.

```r
datos <-  read.csv("./datos/airfoil_self_noise.csv", header=F)
colnames(datos) = c("Hz","ANG","Long","vFlujo","espesor","ruido")

# Correlacion de los datos
correlacion <- cor(datos)
summary(correlacion[upper.tri(correlacion)])
```

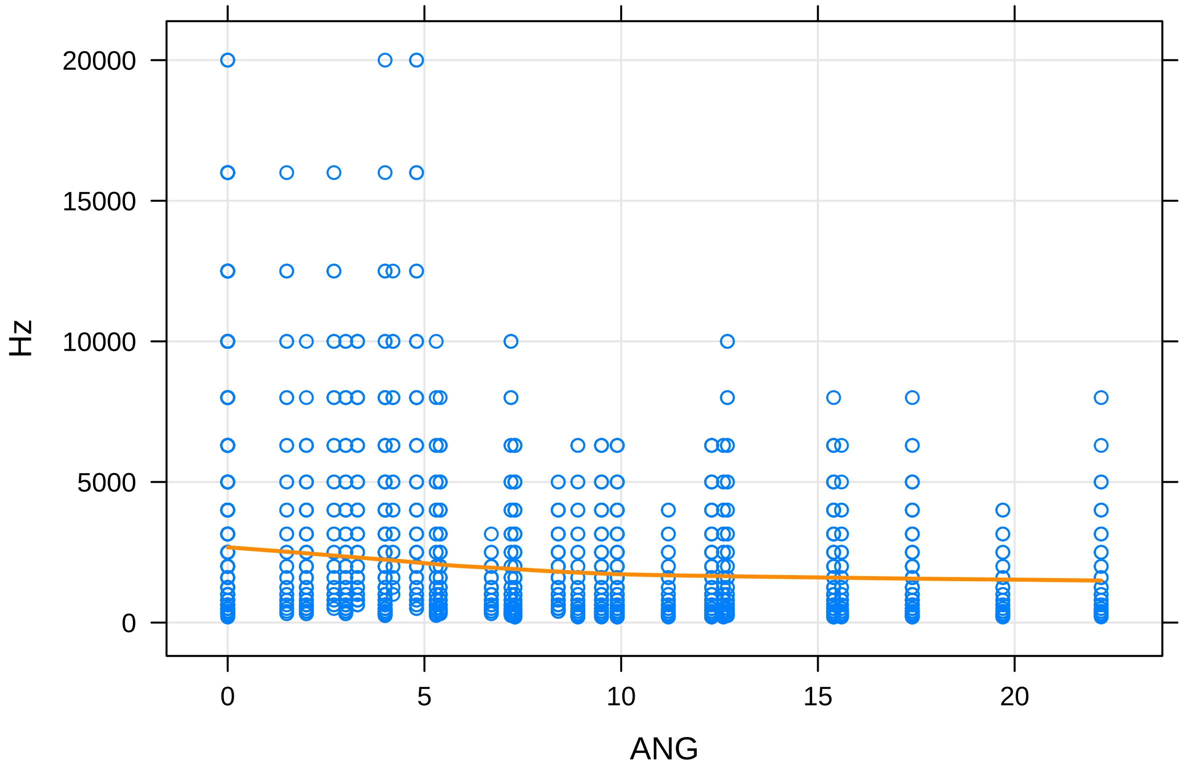
```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.50487 -0.25446 -0.15611 -0.08381  0.03127  0.75339
```

```r
# Buscamos var correlada
correlacion.alta <- findCorrelation(correlacion, cutoff = .75)
sum(correlacion.alta)
```

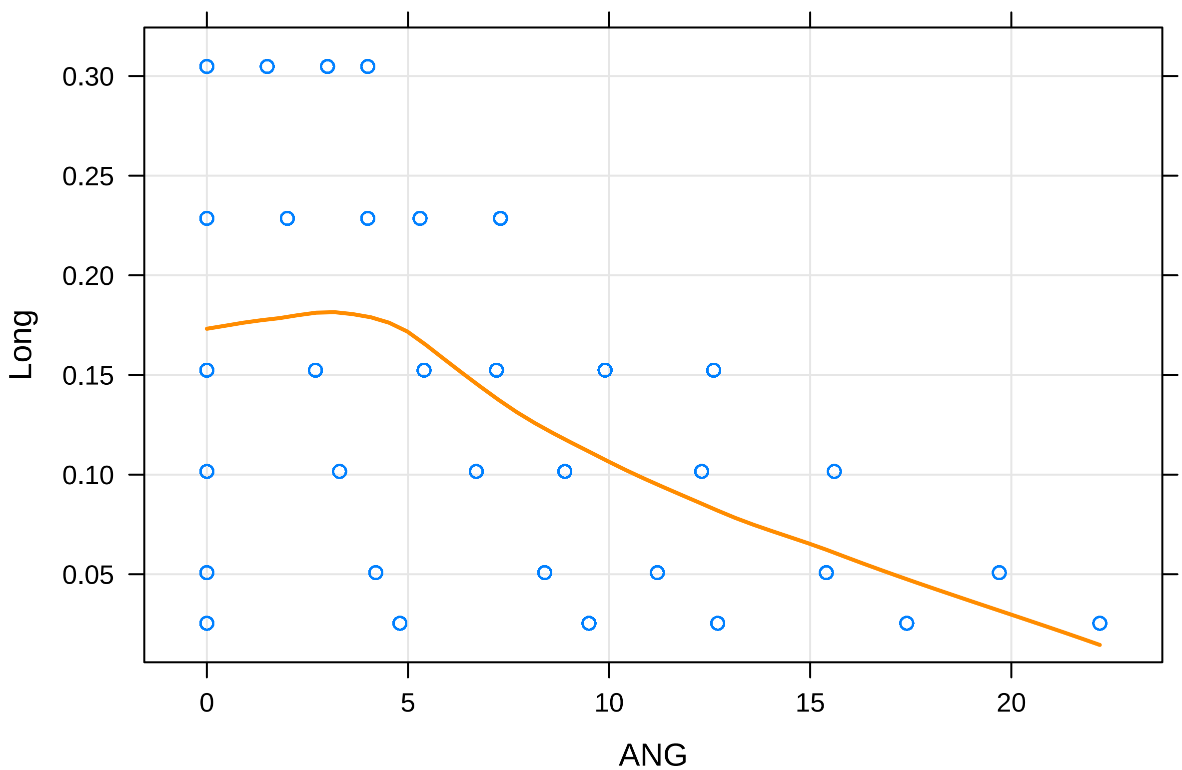
```
## [1] 2
```

Como vemos la variable es la dos asi que la comparamos con todas para ver cuales son las dos que lo están:

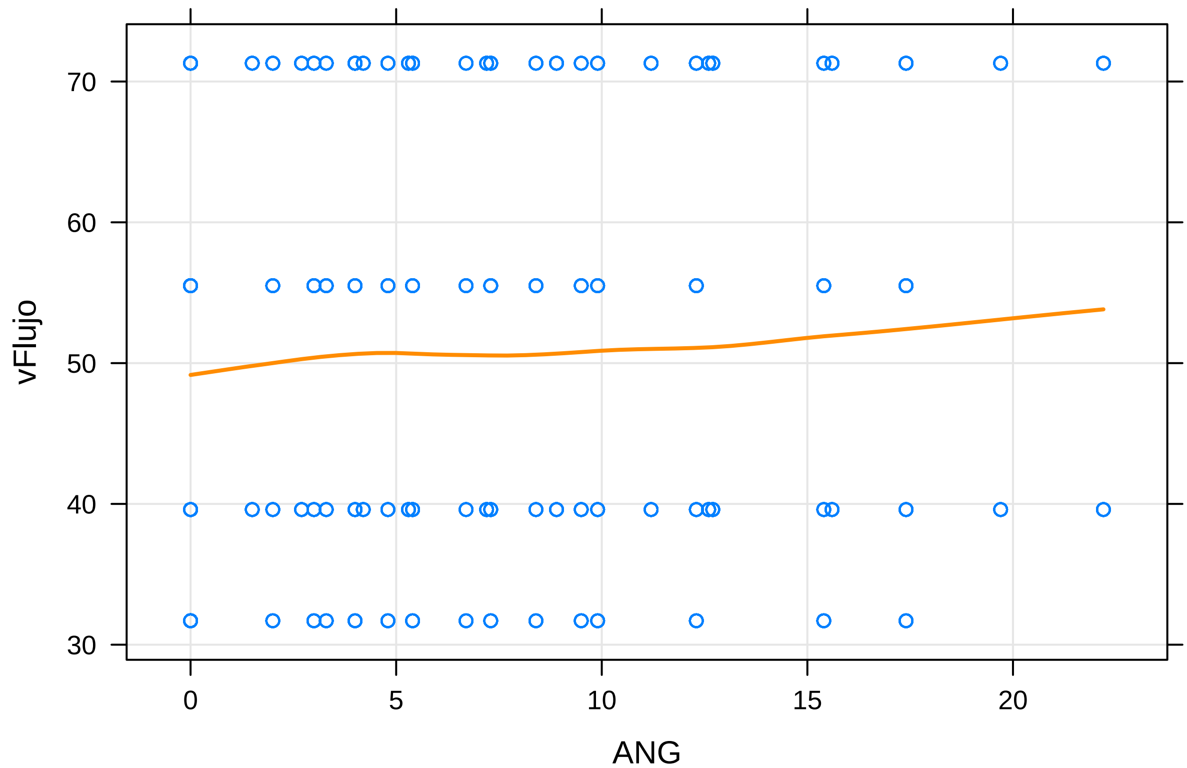
```r
par(mfrow = c(1,2))
xyplot(Hz~ANG,datos,grid=T,type = c("p", "smooth"),, col.line = "darkorange",lwd = 2)
```



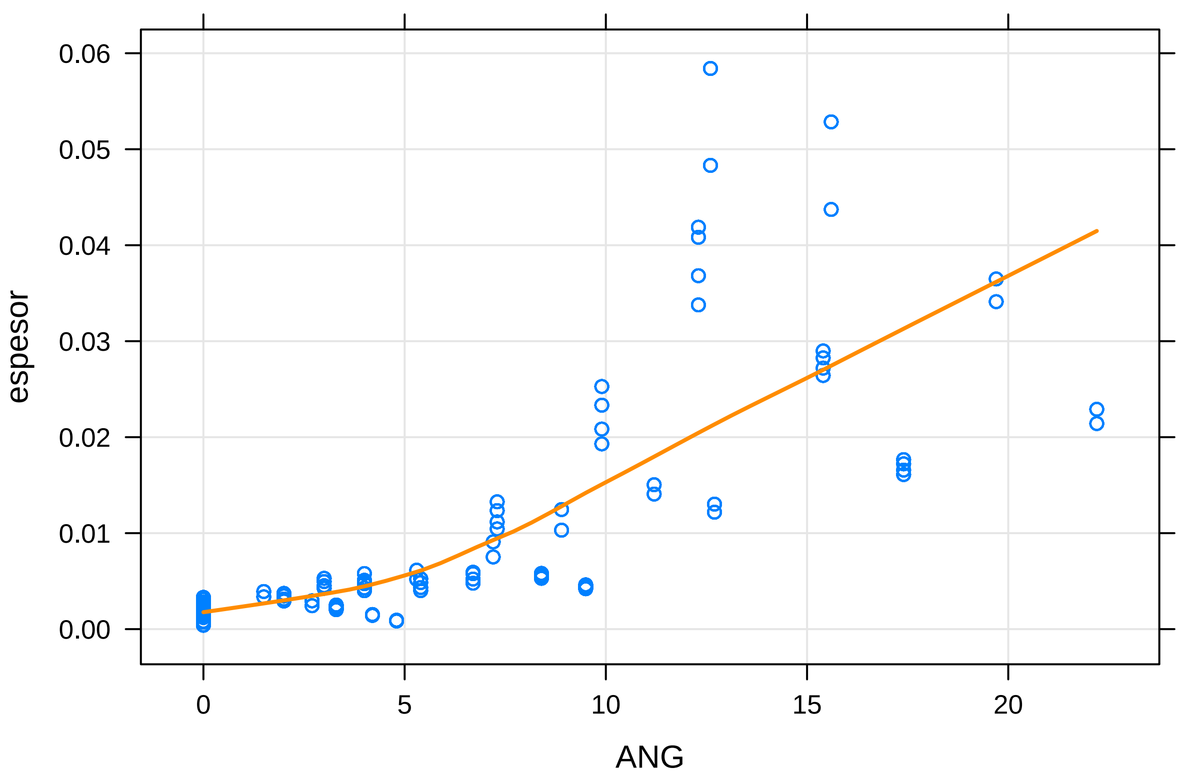

```r
xyplot(Long~ANG,datos,grid=T,type = c("p", "smooth"),, col.line = "darkorange",lwd = 2)
```




```r
xyplot(vFlujo~ANG,datos,grid=T,type = c("p", "smooth"),, col.line = "darkorange",lwd = 2)
```




```
xyplot(espesor~ANG,datos,grid=T,type = c("p", "smooth"),, col.line = "darkorange",lwd = 2)
```




```
# Sacrificamos la caracteristica espesor:
datos = datos[,-5]
```

Además dos de las variables de entrada son: velocidad de flujo y longitud de la cuerda que pueden resumirse en una caracteristica denominada viscosidad cinematica cuya formula es: Re = V * C / nu, donde nu = 1.568e-5

```
# Reducimos una caracteristica que se puede traducir
# por la viscosidad cinemática
nu <- 1.568e-5

datos$Re <- datos$Long * datos$vFlujo / nu
datos <- datos[,-c(3,4)]
datos <- datos[,c(1:2,4,3)]

names(datos)
```

```
## [1] "Hz"    "ANG"   "Re"    "ruido"
```

Además eliminamos la caracteristicas que pueden producir varianza cercana a 0.

```
# Eliminamos las variables muy cercanas a 0 como antes:
nvz <- nearZeroVar(datos, saveMetrics = TRUE)
nvz
```

```
##       freqRatio percentUnique zeroVar   nzv
## Hz     1.009615      1.397206   FALSE FALSE
## ANG    3.537634      1.796407   FALSE FALSE
## Re     1.022222      1.596806   FALSE FALSE
## ruido  1.000000     96.872921   FALSE FALSE
```

```
datos <- datos[,!nvz$nzv]
```

2.3. Selección de clases de funciones a usar.

La clase de funciones a usar de nuevo van a saer Lineales en este cado usaremos Regresión Lineal, además usaremos no lineales: Arboles, Boostin, Bagging y Random Forest.

2.4. Definición de los conjuntos de training, validación y test usados en su caso.

Los datos no vienen separados en conjuntos por lo que los tendremos que separar nosotros mismos de forma manual, para ello haremos uso de del función createDataPartition(). Dividimos en dos conjuntos, train y test, donde train representa el 70% de los datos y el test el 30%.

```
# Partimos el cnj en el 75% de los datos para train y 25 para el test:
cat("Dim Datos: ", dim(datos), "\n")
```

```
## Dim Datos:  1503 4
```

```
enEntrenamiento <- createDataPartition(y=datos$ruido, p=0.70, list = FALSE)

# Guardamos test y train
train <- datos[enEntrenamiento,]
test  <- datos[-enEntrenamiento,]
cat("Dim Train: ", dim(train), "\n")
```

```
## Dim Train:  1055 4
```

```
cat("Dim Test: ", dim(test), "\n")
```

```
## Dim Test:  448 4
```

```
train_o = train
test_o = test
```

2.5. Discutir la necesidad de regularización y en su caso la función usada para ello.

La regularización de los datos no la realizaremos a no ser que sea necesario, cuando estimemos el Einy lo comparemos en el Eout, en caso de ver overfiting.

2.6. Definir los modelos a usar y estimar sus parámetros e hyperparámetros.

Los modelos a usar serán los lineales, en concreto Regresión Lineal, además crearemos 3 modelos más basados en Arboles de regresión, Boosting, Bagging y RandomForest.

- Regresion lineal

```
######################################
##  Modelo 1 Regresion lineal
###################################################

modelo.lm = lm(train_o$ruido ~ ., data=train_o)

train_o = train
test_o = test
```

- Arboles de regresion

```
#################################
## Modelo 2 Arboles de regresion
#######################################################

modelo.tree = rpart(train_o$ruido ~ ., data=train_o)

train_o = train
test_o = test
```

- Boostin caret

```
###########################
## Modelo 3 Boostin
#######################################################
control = trainControl(method = "cv", number = 5, verboseIter=F)
gbmGrid <- expand.grid(interaction.depth = c(20,30,45), n.trees = 500,
                       shrinkage = .1, n.minobsinnode = 10)

#modelo.gbm = train(train_o$ruido ~ ., data = train_o, method = "gbm",
#                  trControl = control, verbose = FALSE, tuneGrid = gbmGrid)

#plot(modelo.gbm)

#res <- eval_model(modelo.gbm)
#cat("Ein : ", res[1], "\n")
#cat("Eout : ", res[2], "\n\n")
```

```
train_o = train
test_o = test
```

- Bagging

```
###########################
## Modelo 4 Bagging- Caret
#######################################################
modelo.bag <- bag(train_o[,-7],train_o$ruido, B = 10,
               bagControl = bagControl(fit = ctreeBag$fit,
                                       predict = ctreeBag$pred,
                                       aggregate = ctreeBag$aggregate))
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

```
train_o = train
test_o = test
```

- Random Forest:

```
#########################################
## Modelo 5 RandomFores
######################################################
modelo.rf <- train(x=train[,-7],y=train$ruido,method="rf",
                trControl=trainControl(method = "cv", number = 4),
                data=train,do.trace=F,ntree=250)
train_o = train
test_o = test
```

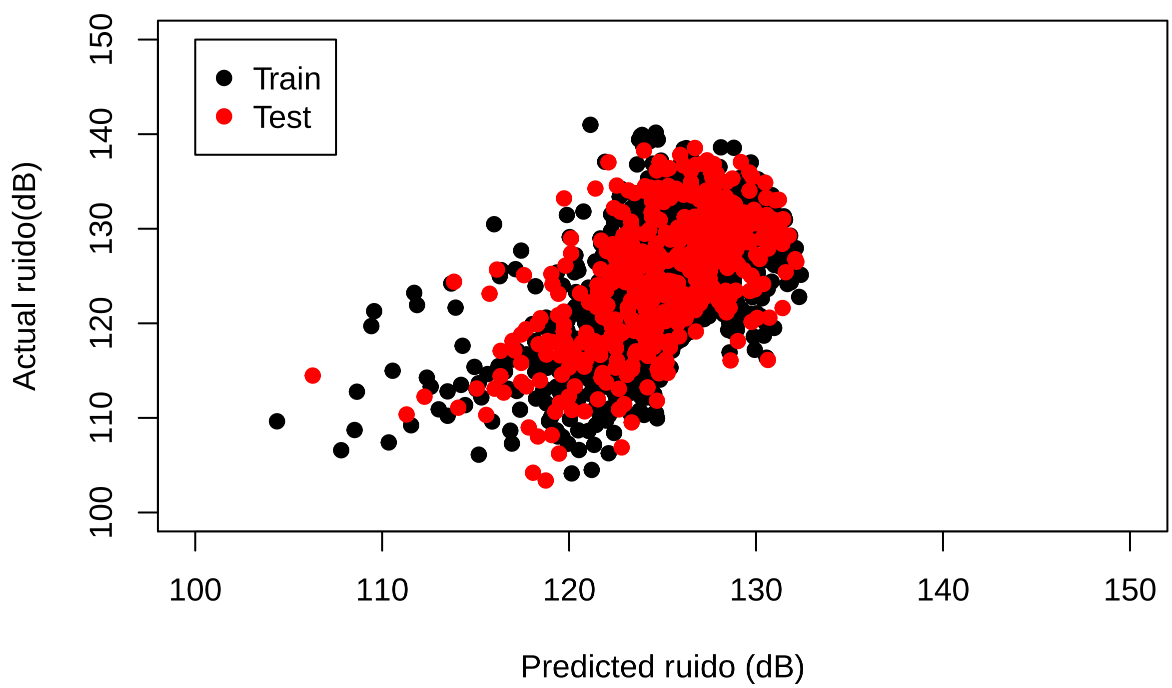
Usamos la función eval_model para obtener las graficas para comparar el train y test con las predicciones realizadas por el modelo:

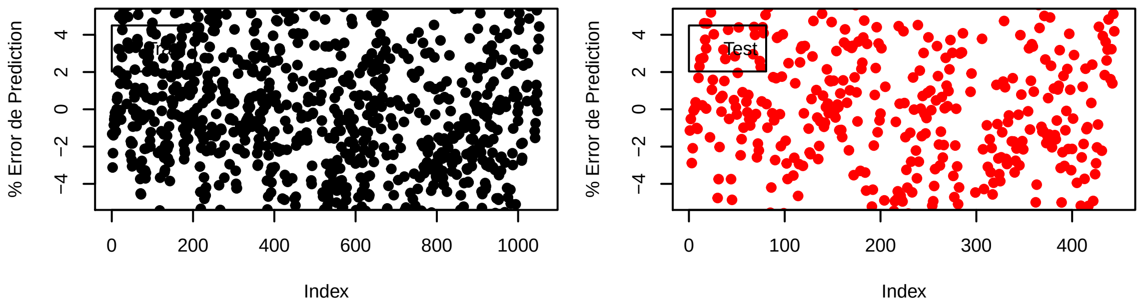
```
# Función para pintar graficas usando un modelo ante los datos train
# y test esta funcion ha sido encontrada en internet, en stackoverflow
eval_model <- function(model) {

  pred_train <- predict(model,newdata = train)
  pred_test <- predict(model,newdata = test)

  # Scatter plots of predictions on Train and test sets
  plot(pred_train,train$ruido,xlim=c(100,150),ylim=c(100,150),col=1,
       pch=19,xlab = "Predicted ruido (dB)",ylab = "Actual ruido(dB)")
  points(pred_test,test$ruido,col=2,pch=19)
  leg <- c("Train","Test")
  legend(100, 150, leg, col = c(1, 2),pch=c(19,19))

  # Scatter plots of % error on predictions on Train and test sets
  par(mfrow = c(2, 2))
  par(cex = 0.6)
  par(mar = c(5, 5, 3, 0), oma = c(2, 2, 2, 2))
  plot((pred_train - train$ruido)* 100 /train$ruido,
       ylab = "% Error de Prediction", xlab = "Index",
       ylim = c(-5,5),col=1,pch=19)
  legend(0, 4.5, "Train", col = 1,pch=19)
  plot((pred_test-test$ruido)* 100 /test$ruido,
       ylab = "% Error de Prediction",  xlab = "Index",
       ylim = c(-5,5),col=2,pch=19)
```

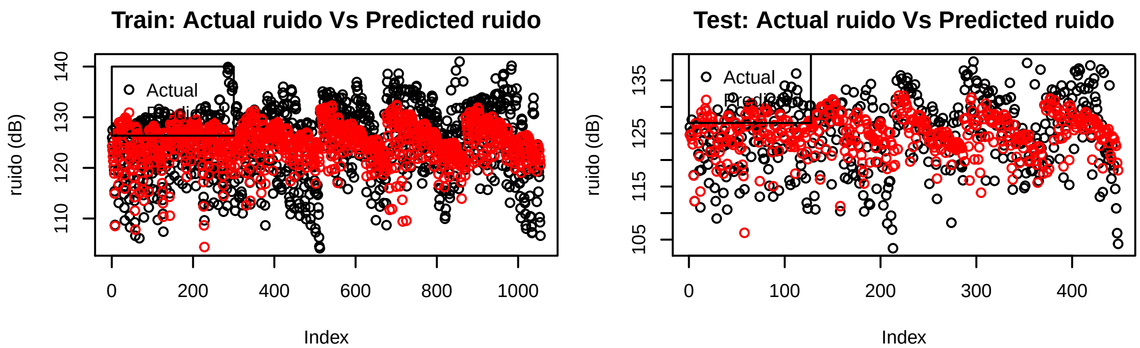
```r
  legend(0, 4.5, "Test", col = 2,pch=19)

  # Actual data Vs Predictions superimposed for Train and test Data
  plot(1:length(train$ruido),train$ruido,pch=21,col=1,
       main = "Train: Actual ruido Vs Predicted ruido",
       xlab = "Index",ylab = "ruido (dB)")
  points(1:length(train$ruido),pred_train,pch=21,col=2)
  #leg <- c("Train","Predicted Train")
  legend(0, 140, c("Actual","Predicted"), col = c(1, 2),pch=c(21,21))
  plot(1:length(test$ruido),test$ruido,pch=21,col=1,
       main = "Test: Actual ruido Vs Predicted ruido",
       xlab = "Index",ylab = "ruido (dB)")
  points(1:length(test$ruido),pred_test,pch=21,col="red")
  legend(0, 140, c("Actual","Predicted"), col = c(1, 2),pch=c(21,21))

  ## Line graph de errors
  plot(pred_train-train$ruido,type='l',ylim=c(-5,+5),
       xlab = "Index",ylab = "Actual - Predicted",main="Train")
  plot(pred_test-test$ruido,type='l',ylim=c(-5,+5),
       xlab = "Index",ylab = "Actual - Predicted",main="Test")

  ISRMSE<- sqrt(mean((pred_train-train$ruido)^2))
  OSRMSE<- sqrt(mean((pred_test-test$ruido)^2))

  return(c( ISRMSE,OSRMSE))
}
```

Obtenemos los Ein para ver cual es el mejor modelo a usar:

```r
# Evaluamos
res <- eval_model(modelo.lm)
```



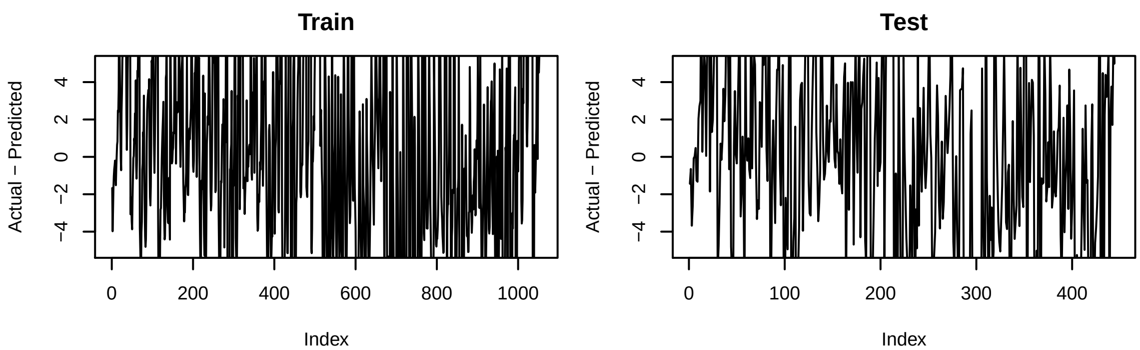




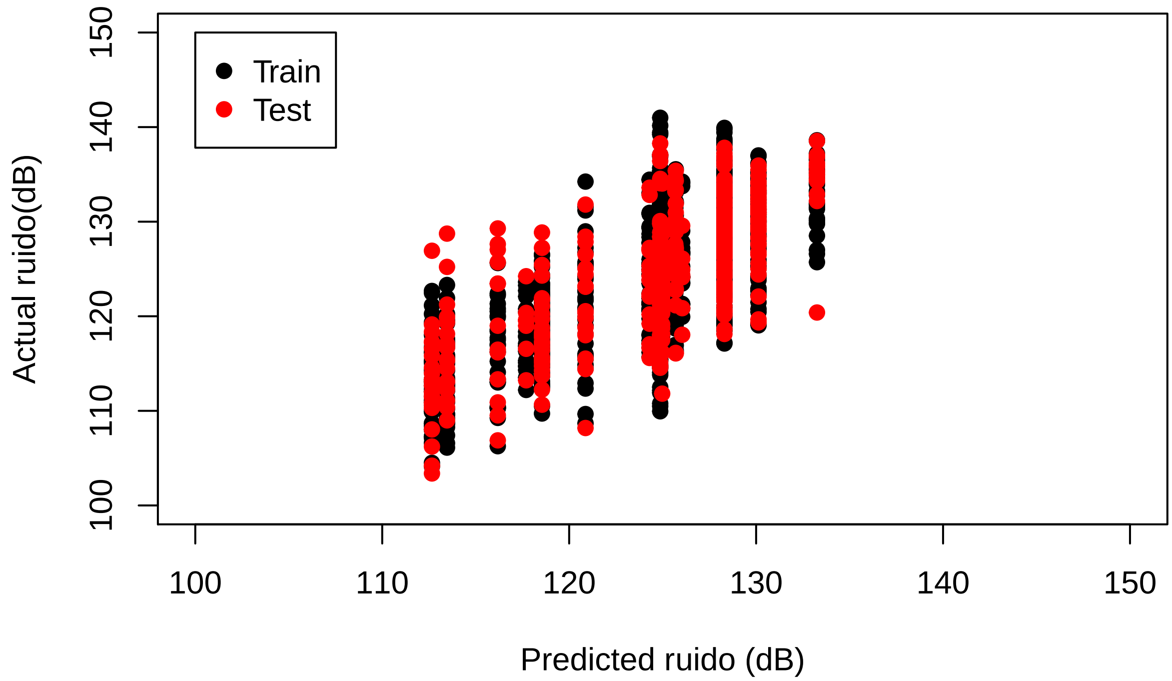

```r
cat("Ein - LM      :", res[1], "\n")
```

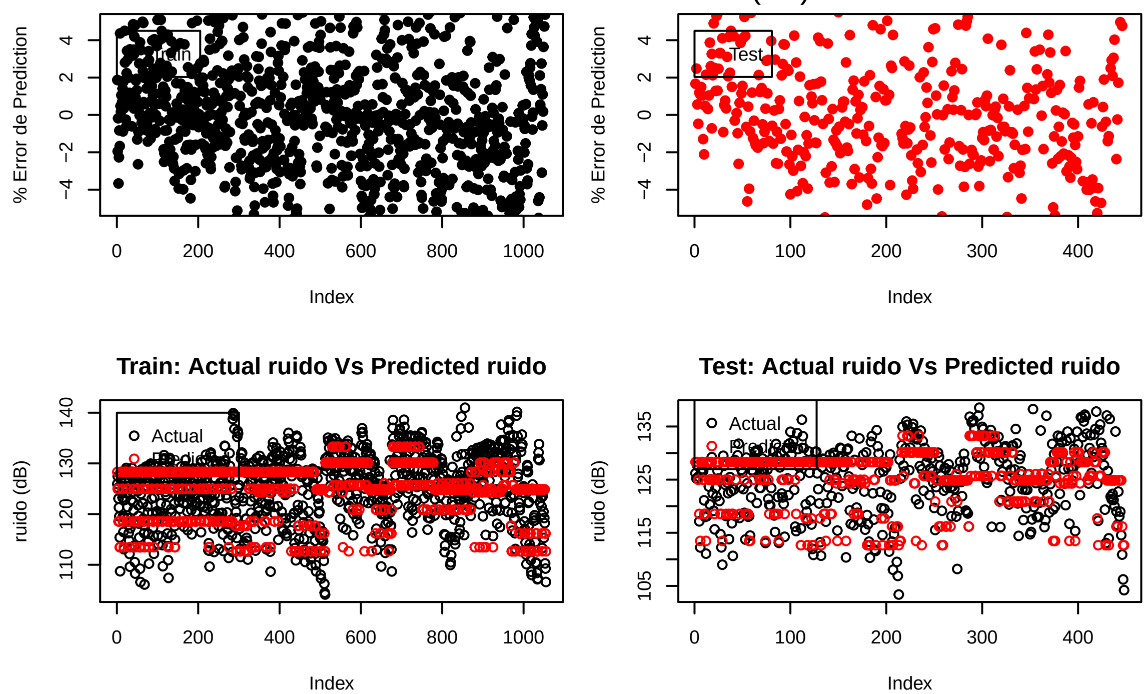
```
## Ein - LM      : 5.80312
```

```r
lm.Eout = res[2]
```




```r
# Evaluamos
res <- eval_model(modelo.tree)
```

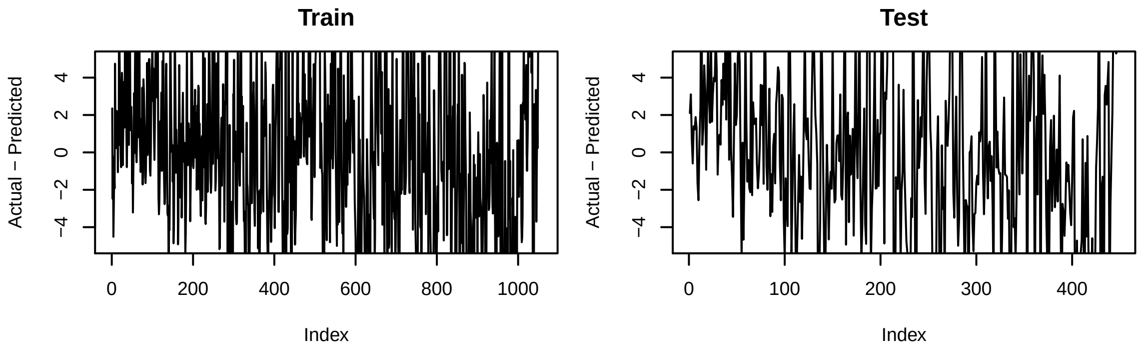




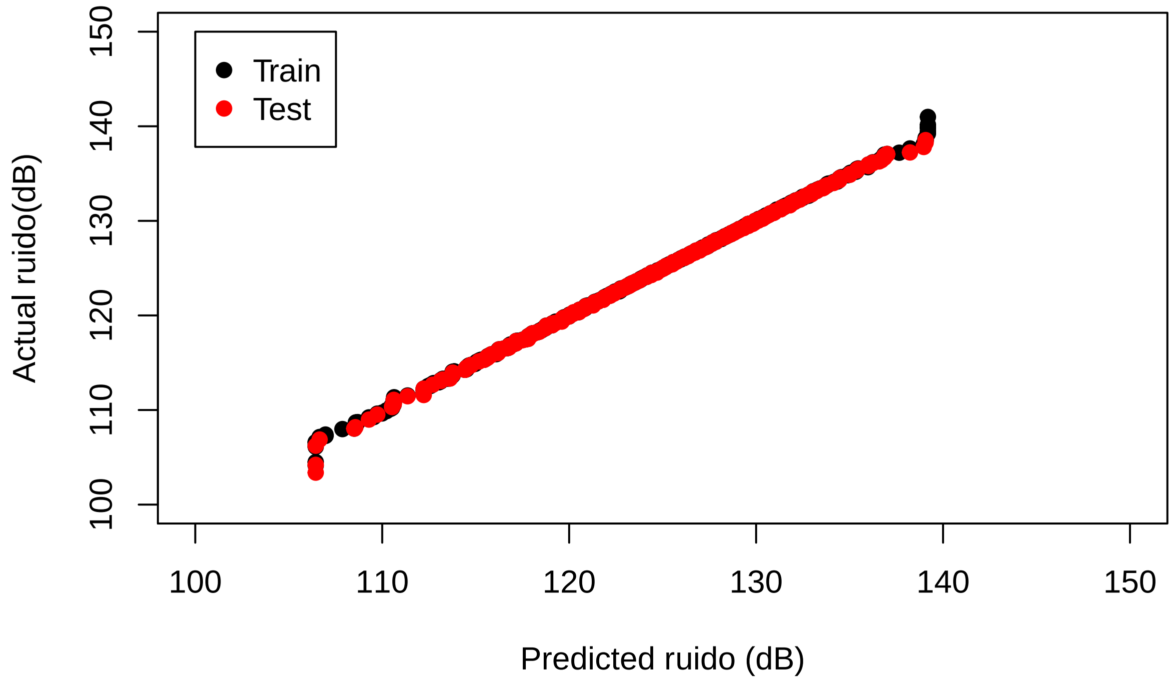

```
cat("Ein - Tree   :", res[1], "\n")
```

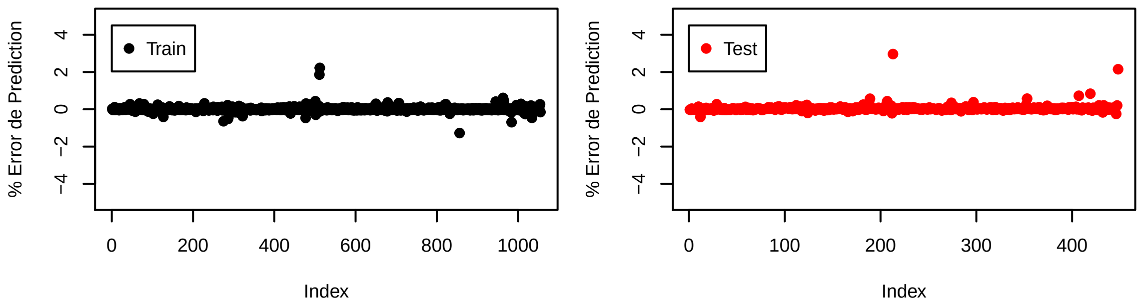
```
## Ein - Tree   : 4.631141
```

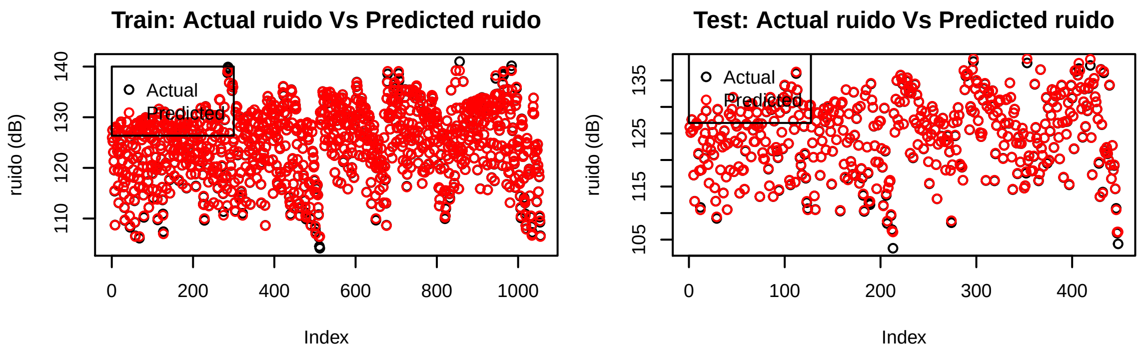
```
tree.Eout = res[2]
```




```r
# Evaluamos
res <- eval_model(modelo.bag)
```



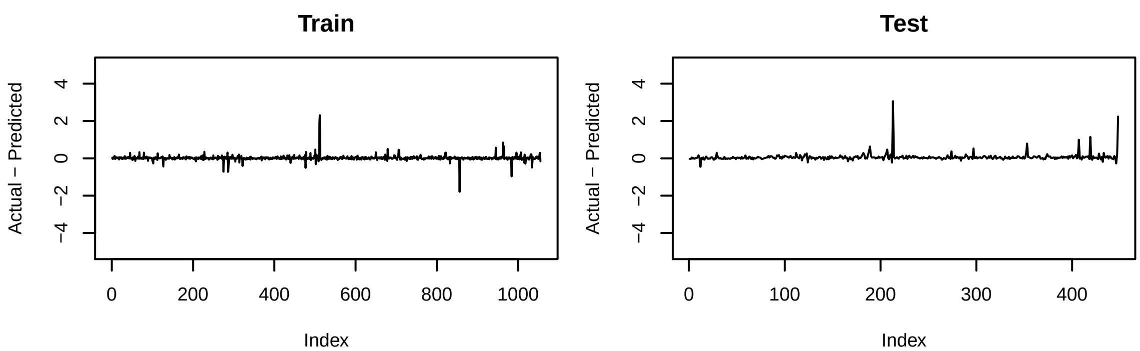




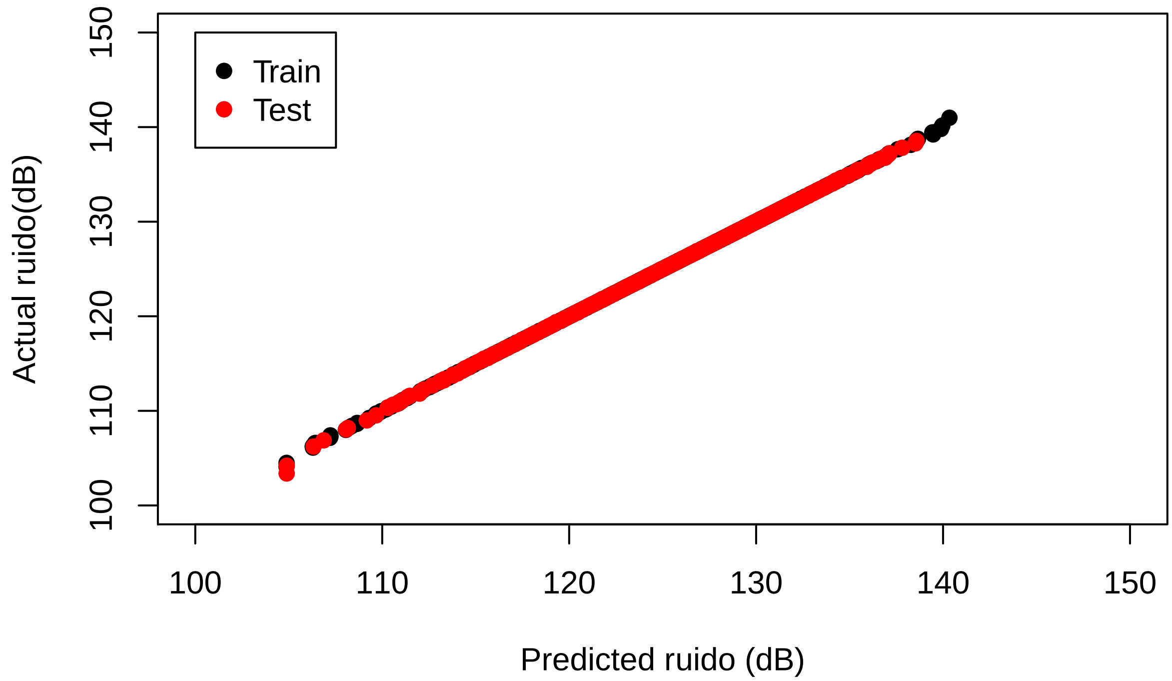

```r
cat("Ein - BAG    :", res[1], "\n")
```

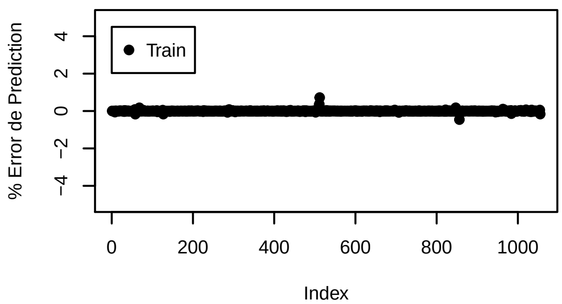
```
## Ein - BAG    : 0.1494231
```

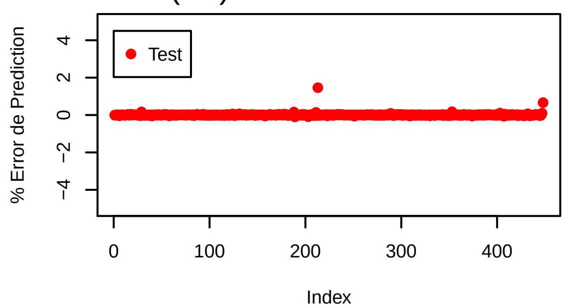
```r
bag.Eout = res[2]
```

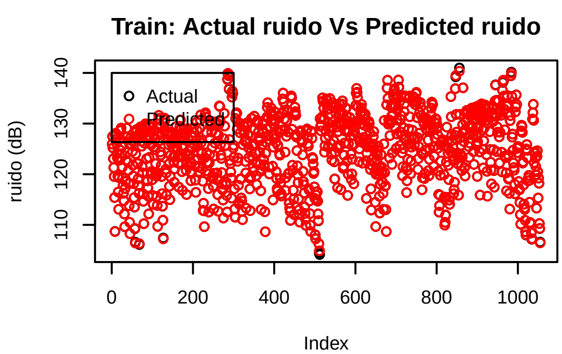


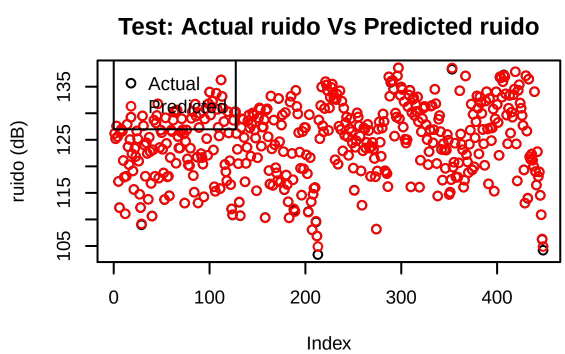

```r
# Evaluamos
res <- eval_model(modelo.rf)
```







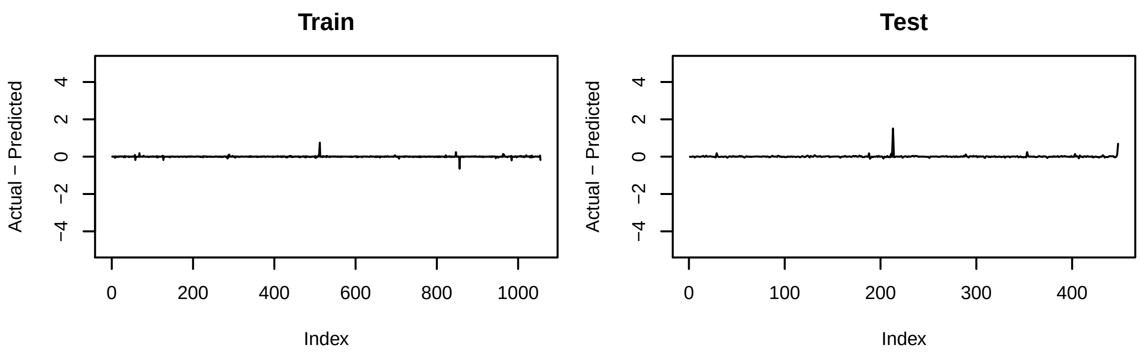





```r
cat("Ein - RF       :", res[1], "\n")
```

```
## Ein - RF       : 0.03947389
```

```r
rf.Eout = res[2]
```



2.7. Selección y ajuste modelo final.

Según los valores obtenidos en el Ein el mejor modelo es el de Random Forest pero aún así veremos los resultados Eout de los otros modelos.

2.8. Estimacion del error Eout del modelo lo más ajustada posible.

El que produce menor error es RandomForest que posee un error casi igual que el Ein por lo que entendemos que el modelo no está sobreajustado y no lo ajustaremos más pues hace muy buenas predicciones.

```
# Mostramos el Eout de los modelos.
cat("Eout - LM  : ", lm.Eout, "\n")
```

```
## Eout - LM  :  5.608886
```

```
cat("Eout - Tree: ", tree.Eout, "\n")
```

```
## Eout - Tree:  4.843663
```

```
cat("Eout - Bag  : ", bag.Eout, "\n")
```

```
## Eout - Bag  :  0.2208917
```

```
cat("Eout - RF  : ", rf.Eout, "\n")
```

```
## Eout - RF  :  0.08424525
```

2.9. Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.

El modelo se ajusta perfectamente a los datos. Con el Ein podríamos haber pensado un sobreajuste pero al hacer el Eout se ve que el modelo se ajusta a la perfección.