Proyecto "Parkinson Telemonitoring"

Antonio Miguel Morillo Chica

30/05/2018

1. Parkinson Telemonitoring

1.1. Entender el problema a resolver.

El proyecto que voy ha desarrollar usa la base de datos "Parkinson Telemonitoring". Esta base de datos se compone de una gama de mediciones de voz biomédica de 42 personas con enfermedad de Parkinson en etapa temprana contratadas para una prueba de seis meses de un dispositivo de telemonitorización para la monitorización remota de la progresión de los síntomas. Las grabaciones fueron capturadas automáticamente en el hogar del paciente.

Los datos están en formato ASCII-CSV. Las filas del archivo CSV contienen una instancia correspondiente a una grabación de voz. Hay alrededor de 200 grabaciones por paciente, el número de sujeto del paciente se identifica en la primera columna.

El objetivo principal será predecir los valores de motor_UPDRS y total_UPDRS.

1.2. Lectura de los datos.

La base de datos está compuesta por un archivo donde encontramos 5879 ejemplos con 26 columnas, la primera identifica al paciente y el resto corresponde a 25 caracteristicas que son:

- **Sexo**: 0 es hombre, 1 es mujer.
- **test_time** - Tiempo del reclutamiento, la parte entera representa el número de dias.
- **motor_UPDRS** - Escala de puntuación UDPRS (Unified Parkinson's Disease Rating Scale) **que deberemos de pronosticar**
- **total_UPDRS** - Escala de puntuación UDPRS (Unified Parkinson's Disease Rating Scale) **que deberemos de pronosticar**
- **Jitter(%),Jitter(Abs),Jitter:RAP,Jitter:PPQ5,Jitter:DDP** - Varias medidas de variación en la frecuencia.
- **Shimmer,Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,Shimmer:APQ11,Shimmer:DDA** - Varias medidas de variación en la amplitud
- **NHR,HNR** - Dos medidas de proporción de ruido a componentes tonales en la voz.
- **RPDE** - Una medida de complejidad dinámica no lineal
- **DFA** - Exponente de escala de fractal de señal
- **PPE** - Una medida no lineal de la variación de frecuencia fundamental.

Lo primero que haremos será leer el conjunto del archivo csv, como este ya contiene una fila con los nombres por columna, lo leemos, de igual forma sabemos que el separador de datos de una misma fila es una coma por lo que lo indicamos aunque no haría falta.

```
datos <- read.csv("./datos/parkinsons_updrs.data", head=T, sep = ",")
```

Como los datos de partida están contenidos en el mismo archivo lo que vamos a hacer es dividir el conjunto de datos en dos, un conjunto train que nos servirá para entrenar los modelos y un train para probar la efectividad de nuestro aprendizaje. De igual forma he de recalcar que no he realizado validación cruzada aunque para obtener unos resultados más "reales" esto sería muy conveniente.

Por otro lado he decidido eliminar 4 caracteristicas que no me parecen relevantes, aunque más que no ser relevantes me voy a basar unicamente en las caracteristicas de la voz sin tener en cuenta el sexo.

```
# Eliminamos: sujeto, edad, sexo y test_time:
datos <- datos[,-(1:4)]
summary(datos)
```

```
##   motor_UPDRS      total_UPDRS      Jitter...         Jitter.Abs.
##  Min.   : 5.038   Min.   : 7.00   Min.   :0.000830   Min.   :2.250e-06
##  1st Qu.:15.000   1st Qu.:21.37   1st Qu.:0.003580   1st Qu.:2.244e-05
##  Median :20.871   Median :27.58   Median :0.004900   Median :3.453e-05
##  Mean   :21.296   Mean   :29.02   Mean   :0.006154   Mean   :4.403e-05
##  3rd Qu.:27.596   3rd Qu.:36.40   3rd Qu.:0.006800   3rd Qu.:5.333e-05
##  Max.   :39.511   Max.   :54.99   Max.   :0.099990   Max.   :4.456e-04
##    Jitter.RAP        Jitter.PPQ5         Jitter.DDP
##  Min.   :0.000330   Min.   :0.000430   Min.   :0.000980
##  1st Qu.:0.001580   1st Qu.:0.001820   1st Qu.:0.004730
##  Median :0.002250   Median :0.002490   Median :0.006750
##  Mean   :0.002987   Mean   :0.003277   Mean   :0.008962
##  3rd Qu.:0.003290   3rd Qu.:0.003460   3rd Qu.:0.009870
##  Max.   :0.057540   Max.   :0.069560   Max.   :0.172630
##     Shimmer          Shimmer.dB.      Shimmer.APQ3      Shimmer.APQ5
##  Min.   :0.00306   Min.   :0.026   Min.   :0.00161   Min.   :0.00194
##  1st Qu.:0.01912   1st Qu.:0.175   1st Qu.:0.00928   1st Qu.:0.01079
##  Median :0.02751   Median :0.253   Median :0.01370   Median :0.01594
##  Mean   :0.03404   Mean   :0.311   Mean   :0.01716   Mean   :0.02014
##  3rd Qu.:0.03975   3rd Qu.:0.365   3rd Qu.:0.02057   3rd Qu.:0.02375
##  Max.   :0.26863   Max.   :2.107   Max.   :0.16267   Max.   :0.16702
##  Shimmer.APQ11      Shimmer.DDA           NHR                HNR
##  Min.   :0.00249   Min.   :0.00484   Min.   :0.000286   Min.   : 1.659
##  1st Qu.:0.01566   1st Qu.:0.02783   1st Qu.:0.010955   1st Qu.:19.406
##  Median :0.02271   Median :0.04111   Median :0.018448   Median :21.920
##  Mean   :0.02748   Mean   :0.05147   Mean   :0.032120   Mean   :21.680
##  3rd Qu.:0.03272   3rd Qu.:0.06173   3rd Qu.:0.031463   3rd Qu.:24.444
##  Max.   :0.27546   Max.   :0.48802   Max.   :0.748260   Max.   :37.875
##       RPDE              DFA               PPE
##  Min.   :0.1510   Min.   :0.5140   Min.   :0.02198
##  1st Qu.:0.4698   1st Qu.:0.5962   1st Qu.:0.15634
##  Median :0.5423   Median :0.6436   Median :0.20550
##  Mean   :0.5415   Mean   :0.6532   Mean   :0.21959
##  3rd Qu.:0.6140   3rd Qu.:0.7113   3rd Qu.:0.26449
##  Max.   :0.9661   Max.   :0.8656   Max.   :0.73173
```

```
# Elegimos los indices para el train, 80% de los datos
idx.Train = sample(nrow(datos),size = nrow(datos)*0.8)

# Formamos loc conjuntos train y test.
train <- as.data.frame(datos[idx.Train,])
test  <- as.data.frame(datos[-idx.Train,])

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

```
## Dim train :  4700 18
```

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

```
## Dim test  :  1175 18
```

```r
Sys.sleep(3)
```

1.3. Tratamiento basico de los datos (normalización)

Para este apartado haremos dos tareas muy sencillas y una tercera que dejaremos para más delante. La primera es averiguar variables que provoquen que la varianza sea cercana a 0 cosa que no nos interesa ya extraeremos poca información de estas características y lo único que harán serán entorpecer al modelo de aprendizaje.

```r
nearZeroVar(train)
```

```
## integer(0)
```

```r
nearZeroVar(test)
```

```
## integer(0)
```

Como podemos ver en la salida nos dice que no existe ninguna caracteristica cuya varianza sea proxima a 0 por lo que asumismos que por ahora las caracteristicas usadas son buenas y variadas, serán buenos predictores. Ahora vamos a ver si existen valores perdidos, esto es, valores para características cuyo valor es Na.

```r
perdidos.train <- sum(is.na(train))
perdidos.test  <- sum(is.na(test ))

cat("Num valores perdidos: ", perdidos.train, ",", perdidos.test,"\n")
```

```
## Num valores perdidos:  0 , 0
```

```r
Sys.sleep(3)
```

Vemos como el número de valores perdidos es 0 por lo que no deberemos tratarlos.

El último punto de esta sección tendría que ver con la normalización pero no la aplicaremos ya que primero vamos a ver los resultados de los modelos sin que los datos estén preProcesados, pero, ¿es bueno hacer un preprocesado, normalizar etc.?

En el tratamiento y analisis masivo de datos nos encontramos todo el tiempo que datos sucios, ruido o valores inconsistentes por lo que obviamente cuanta mejor sea la calidad de las mediciones, no existan inconsistencias, como valores perdidos, caracteristicas que no aportan información alguna o información contradictoria (ej. sexo: hombre, embarazada: sí) lo mejor será resolverlas. Además, la normalización ayuda casi siempre al modelo ya que "elimina" valores extremos que pueden conducirnos a un mal aprendizaje.

1.5 Modelos que vamos a usar.

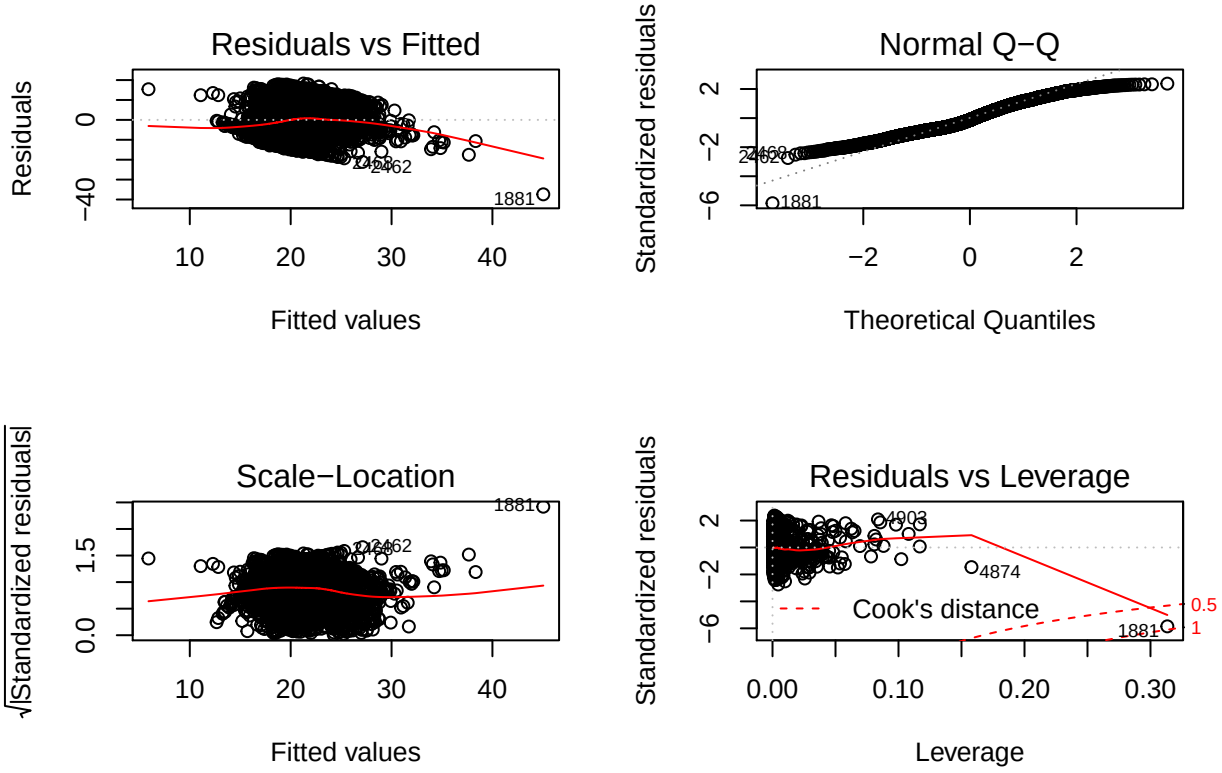
Existen multitud de modelos que podemos usar para problemas de regresión pero nos vamos a centrar en un modelo lineal concreto, **Regresión Lineal** y tres no son lineales, SVM (Support Vector Machine), Boosting y RF (RandomForest). Que explicaremos a continuación.

- **Regresión Lineal**: Es el modelo más básico de aprendizaje y del que más hemos estudiado en teoría. El modelo asume que, dado un conjunto de datos $ {y_i, x_{i1}, ..., x_{ip}}^n_{i=1} $ de n unidades estadísticas, un modelo lineal asume que la relación entre la variable dependiente y y el $p - vector$ de regresores x es lineal. Esta relación se modela a través de un término de perturbación o variable de

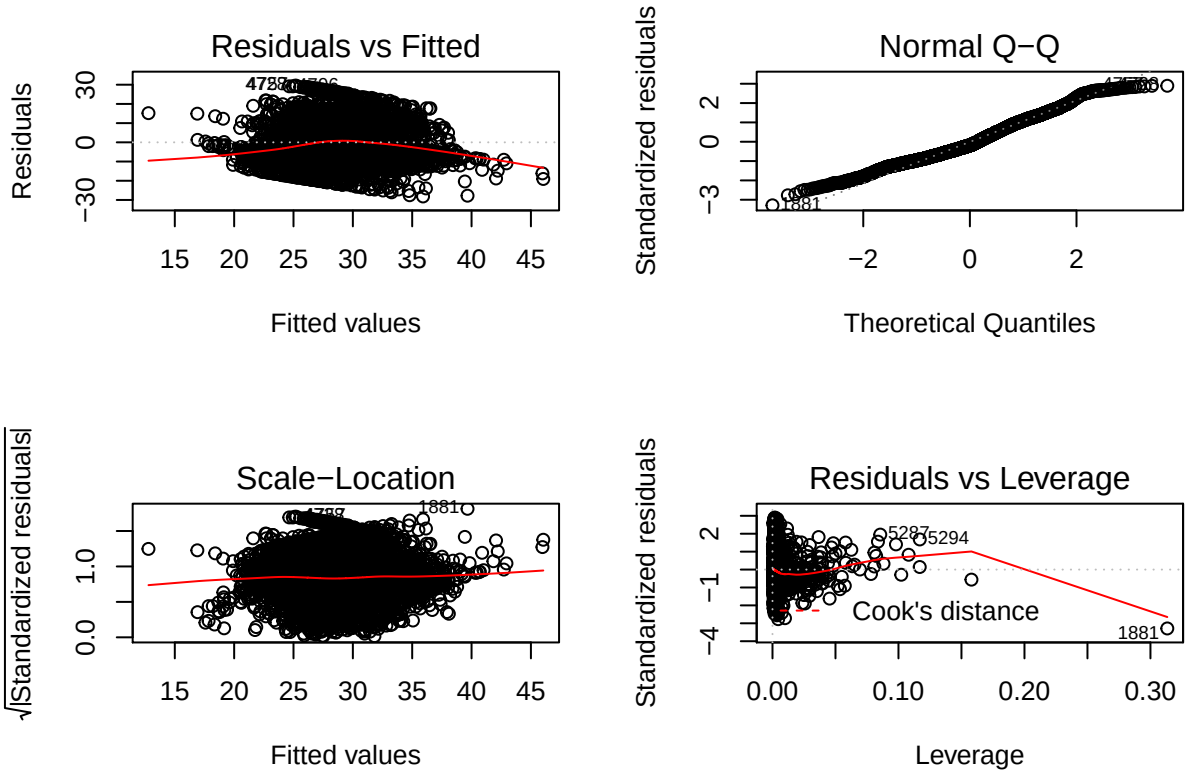
error e, una variable aleatoria no observada que agrega "ruido" a la relación lineal entre la variable dependiente y los regresores.

```
# Aprendemos del train
modelo.lm1 = lm(motor_UPDRS ~ . - total_UPDRS, data = train)
modelo.lm2 = lm(total_UPDRS ~ . - motor_UPDRS, data = train)

# Mostramos la interpretación que ha hecho el modelo
par (mfrow = c (2,2))
plot(modelo.lm1)
```




```
plot(modelo.lm2)
```




```
Sys.sleep(3)
```

- **SVM** (Support Vector Machine): Este modelo busca un hiperplano que separe de forma óptima a los puntos de una clase de la de otra, que eventualmente han podido ser previamente proyectados a un espacio de dimensionalidad superior. En el concepto de "separación óptima" es donde reside la característica fundamental de SVM: este tipo de algoritmos buscan el hiperplano que tenga la máxima distancia (margen) con los puntos que estén más cerca de él mismo. La manera más simple de realizar la separación es mediante una línea recta, un plano recto o un hiperplano N-dimensional. Desafortunadamente los universos a estudiar no se suelen presentar en casos idílicos de dos dimensiones he aquí donde tenemos que tener en cuenta el "kernel" que vamos a usar.

En nuestro caso vamos a usar un nucleo gaussiano que es el nucleo por defecto que usa la librería caret para svm. El parámetro libre está para los dos modelos es por defecto.

```
# Aprendemos cual es el mejor valor libre de dos cifras.
obj1 = best.tune(svm, motor_UPDRS ~ . - total_UPDRS, data = train)
obj2 = best.tune(svm, total_UPDRS ~ . - motor_UPDRS, data = train)
print(obj1)
```

```
##
## Call:
## best.tune(svm, motor_UPDRS ~ . - total_UPDRS, data = train)
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.0625
##     epsilon:  0.1
##
```

```
##
## Number of Support Vectors:  4235
```

```
print(obj2)
```

```
##
## Call:
## best.tune(svm, total_UPDRS ~ . - motor_UPDRS, data = train)
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.0625
##     epsilon:  0.1
##
##
## Number of Support Vectors:  4213
```

El mejor gamma para SVM como podemos ver es: por lo que vamos a aprender el modelo con este parámetro. Se puede buscar el parámetro de la siguiente forma `best.tune(svm, total_UPDRS ~ . - motor_UPDRS, data = train, ranges = list(epsilon = seq(0,1,0.01), cost = 2^(2:9)))` pero la ejecución es demasiado lenta para ejecutar la documentación.

```
# Aprendemos del train usando ese gamma
modelo.svm1 = svm(motor_UPDRS ~ . - total_UPDRS,
                  data = train, gamma = 0.06, epsilon = 0.1)
modelo.svm2 = svm(total_UPDRS ~ . - motor_UPDRS,
                  data = train, gamma = 0.06, epsilon = 0.1)
```

- **Boosting**: El boosting consiste en combinar los resultados de varios clasificadores débiles para obtener un clasificador robusto. Cuando se añaden estos clasificadores débiles, se lo hace de modo que estos tengan diferente peso en función de la exactitud de sus predicciones. Luego de que se añade un clasificador débil, los datos cambian su estructura de pesos: los casos que son mal clasificados ganan peso y los que son clasificados correctamente pierden peso. Así, los clasificadores débiles se centran de mayor manera en los casos que fueron mal clasificados por los clasificadores débiles.

```
modelo.gbm1 = gbm(motor_UPDRS ~ . - total_UPDRS, data = train)
```

```
## Distribution not specified, assuming gaussian ...
```

```
modelo.gbm2 = gbm(total_UPDRS ~ . - motor_UPDRS, data = train)
```

```
## Distribution not specified, assuming gaussian ...
```

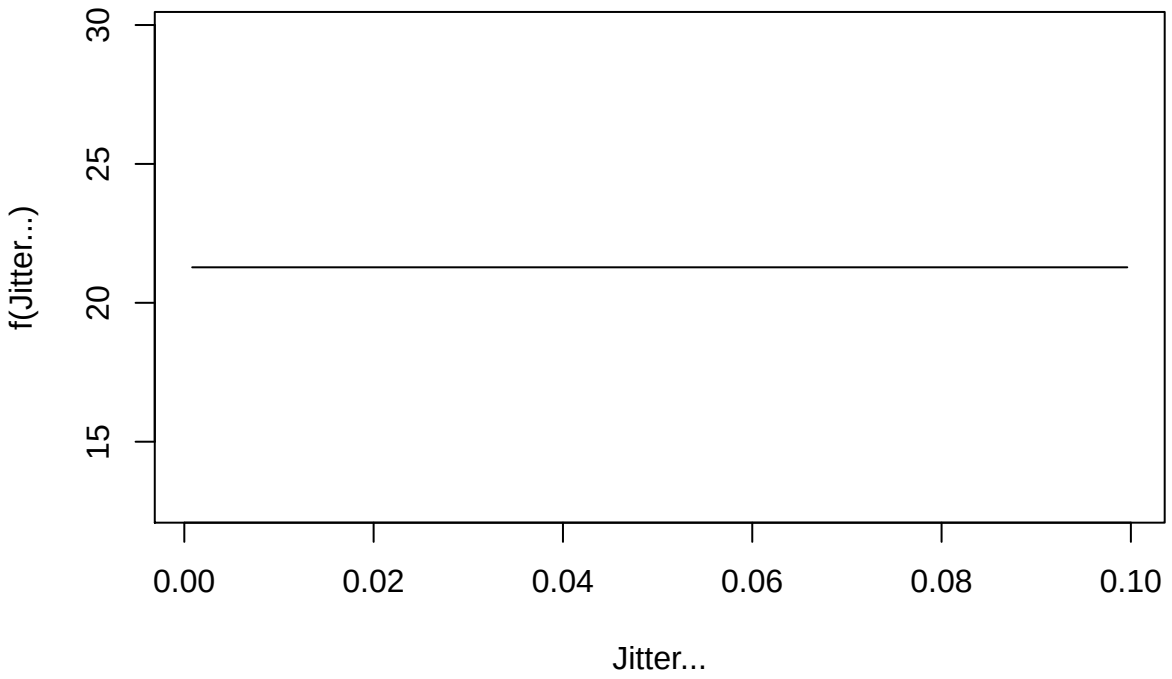
```
print(modelo.gbm1)
```

```
## gbm(formula = motor_UPDRS ~ . - total_UPDRS, data = train)
## A gradient boosted model with gaussian loss function.
## 100 iterations were performed.
## There were 16 predictors of which 4 had non-zero influence.
```

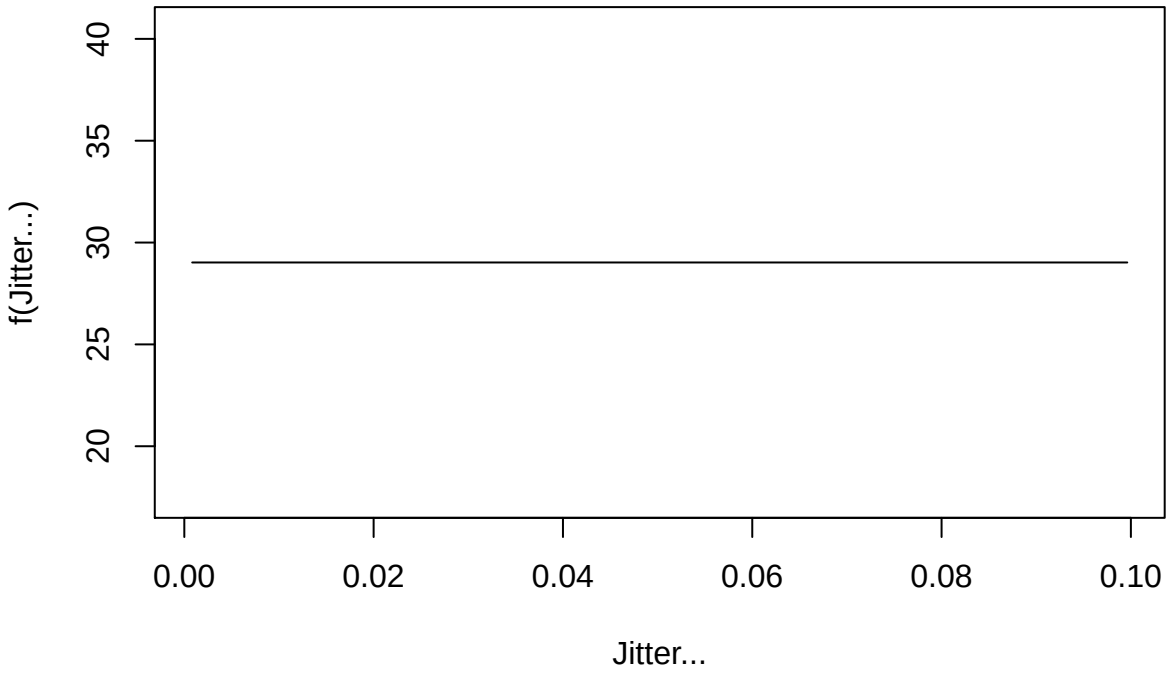
```
print(modelo.gbm2)
```

```
## gbm(formula = total_UPDRS ~ . - motor_UPDRS, data = train)
## A gradient boosted model with gaussian loss function.
## 100 iterations were performed.
## There were 16 predictors of which 6 had non-zero influence.
```

```
plot(modelo.gbm1)
```




```
plot(modelo.gbm2)
```




```
Sys.sleep(3)
```

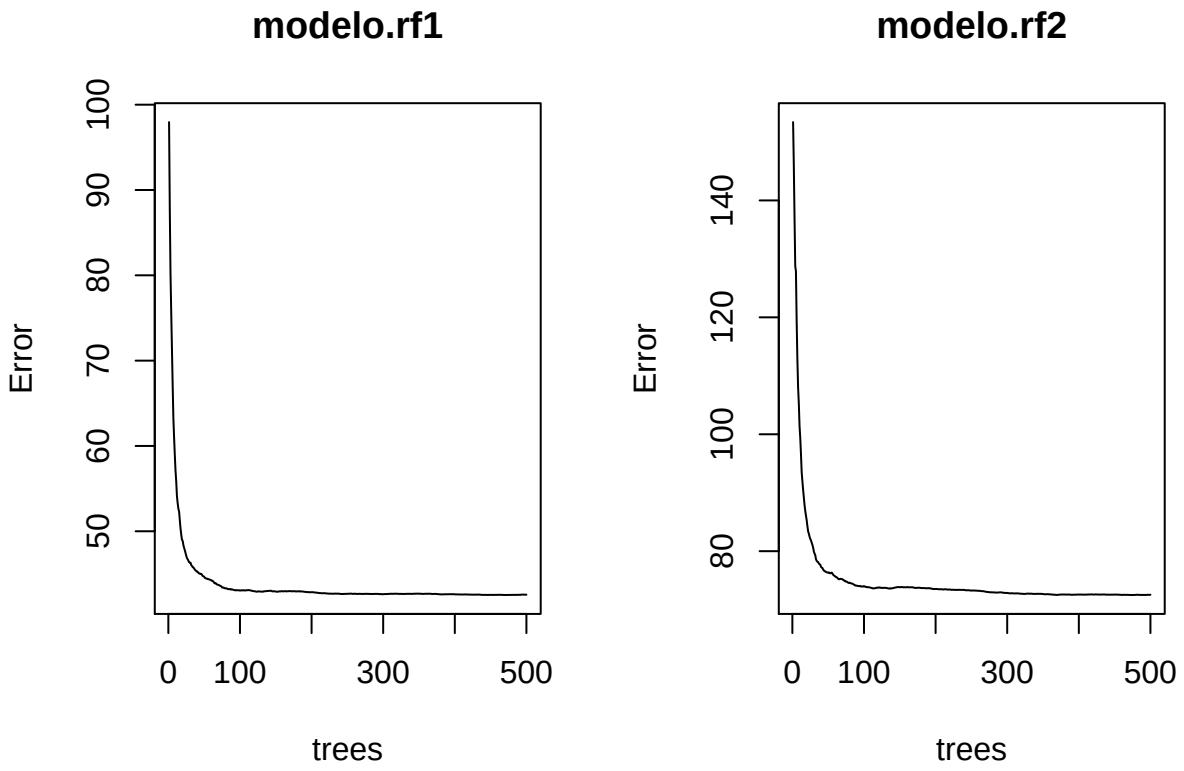
- **RT** (Random Forest): Es una combinación de árboles predictores tal que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos. Es una modificación sustancial de bagging que construye una larga colección de árboles no correlacionados y luego los promedia.

Como es un problema de regresión por defecto Random Forest del paquete que he usado toma como m=p/3

donde p es el número de características. Además usa unos 500 arboles por defecto. Una vez pintemos el modelo nos daremos cuenta de los parámetro optimos que en nuestro caso es 100 arboles como vemos en las fraficas, cuando ajustemos el modelo buscaremos el mejor m, mtry para RandomForest.

```
# Aprendemos el modelo
modelo.rf1 = randomForest(motor_UPDRS ~ . - total_UPDRS, data = train)
modelo.rf2 = randomForest(total_UPDRS ~ . - motor_UPDRS, data = train)

par (mfrow = c (1,2))
plot(modelo.rf1)
plot(modelo.rf2)
```




```
Sys.sleep(3)
```

1.6 Evaluación de los modelos.

Una vez creados los modelos y haberlos entrenado tenemos que ver los resultados, recordemos que aún no hemos normalizado los datos por lo que los resultados no son definitivos. Por cada método tenemos dos modelos, uno que pronostica los valores $motor_UPDRS$ y otro para $total_UPDRS$. He creado una pequeña función para este cometido. La función evalua el E_{in} y el E_{out}.

```
eval_model <- function(model,type=1, Boosting = F) {

  ein <- 0
  eout <- 0
  # Para boosting neceesitamos el número de arboles que van a ser
  # usados para pronosticar, que los ajustaremos a 100.
  if(Boosting == FALSE){
    pred_train <- predict(model,newdata = train)
    pred_test <- predict(model,newdata = test)
```

```
    # Tipo indica si estamos hablando de motor_UDPRS o total_UDPRS
    if (type == 1) {
      ein <- sqrt(mean((pred_train-train$motor_UPDRS)^2))
      eout <- sqrt(mean((pred_test-test$motor_UPDRS)^2))
    }else{
      ein<- sqrt(mean((pred_train-train$total_UPDRS)^2))
      eout<- sqrt(mean((pred_test-test$total_UPDRS)^2))
    }
  }else{
    pred_train <- predict(model,newdata = train, n.trees = 100)
    pred_test <- predict(model,newdata = test, n.trees = 100)

    if (type == 1) {
      ein<- sqrt(mean((pred_train-train$motor_UPDRS)^2))
      eout<- sqrt(mean((pred_test-test$motor_UPDRS)^2))
    }else{
      ein<- sqrt(mean((pred_train-train$total_UPDRS)^2))
      eout<- sqrt(mean((pred_test-test$total_UPDRS)^2))
    }
  }

  return (c(ein,eout))
}
```

La medida de error que he usado es el error cuadratico medio que se define de la sigiente forma: Si S es un vector de n predicciones e Y el vector de valores verdaderos entonces la estimación de ECM sería:

$ECM = \frac{1}{n}\sum_{i=1}^{n}(S_i - Y_i)^2$

Vamos ahora a evaluar los modelos obtenidos y a ver sus resultados.

```
eval.lm1 = eval_model(modelo.lm1,1)
eval.lm2 = eval_model(modelo.lm2,2)

eval.svm1 = eval_model(modelo.svm1,1)
eval.svm2 = eval_model(modelo.svm2,2)

eval.rf1 = eval_model(modelo.rf1,1)
eval.rf2 = eval_model(modelo.rf2,2)

eval.gbm1 = eval_model(modelo.gbm1,1,T)
eval.gbm2 = eval_model(modelo.gbm2,2,T)

cat("LM    \n     motor_UPDRS - Ein: ", eval.lm1[1],
    "      \n     total_UPDRS - Ein: ", eval.lm2[1],

    "\nSVM \n     motor_UPDRS - Ein: ", eval.svm1[1],
    "      \n     total_UPDRS - Ein: ", eval.svm2[1],

    "\nBST \n     motor_UPDRS - Ein: ", eval.gbm1[1],
    "      \n     total_UPDRS - Ein: ", eval.gbm2[1],

    "\nRF  \n     motor_UPDRS - Ein: ", eval.rf1[1],
    "      \n     total_UPDRS - Ein: ", eval.rf2[1])
```

```
## LM
```

```
##      motor_UPDRS - Ein:  7.696771
##      total_UPDRS - Ein:  10.17872
## SVM
##      motor_UPDRS - Ein:  6.779414
##      total_UPDRS - Ein:  9.059235
## BST
##      motor_UPDRS - Ein:  8.144897
##      total_UPDRS - Ein:  10.71589
## RF
##      motor_UPDRS - Ein:  2.716625
##      total_UPDRS - Ein:  3.539144
```

```
Sys.sleep(3)
```

Como podemos ver los resultados son bastante buenos, en general el Ein es muy bajo pero claro, estamos viendo el valor del error sobre los datos que hemos aprendido por lo que deberiamos esperarnos un error muy bajo y ha sido así. Sin embargo debemos de ver el Ein ya que nos indicará si nuestros modelos predicen de forma correcta o si estamos cometiendo un sobreaprendidaje.

```
cat(
    "\nLM  \n     motor_UPDRS - Eout: ", eval.lm1[2],
    "       \n     total_UPDRS - Eout: ", eval.lm2[2],

    "\nSVM \n     motor_UPDRS - Eout: ", eval.svm1[2],
    "       \n     total_UPDRS - Eout: ", eval.svm2[2],

    "\nBST \n     motor_UPDRS - Eout: ", eval.gbm1[2],
    "       \n     total_UPDRS - Eout: ", eval.gbm2[2],

    "\nRF  \n     motor_UPDRS - Eout: ", eval.rf1[2],
    "       \n     total_UPDRS - Eout: ", eval.rf2[2])
```

```
##
## LM
##      motor_UPDRS - Eout:  7.680498
##      total_UPDRS - Eout:  10.04321
## SVM
##      motor_UPDRS - Eout:  6.827234
##      total_UPDRS - Eout:  9.069777
## BST
##      motor_UPDRS - Eout:  7.953451
##      total_UPDRS - Eout:  10.45873
## RF
##      motor_UPDRS - Eout:  6.449679
##      total_UPDRS - Eout:  8.403252
```

```
Sys.sleep(3)
```

Podemos observar que no hemos cometido ningún tipo de sobreaprendizaje, es más, el error medio es bajo, ningún modelo supera el 10% y claramente Random Forest es el mejor modelo con los parámetros prestablecidos.

Vamos a aplicar normalización a los datos para ver como cambian las predicciones realizadas por los modelos. Tras esto vamos a ajustar el mejor modelo hasta el momento.

1.7. Preprocesando los datos.

Para el preprocesado vamos a usar la librería caret, como tenemos 18 caracteristicas bajo mi punto de vista no hace falta aplicar un métodos de preProcesado como PCA (Analisis Principales Componentes) aunque como prueba lo hemos voy a mostrar veremos que se reducen el número de características enormente.

El preProcesado que vamos a hacer es una estandarización, esto es aplicar dos transformaciones, scala y centro que provocan que los atributos tengan un valor medio 0 y una desviación standar de 1:

```
# Calculamos la transformación
trans = preProcess(train, method = c("center","scale"))
trans2 = preProcess(train, method = c("center","scale", "pca"))

# Obtenemos el nuevo conunto de datos.
train = predict(trans, train)
test = predict(trans, test)

train.pca = predict(trans2, train)
test.pca = predict(trans2, test)

cat("Dim Train con PCA: ", dim(train.pca))
```

```
## Dim Train con PCA:  4700 6
```

```
Sys.sleep(3)
```

Como podemos observar la dimensión si hubiesemos aplicado PCA se reduce muchísimo peor vamos a usar solo los datos con la estandarización.

Vamos ahora a calcular los nuevos modelos con los nuevos datos transformados:

```
# Aprendemos de nuevo los modelos
modelo2.lm1 = lm(motor_UPDRS ~ . - total_UPDRS, data = train)
modelo2.lm2 = lm(total_UPDRS ~ . - motor_UPDRS, data = train)
modelo2.svm1 = svm(motor_UPDRS ~ . - total_UPDRS,
                   data = train, gamma = 0.06, epsilon = 0.1)
modelo2.svm2 = svm(total_UPDRS ~ . - motor_UPDRS,
                   data = train, gamma = 0.06, epsilon = 0.1)
modelo2.rf1 = randomForest(motor_UPDRS ~ . - total_UPDRS, data = train)
modelo2.rf2 = randomForest(total_UPDRS ~ . - motor_UPDRS, data = train)
modelo2.gbm1 = gbm(motor_UPDRS ~ . - total_UPDRS, data = train)
```

```
## Distribution not specified, assuming gaussian ...
```

```
modelo2.gbm2 = gbm(total_UPDRS ~ . - motor_UPDRS, data = train)
```

```
## Distribution not specified, assuming gaussian ...
```

```
# Evaluamos los modelos
eval2.lm1 = eval_model(modelo2.lm1,1)
eval2.lm2 = eval_model(modelo2.lm2,2)
eval2.svm1 = eval_model(modelo2.svm1,1)
eval2.svm2 = eval_model(modelo2.svm2,2)
eval2.rf1 = eval_model(modelo2.rf1,1)
eval2.rf2 = eval_model(modelo2.rf2,2)
eval2.gbm1 = eval_model(modelo2.gbm1,1,T)
eval2.gbm2 = eval_model(modelo2.gbm2,2,T)
```

Mostramos los nuevos resultaados:

```r
# Mostramos el Ein
# -----------------------------------------------------
cat("LM    \n     motor_UPDRS - Ein: ", eval2.lm1[1],
    "      \n     total_UPDRS - Ein: ", eval2.lm2[1],

    "\nSVM \n     motor_UPDRS - Ein: ", eval2.svm1[1],
    "      \n     total_UPDRS - Ein: ", eval2.svm2[1],

    "\nBST \n     motor_UPDRS - Ein: ", eval2.gbm1[1],
    "      \n     total_UPDRS - Ein: ", eval2.gbm2[1],

    "\nRF  \n     motor_UPDRS - Ein: ", eval2.rf1[1],
    "      \n     total_UPDRS - Ein: ", eval2.rf2[1])
```

```
## LM
##      motor_UPDRS - Ein:  0.9423218
##      total_UPDRS - Ein:  0.9466628
## SVM
##      motor_UPDRS - Ein:  0.8300091
##      total_UPDRS - Ein:  0.8425535
## BST
##      motor_UPDRS - Ein:  0.9972129
##      total_UPDRS - Ein:  0.9966263
## RF
##      motor_UPDRS - Ein:  0.3320566
##      total_UPDRS - Ein:  0.3292721
```

```r
Sys.sleep(3)

# Mostramos el Eout
# -----------------------------------------------------
cat(
    "\nLM  \n     motor_UPDRS - Eout: ", eval2.lm1[2],
    "      \n     total_UPDRS - Eout: ", eval2.lm2[2],

    "\nSVM \n     motor_UPDRS - Eout: ", eval2.svm1[2],
    "      \n     total_UPDRS - Eout: ", eval2.svm2[2],

    "\nBST \n     motor_UPDRS - Eout: ", eval2.gbm1[2],
    "      \n     total_UPDRS - Eout: ", eval2.gbm2[2],

    "\nRF  \n     motor_UPDRS - Eout: ", eval2.rf1[2],
    "      \n     total_UPDRS - Eout: ", eval2.rf2[2])
```

```
##
## LM
##      motor_UPDRS - Eout:  0.9403294
##      total_UPDRS - Eout:  0.9340598
## SVM
##      motor_UPDRS - Eout:  0.8358636
##      total_UPDRS - Eout:  0.8435314
## BST
##      motor_UPDRS - Eout:  0.9737763
##      total_UPDRS - Eout:  0.9727055
## RF
```

```
##      motor_UPDRS - Eout:  0.7913215
##      total_UPDRS - Eout:  0.7807582
```

```
Sys.sleep(3)
```

Gracias a la standarización hemos obtenido unos nuevos resultados muy buenos. Todos los modelos han bajado su error medio mucho pero Random Forest sigue siendo el mejor modelo, el que mejor hace predicciones sobre este problema por lo que pasará a ser el modelo que vamos a ajustar.

1.8. Ajuste del modelo final.

Para el ajuste del modelo final de random fores voy a usar diferentes valores de para m. La variable m identifica al número de caracteristicas aleatorias que vamos a cambiar por arbol de decisión. En clase hemos visto tres valores que podemos usar que son los siguientes: - m=p - m=p/2 - m=sqrt(p)

Como he dicho anteriormente de forma predeterminada Random Forest usa 500 arboles y un m=p/3.

```
# Aprendemos los modelos
rf.ajuste1 = randomForest(motor_UPDRS ~ . - total_UPDRS, data = train,
                          ntree=200, importance = T, mtry=dim(train)[2]-1)
```

```
## Warning in randomForest.default(m, y, ...): invalid mtry: reset to within
## valid range
```

```
rf.ajuste2 = randomForest(total_UPDRS ~ . - motor_UPDRS, data = train,
                          ntree=200, importance = T, mtry=dim(train)[2]-1)
```

```
## Warning in randomForest.default(m, y, ...): invalid mtry: reset to within
## valid range
```

```
rf.ajuste3 = randomForest(motor_UPDRS ~ . - total_UPDRS, data = train,
                          ntree=200, importance = T, mtry=dim(train)[2] / 2)
rf.ajuste4 = randomForest(total_UPDRS ~ . - motor_UPDRS, data = train,
                          ntree=200, importance = T, mtry=dim(train)[2] / 2)

rf.ajuste5 = randomForest(motor_UPDRS ~ . - total_UPDRS, data = train,
                          ntree=200, importance = T, mtry=sqrt(dim(train)[2]))
rf.ajuste6 = randomForest(total_UPDRS ~ . - motor_UPDRS, data = train,
                          ntree=200, importance = T, mtry=sqrt(dim(train)[2]))

# Los evaluamos
rf.eval1 = eval_model(rf.ajuste1,1)
rf.eval2 = eval_model(rf.ajuste2,2)
rf.eval3 = eval_model(rf.ajuste3,1)
rf.eval4 = eval_model(rf.ajuste4,2)
rf.eval5 = eval_model(rf.ajuste5,1)
rf.eval6 = eval_model(rf.ajuste6,2)
```

```
# Mostramos los resultados:
cat("\nRF m=p
          \n     motor_UPDRS - Ein: ", rf.eval1[1],
         "\n     total_UPDRS - Ein: ", rf.eval2[1],
         "\n     motor_UPDRS - Eout: ", rf.eval1[2],
         "\n     total_UPDRS - Eout: ", rf.eval2[2])
```

```
##
## RF m=p
```

```
##
##      motor_UPDRS - Ein:  0.3200942
##      total_UPDRS - Ein:  0.3172451
##      motor_UPDRS - Eout:  0.7785105
##      total_UPDRS - Eout:  0.7736232
```

```
cat("\nRF m=p/2
          \n     motor_UPDRS - Ein: ", rf.eval3[1],
         "\n     total_UPDRS - Ein: ", rf.eval4[1],
         "\n     motor_UPDRS - Eout: ", rf.eval3[2],
         "\n     total_UPDRS - Eout: ", rf.eval4[2])
```

```
##
## RF m=p/2
##
##      motor_UPDRS - Ein:  0.3246054
##      total_UPDRS - Ein:  0.3211603
##      motor_UPDRS - Eout:  0.7814161
##      total_UPDRS - Eout:  0.7768244
```

```
cat("\nRF m=sqrt(p)
          \n     motor_UPDRS - Ein: ", rf.eval5[1],
         "\n     total_UPDRS - Ein: ", rf.eval6[1],
         "\n     motor_UPDRS - Eout: ", rf.eval5[2],
         "\n     total_UPDRS - Eout: ", rf.eval6[2])
```

```
##
## RF m=sqrt(p)
##
##      motor_UPDRS - Ein:  0.3395401
##      total_UPDRS - Ein:  0.3377289
##      motor_UPDRS - Eout:  0.7937592
##      total_UPDRS - Eout:  0.7873149
```

```
Sys.sleep(3)
```

El ajuste del modelo ha sido efectivo y hemos mejorado en 0,01% el error, algo practicamente insignificannte pero porque nuestro modelo es muy bueno. Además podemos destacar que efectivamente no estamos cometiendo sobreajuste cosa de la que nos hubiesemos dado cuenta comparando los direntes Ein y Eout.