

# **Bagging, Random Forest, Gradient Boosting**

## Desventajas de los árboles

- Poca fiabilidad y mala generalización: cada hoja es un parámetro y esto provoca modelos sobreajustados e inestables para la predicción. Añadir una variable nueva o un nuevo conjunto de observaciones puede alterar mucho el árbol.
- Complejidad en la construcción del árbol y casuística: dos plataformas (programas) diferentes dan dos árboles diferentes
- Poca eficacia predictiva, sobre todo en regresión: toscos en los valores de predicción.

Las desventajas de los árboles no han podido ser solucionadas mejorando las funciones de error o algoritmos de construcción, pero **sí combinando** el resultado de muchos árboles.

## Primer intento importante: **Bagging (Bootstrap Averaging) (Breiman, 1996)**

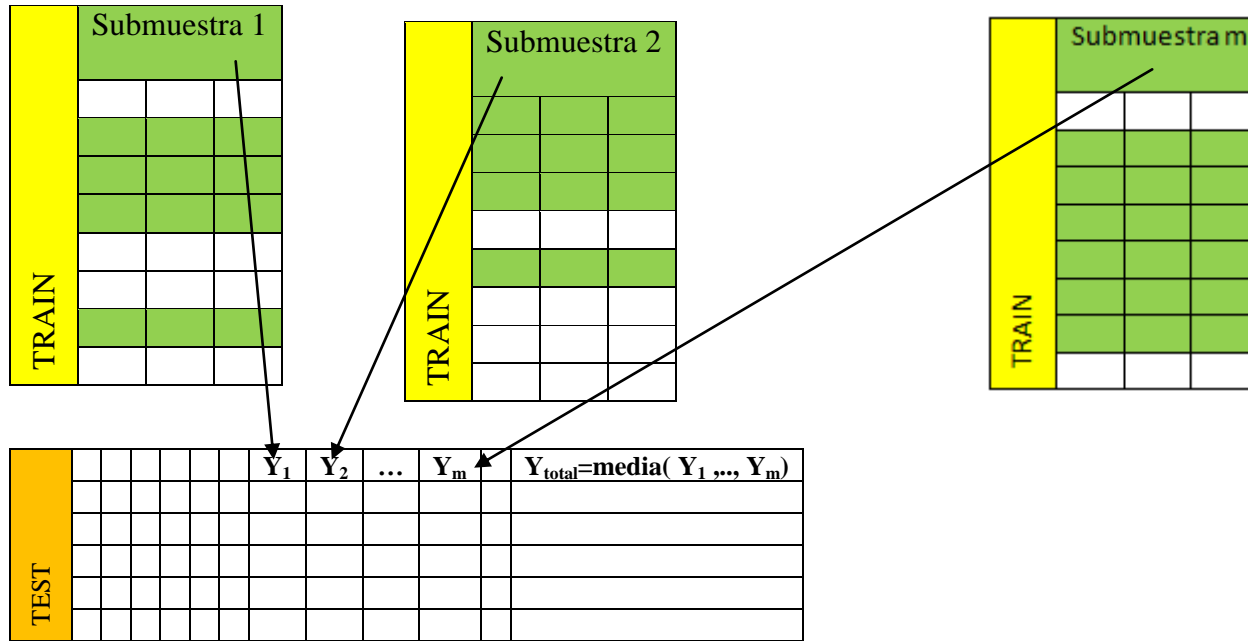
### **Bagging**

Dados los datos de tamaño  $N$ ,

1) Repetir  $m$  veces i) y ii):

- (i) Seleccionar  $N$  observaciones con reemplazamiento de los datos originales
- (ii) Aplicar un árbol y obtener predicciones para todas las observaciones originales  $N$

2) Promediar las  $m$  predicciones obtenidas en el apartado 1)



Con cada submuestra se genera un modelo con el que se predicen los datos test. La predicción final será la media de las m diferentes predicciones.

Al utilizarse diferentes submuestras, se reduce la dependencia de la estructura de los datos completos para construir el modelo, y como consecuencia se reduce la varianza del modelo (y a menudo también el sesgo).

Aunque se puede utilizar la técnica con algoritmos diferentes de los árboles, su influencia es mucho más grande con árboles, al ser estos muy dependientes de los datos utilizados.

## Bagging

Dados los datos de tamaño  $N$ ,

1) Repetir  $m$  veces i) y ii):

(i) Seleccionar  $N$  observaciones con reemplazamiento de los datos originales

(ii) Aplicar un árbol y obtener predicciones para todas las observaciones originales  $N$

2) Promediar las  $m$  predicciones obtenidas en el apartado 1)

### Notas:

- El apartado (i) admite todo tipo de variaciones: tomar  $n < N$  **con** o **sin** reemplazamiento, estratificación, etc.
- En cuanto a (ii), la complejidad del árbol a utilizar es un tema a debate. Inicialmente se propuso árboles débiles (pocas hojas finales) y muchas iteraciones  $m$ . Pero en algunas versiones se utilizan árboles desarrollados hasta el final sin prefijar el número de hojas o profundidad.
- Si se trata de un problema de clasificación, dos estrategias pueden ser utilizadas:
  - a) promediar las probabilidades estimadas y obtener una clasificación a partir de un punto de corte.
  - b) clasificar en cada iteración y asignar a cada observación la clasificación mayoritaria entre todas las iteraciones (majority voting).

La idea del submuestreo es controlar el sobreajuste implícito en la selección rígida de variables y estimación fija de parámetros que caracteriza a los métodos directos.

En general, bagging funciona bien:

- Cuando los modelos no están claros (muchas variables con relación débil pero estable con la variable dependiente, multiplicidad de modelos-opciones)
- Cuando existen relaciones no lineales (regresión) o separaciones no lineales (clasificación)
- Cuando existen interacciones ocultas, muchas variables categóricas, etc.

## Principales parámetros a controlar en bagging

- El tamaño de las muestras **n** y si se va a utilizar bootstrap (con reemplazo) o sin reemplazamiento (si el número de observaciones es pequeño, mejor utilizar con reemplazamiento. Si es grande, es indiferente pues el resultado con o sin reemplazamiento es muy similar).
- El número de iteraciones **m** a promediar (no importante, no se produce sobreajuste por demasiadas iteraciones y en general a partir de un cierto momento temprano no se mejora nada)
- Características de los árboles. Son bastante influyentes:
  - El número de hojas final o, en su defecto, la profundidad del árbol
  - El maxbranch (número de divisiones máxima en cada nodo. Por defecto se dejará en 2, árboles binarios)
  - El número de observaciones mínimo en una rama-nodo. Se puede ampliar para evitar sobreajuste (reducir varianza) o reducir para ajustar mejor (reducir sesgo).
  - Otros, como el parámetro de complejidad (que no suele estar incluido en los paquetes)

# Bagging con caret

Para aplicar bagging, se utilizan los paquetes de Random Forest pues como se verá Bagging es un caso particular de Random Forest.

## Parámetros básicos:

- mtry (el único que tunea caret): para bagging se pone el número total de variables independientes del modelo
- nodesize: tamaño máximo de nodos finales (el parámetro que mide la complejidad)
- ntree=el número de iteraciones (árboles)
- sampsize=el tamaño de cada muestra bagging
- replace=TRUE (con reemplazamiento o FALSE sin reemplazamiento)

## Control de la semilla para reproducibilidad en caret

Una semilla general (set.seed)



## ¿Para qué se usan las semillas en bagging?

- a) para sortear que observaciones caen en cada fold de validación cruzada si se usa method="cv"
- b) para sortear las observaciones que se usan para construir un árbol en cada iteración de bagging (sampsize=200)

En este ejemplo en cada ejecución de validación cruzada se deja fuera un fold de tamaño  $(462/4)=115$  y se utilizan  $(3/4)*462=346$  observaciones training para construir el modelo con lo cual sampsize máximo ha de ser 345 aprox).

```
rfgrid<-expand.grid(mtry=c(6))
set.seed(123456)
control<-trainControl(method = "cv",number=4,savePredictions = "all",
classProbs=TRUE,seeds=seeds)
```

```
rf<- train(data=saheartbis,
  factor(chd)~age+tobacco+ldl+adiposity+typea+famhist.Absent,
  method="rf",trControl=control,tuneGrid=rfgrid,
  linout = FALSE,ntree=5000,sampsize=200,nodesize=10,replace=TRUE)
```

```
rf
```

```
Accuracy  Kappa
0.7077399 0.3309909
```

```
# Probamos un árbol solo para ver el error
```

```
arbolgrid <-expand.grid(cp=c(0))
```

```
arbol<- train(factor(chd)~age+tobacco+ldl+adiposity+typea+famhist.Absent,
data=saheartbis,
  method="rpart",trControl=control,tuneGrid=arbolgrid, minbucket=10)
```

```
arbol
```

```
Accuracy  Kappa
0.640461 0.1901329
```

Se observa cómo la precisión del bagging, que combina en este caso  $n_{tree}=5000$  árboles, es mucho mejor que la de un árbol solo (0.70 frente a 0.67).

El concepto del Out Of Bag (OOB): es el error cometido en las observaciones que no caen en la muestra, y por tanto pueden ser tomados como observaciones test y sirven para observar el error cometido sobre test a medida que avanzan las iteraciones.

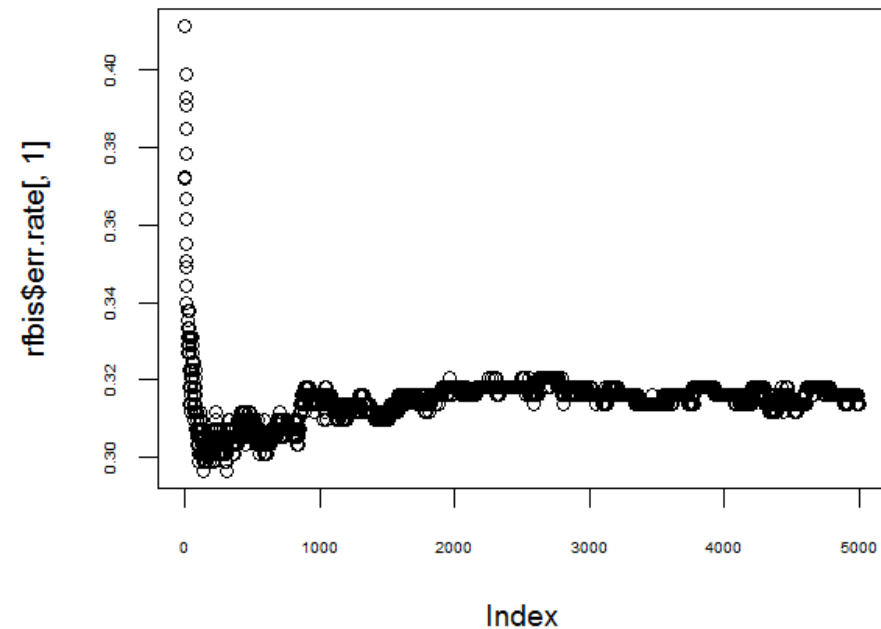
Caret no permite observar esto, pero sí el paquete randomForest

```
# PARA PLOTEAR EL ERROR OOB A MEDIDA QUE AVANZAN LAS ITERACIONES  
# SE USA DIRECTAMENTE EL PAQUETE randomForest
```

```
library(randomForest)
```

```
rfbis<-randomForest( factor(chd)~age+tobacco+ldl+  
adiposity+typea+famhist.Absent,  
data=saheartbis,  
mtry=3,ntree=5000,sampsize=300,nodesize=10,replace=TRUE)
```

```
plot(rfbis$err.rate[,1])
```



## Validación cruzada repetida.

```
# La función cruzadarfbin permite plantear bagging
# (para bagging hay que poner mtry=numero de variables independientes)
```

```
load ("saheartbis.Rda")
source ("cruzadas avnnet y log binaria.R")
source ("cruzada arbolbin.R")
source ("cruzada rf binaria.R")
```

```
medias1<-cruzadalogistica(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""), grupos=4,sinicio=1234, repe=5)
```

```
medias1$modelo="Logística"
```

```
medias2<-cruzadaavnnetbin(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),grupos=4,sinicio=1234, repe=5,
  size=c(5),decay=c(0.1),repeticiones=5,itera=200)
```

```
medias2$modelo="avnnet"
```

```
medias3<-cruzadaarbolbin(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),grupos=4,sinicio=1234, repe=5,
  cp=c(0),minbucket =5)
```

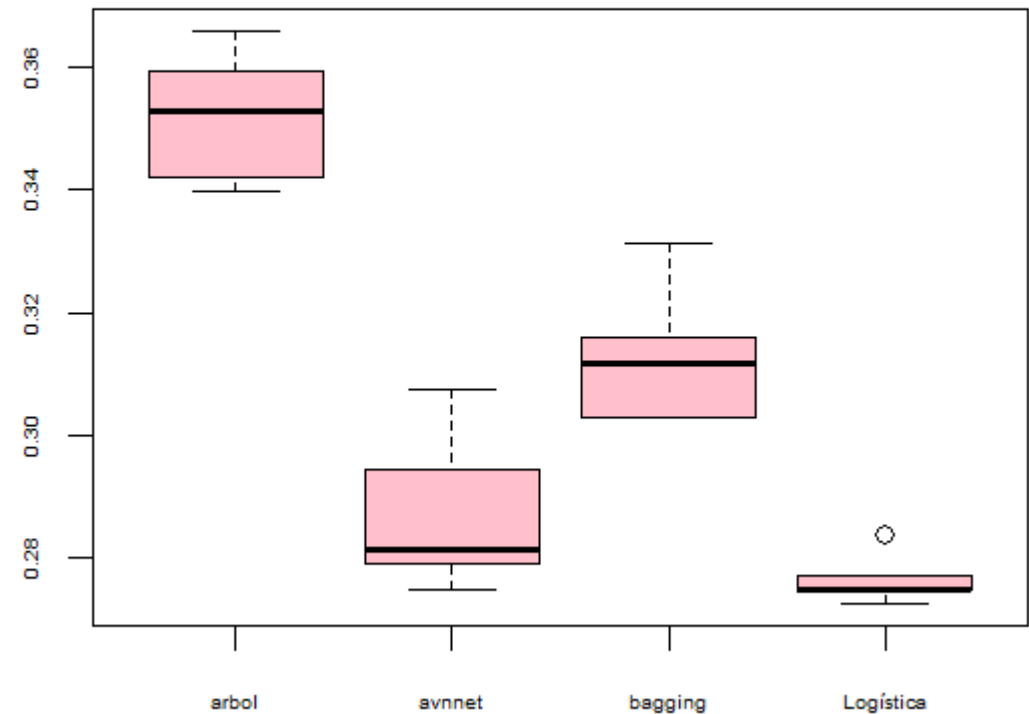
```
medias3$modelo="arbol"
```

```
medias4<-cruzadarfbin(data=saheartbis, vardep="chd",
  listconti=listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,nodesize=10,
  mtry=6,ntree=200,replace=TRUE)
```

```
medias4$modelo="bagging"
```

```
union1<-rbind(medias1,medias2,medias3,medias4)
par(cex.axis=0.5)
boxplot(data=union1,tasa~modelo,main="TASA FALLOS")
boxplot(data=union1,auc~modelo,main="AUC")
```

## TASA FALLOS

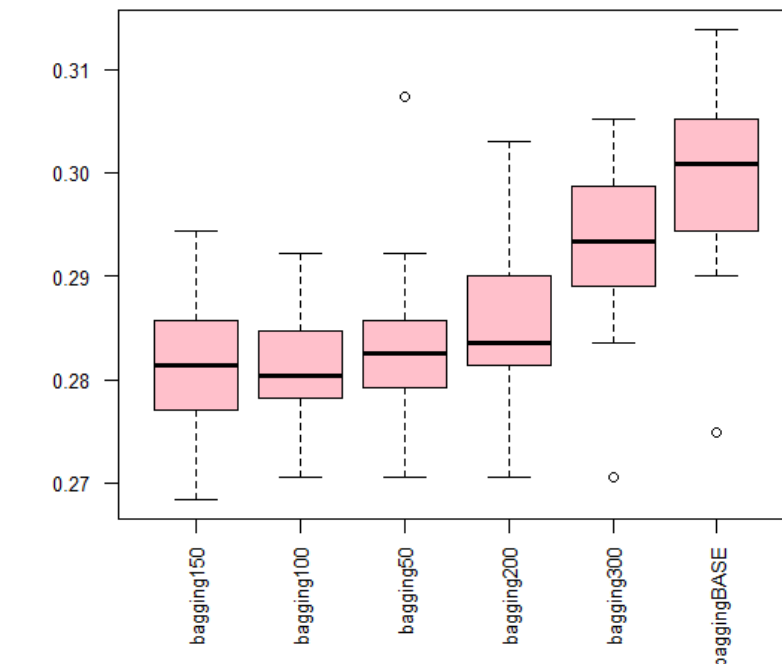


Podemos manipular el tamaño muestral para observar su efecto sobre el modelo bagging, incorporando el parámetro `sampsize` en la función `cruzararfbn`.

`Sampsize` debe ser menor que el número de observaciones training, por lo cual si se usa por ejemplo validación cruzada de 4 grupos `sampsize` debe ser menor que  $0.75 \cdot n$ .

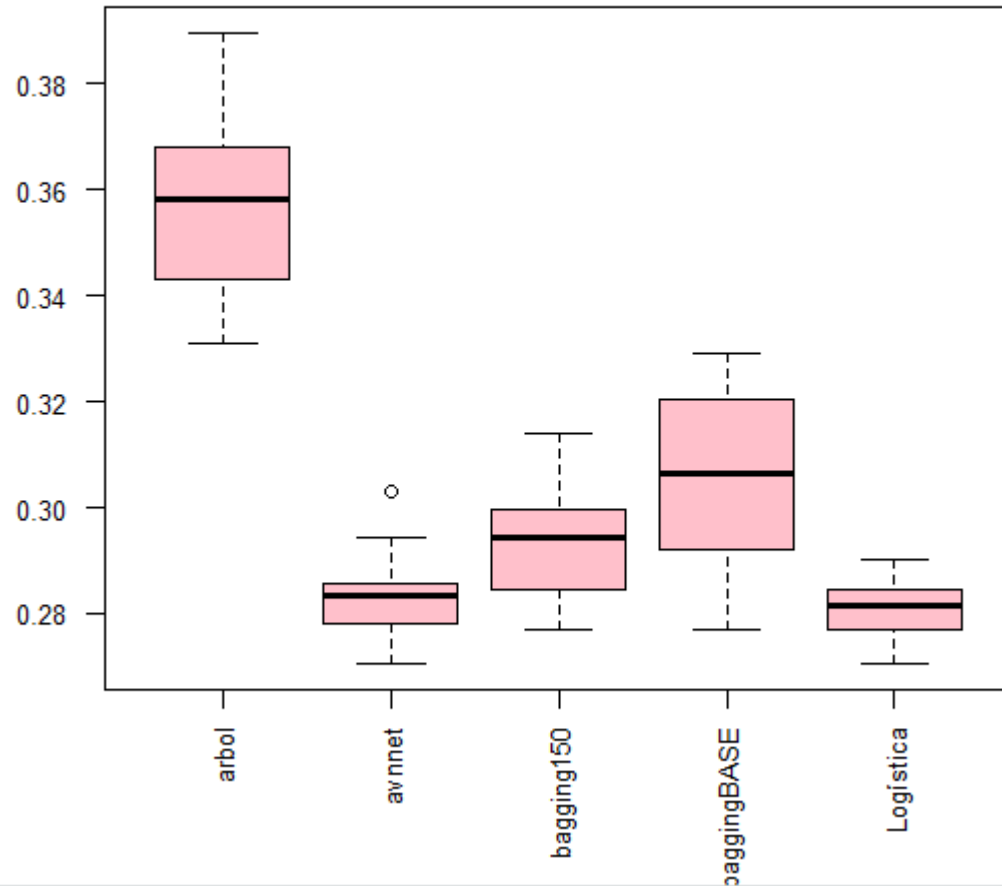
Para controlar bien el tamaño óptimo de la muestra (pues al final aplicaríamos el modelo sobre todos los datos) es mejor aumentar el número de grupos de validación cruzada a 10 por ejemplo. Aumentamos el número de repeticiones a 20 para ver mejor el efecto.

En el ejemplo `saheartbis` hay 462 observaciones, con 10 grupos de CV cada grupo tiene  $0.9 \cdot 462 = 415$  observaciones. Es el máximo de tamaño de muestra que podemos probar. Y es el `sampsize` que prueba por defecto el paquete `caret`.



Parece que 150 es un buen tamaño. Lo aplicamos comparando con los otros modelos con el mismo esquema de 10 grupos.

### TASA FALLOS



## Ejemplo con variable continua dependiente.

```
# EJEMPLO CON VARIABLE CONTINUA
# La función cruzadarf permite plantear bagging PARA VDEP CONTINUA
# (para bagging hay que poner mtry=numero de variables independientes)
# No se puede plotear oob
```

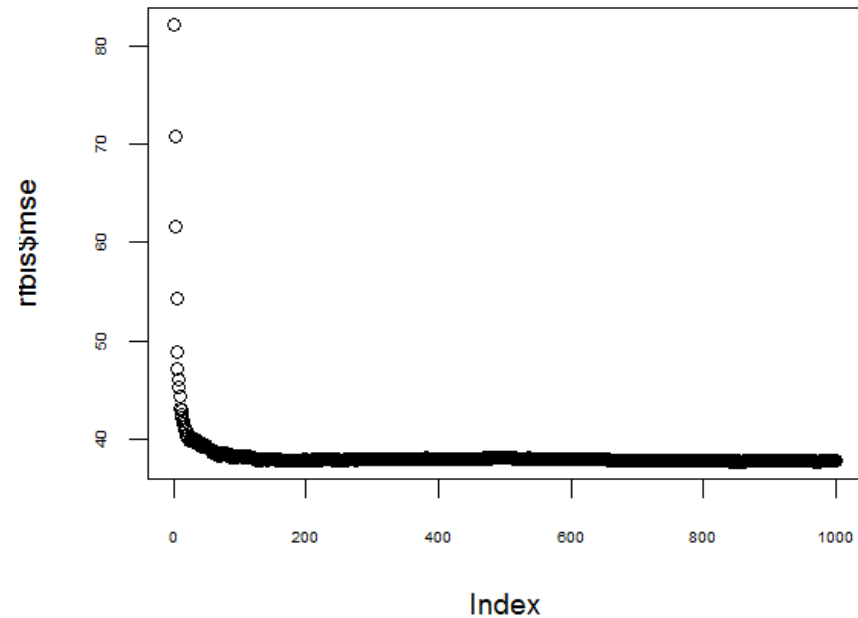
```
load("compressbien.Rda")
```

```
library(randomForest)
```

```
rfbis<-randomForest(cstrength~age+water+cement+blast,
  data=compressbien,
  mtry=4,ntree=1000,samplesize=300,nodesize=10,replace=TRUE)
```

```
# En continuas No estoy seguro de que
# este gráfico esté realizado sobre oob
# pero puede valer para decidir iteraciones
```

```
plot(rfbis$mse)
```



# Validación cruzada repetida

```
data<-compressbien
```

```
medias1<-cruzadaavnnet (data=data,  
vardep="cstrength",listconti=c("age","water","cement","blast"),  
listclass=c(""),grupos=4,sinicio=1234,repe=5,  
size=c(15),decay=c(0.01),repeticiones=5,itera=100)  
medias1$modelo="avnnet"
```

```
medias2<-cruzadalin (data=data,  
vardep="cstrength",listconti=c("age","water","cement","blast"),  
listclass=c(""),grupos=4,sinicio=1234,repe=5)  
medias2$modelo="lineal"
```

```
medias3<-cruzadaarbol (data=data,  
vardep="cstrength",listconti=c("age","water","cement","blast"),  
listclass=c(""),  
grupos=4,sinicio=1234,repe=5,cp=0,minbucket=5)  
medias3$modelo="arbol"
```

```
medias4<-cruzadarf (data=data,  
vardep="cstrength",listconti=c("age","water","cement","blast"),  
listclass=c(""),  
grupos=4,sinicio=1234,repe=5,  
nodesize=20,replace=TRUE,ntree=200,mtry=4)  
medias4$modelo="bagging"
```

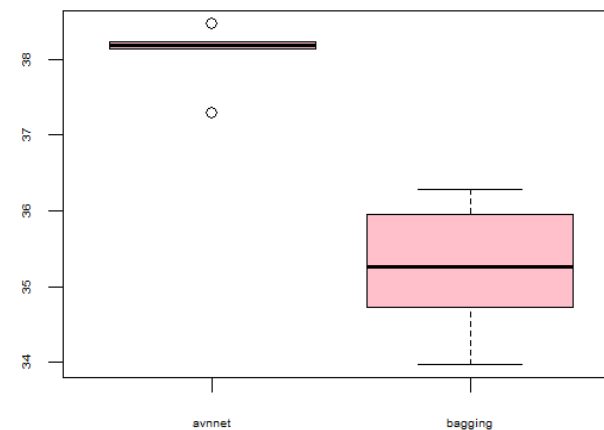
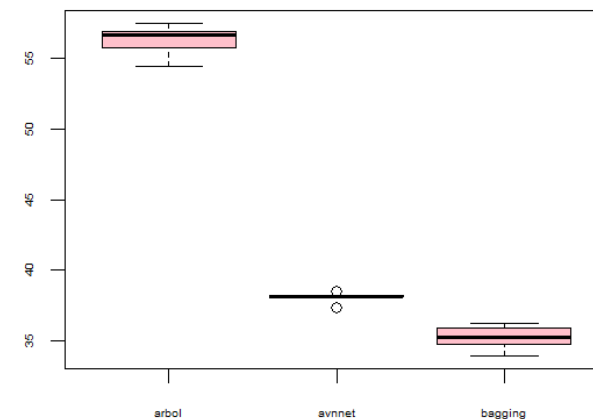
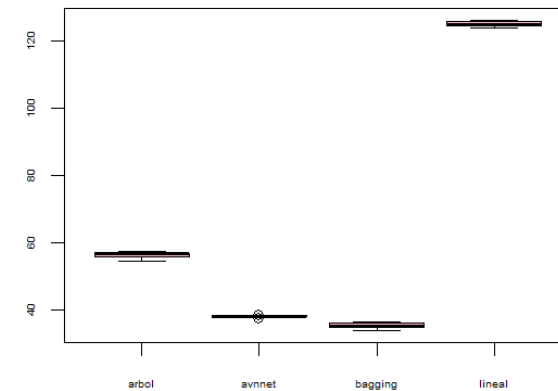
```
union1<-rbind(medias1,medias2,medias3,medias4)
```

```
par(cex.axis=0.5)  
boxplot (data=union1,error~modelo)
```

```
union1<-rbind(medias1,medias2,medias3,medias4)
```

```
par(cex.axis=0.5)  
boxplot (data=union1,error~modelo,col="pink")  
union1<-rbind(medias1,medias3,medias4)  
par(cex.axis=0.5)  
boxplot (data=union1,error~modelo,col="pink")  
union1<-rbind(medias1,medias4)  
par(cex.axis=0.5)  
boxplot (data=union1,error~modelo,col="pink")
```

Ejercicio: tunear con validación cruzada repetida el tamaño de muestra



## Segundo intento importante: **Random Forest (Breiman, 2001)**

Es una modificación del bagging que consiste en incorporar aleatoriedad en las variables utilizadas para segmentar cada nodo del árbol.

### **Random Forest**

Dados los datos de tamaño  $N$ ,

1) Repetir  $m$  veces i), ii), iii):

(i) Seleccionar  $N$  observaciones con reemplazamiento de los datos originales

(ii) Aplicar un árbol de la siguiente manera:

En cada nodo, seleccionar  $p$  variables de las  $k$  originales y de las  $p$  elegidas, escoger la mejor variable para la partición del nodo.

(iii) Obtener predicciones para todas las observaciones originales  $N$

2) Promediar las  $m$  predicciones obtenidas en el apartado 1)



- El algoritmo Random Forest da un paso más en soslayar el problema de selección de variables, evitando decidirse rígidamente por un set de variables y aprovechando a la vez las ventajas del bagging.
- Se trata de **incorporar dos fuentes de variabilidad** (remuestreo de observaciones y de variables) para ganar en **capacidad de generalización**, y reducir el sobreajuste conservando a la vez la facultad de **ajustar bien relaciones particulares** en los datos (interacciones, no linealidad, cortes, problemas de extrapolación, etc.)
- Random Forest evita también el **problema de variables predictoras muy dominantes**. Con solo bagging, en caso de un par de variables muy dominantes los árboles serían todos muy parecidos. Añadiendo aleatoriedad en las variables usadas se obtienen árboles diferentes lo que reduce la varianza del modelo (se verá más adelante por qué).

---

**Algorithm 15.1** *Random Forest for Regression or Classification.*

---

1. For  $b = 1$  to  $B$ :
  - (a) Draw a bootstrap sample  $\mathbf{Z}^*$  of size  $N$  from the training data.
  - (b) Grow a random-forest tree  $T_b$  to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size  $n_{min}$  is reached.
    - i. Select  $m$  variables at random from the  $p$  variables.
    - ii. Pick the best variable/split-point among the  $m$ .
    - iii. Split the node into two daughter nodes.
2. Output the ensemble of trees  $\{T_b\}_1^B$ .

To make a prediction at a new point  $x$ :

*Regression:*  $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$ .

*Classification:* Let  $\hat{C}_b(x)$  be the class prediction of the  $b$ th random-forest tree. Then  $\hat{C}_{\text{rf}}^B(x) = \text{majority vote } \{\hat{C}_b(x)\}_1^B$ .

---

# Principales parámetros a controlar en Random Forest

- El tamaño o % de las muestras **n** y si se va a utilizar bootstrap (con reemplazo) o sin reemplazamiento.
- El número de iteraciones **m** a promediar
- El número de variables **p** a muestrear en cada nodo (si es igual al número inicial de variables **k** el Random Forest es equivalente al Bagging. Dicho de otra manera, **Bagging es un caso particular de Random Forest**)
- Características de los árboles. Son bastante influyentes:
  - El número de hojas final o, en su defecto, la profundidad del árbol
  - El maxbranch (número de divisiones máxima en cada nodo. Por defecto se dejará en 2, árboles binarios)
  - El p-valor para las divisiones en cada nodo. Más alto → árboles menos complejos (más sesgo, menos varianza)
  - El número de observaciones mínimo en una rama-nodo. Se puede ampliar para evitar sobreajuste (reducir varianza) o reducir para ajustar mejor (reducir sesgo).

# Random Forest con caret

## Parámetros básicos:

- mtry (el único que tunea caret): para bagging se pone el número total de variables independientes del modelo
- nodesize: tamaño máximo de nodos finales (el parámetro que mide la complejidad)
- ntree=el número de iteraciones (árboles)
- sampsize=el tamaño de cada muestra bagging
- replace=TRUE (con reemplazamiento o FALSE sin reemplazamiento)

Para tuneado del mtry ponemos muchas variables inicialmente pues el randomforest es bastante robusto a variables malas. Esto no excluye una buena selección de variables, pero por simplificar lo hacemos así en este ejemplo.

# TUNEADO DE MTRY CON CARET

```
library(caret)
```

```
set.seed(12345)
```

```
rfgrid<-expand.grid(mtry=c(3,4,5,6,7,8,9,10,11))
```

```
control<-trainControl(method = "cv", number=10, savePredictions = "all",  
  classProbs=TRUE)
```

```
rf<- train(factor(chd)~., data=saheartbis,  
  method="rf", trControl=control, tuneGrid=rfgrid,  
  linout = FALSE, ntree=3000, nodesize=10, replace=TRUE,  
  importance=TRUE)
```

```
rf
```

mtry	Accuracy	Kappa
3	0.6880204	0.2638950
4	0.6923682	0.2821048
5	0.6880204	0.2748007
6	0.6858002	0.2684658
7	0.6814986	0.2578863
8	0.6814524	0.2582100
9	0.6814524	0.2557267
10	0.6814524	0.2558826
11	0.6814524	0.2553706

```
# IMPORTANCIA DE VARIABLES
```

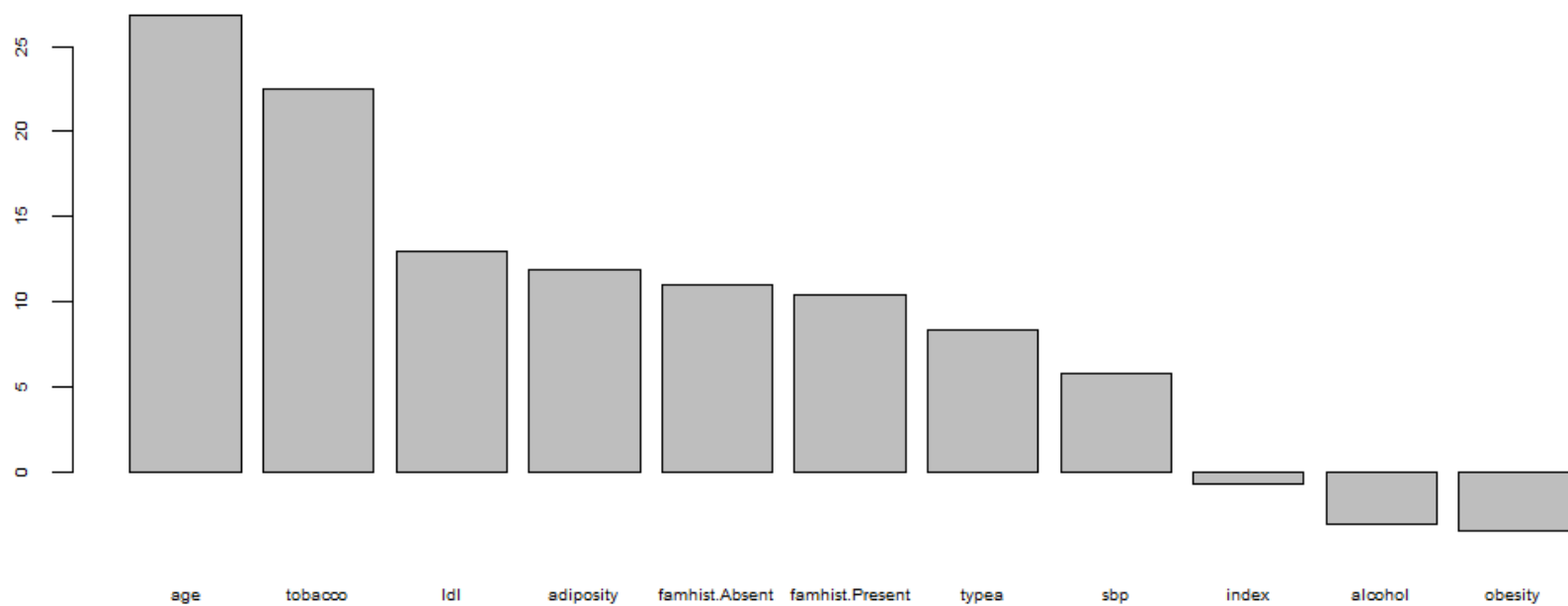
```
final<-rf$finalModel
```

```
tabla<-as.data.frame(importance(final))
```

```
tabla<-tabla[order(-tabla$MeanDecreaseAccuracy),]
```

```
tabla
```

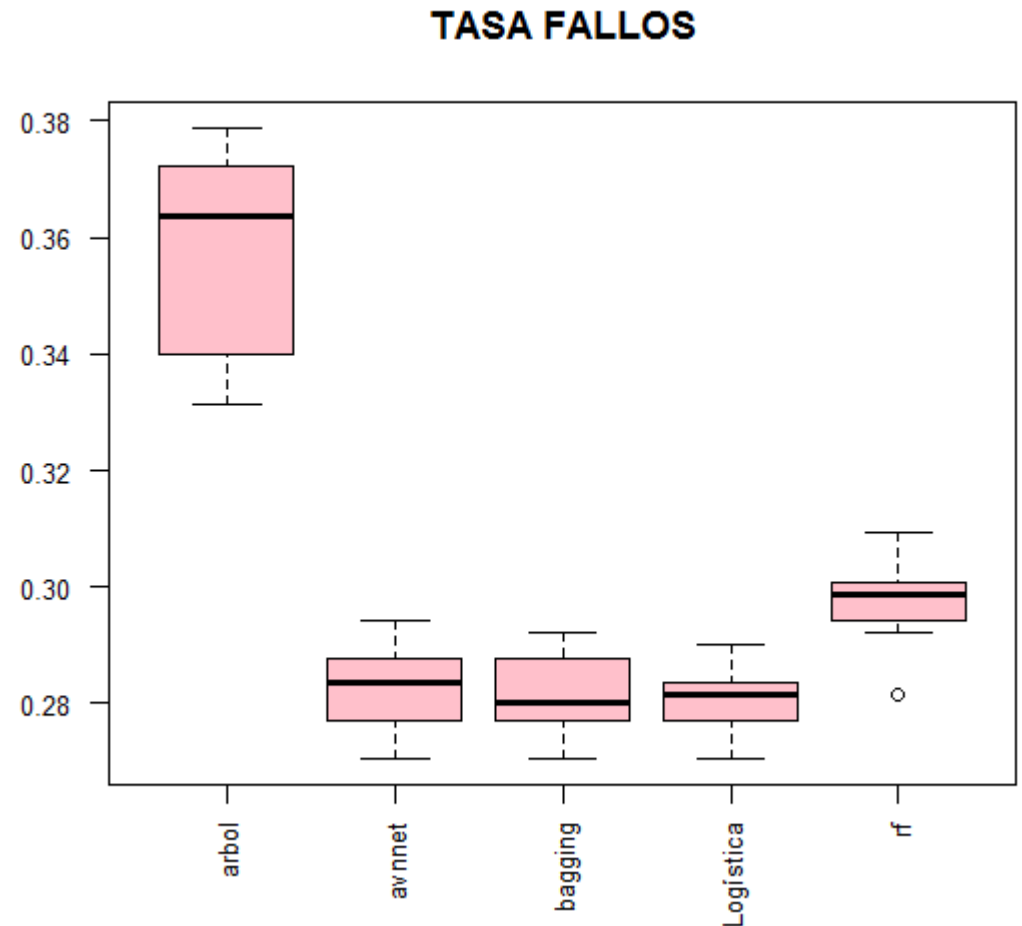
```
barplot(tabla$MeanDecreaseAccuracy,names.arg=rownames(tabla))
```



```
# La función cruzadarfbn permite plantear random forest

load ("saheartbis.Rda")
source ("cruzadas avnnet y log binaria.R")
source ("cruzada arbolbin.R")
source ("cruzada rf binaria.R")

medias1<-cruzadalogistica(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""), grupos=10,sinicio=1234, repe=10)
medias1$modelo="Logística"
medias2<-cruzadaavnnetbin(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),grupos=10,sinicio=1234, repe=10,
  size=c(5),decay=c(0.1),repeticiones=5,itera=200)
medias2$modelo="avnnet"
medias3<-cruzadaarbolbin(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),grupos=10,sinicio=1234, repe=10,
  cp=c(0),minbucket =5)
medias3$modelo="arbol"
medias4<-cruzadarfbn(data=saheartbis, vardep="chd",
  listconti=listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=10,sinicio=1234, repe=10,nodesize=10,
  mtry=6,ntree=3000,replace=TRUE,sampsize=150)
medias4$modelo="bagging"
  medias5<-cruzadarfbn(data=saheartbis, vardep="chd",
    listconti=listconti=c("age", "tobacco", "ldl",
      "adiposity", "typea", "famhist.Absent"),
    listclass=c(""),
    grupos=10,sinicio=1234, repe=10,nodesize=10,
    mtry=4,ntree=3000,replace=TRUE)
medias5$modelo="rf"
union1<-rbind(medias1,medias2,medias3,medias4,medias5)
par(cex.axis=0.8)
boxplot(data=union1,tasa~modelo,main="TASA FALLOS",col="pink")
boxplot(data=union1,auc~modelo,main="AUC")
```



# Ejemplo variable continua dependiente

```
# EJEMPLO CON VARIABLE CONTINUA
```

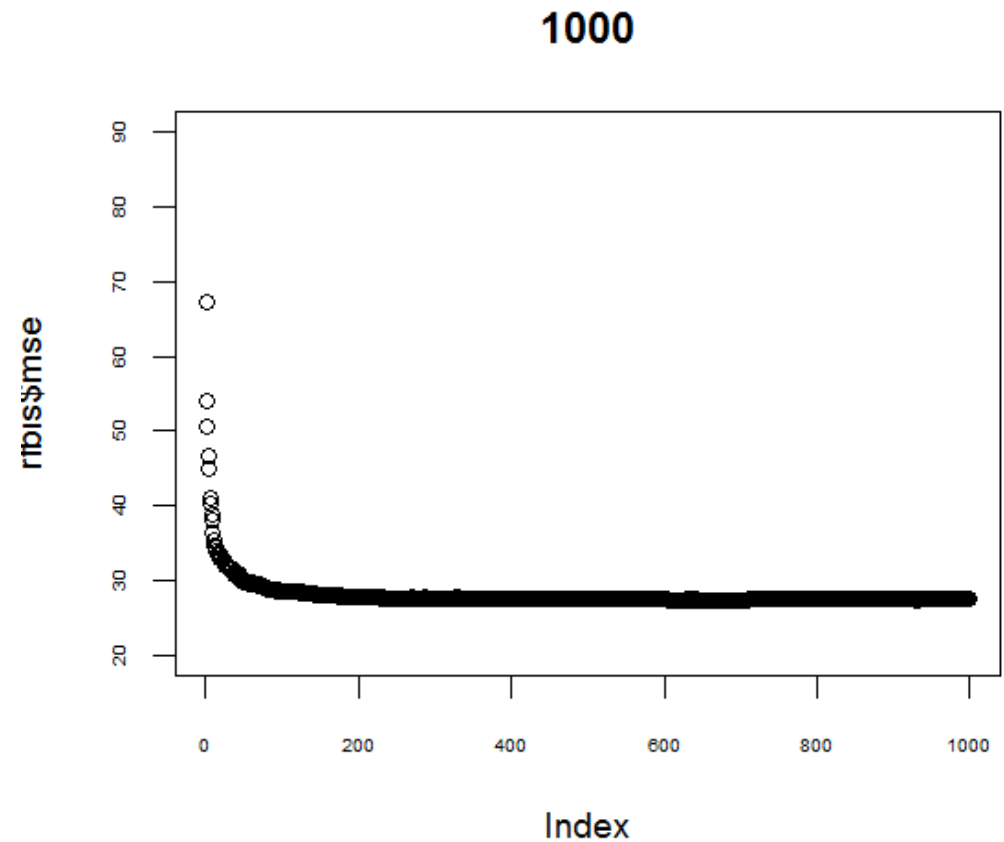
```
load("compressbien.Rda")
```

```
library(randomForest)
```

```
rfbis<-randomForest(cstrength~age+water+cement+blast,  
  data=compressbien,  
  mtry=2,ntree=1000,samplesize=300,nodesize=10,replace=TRUE)
```

```
# En continuas No estoy seguro de que  
# este gráfico esté realizado sobre oob  
# pero puede valer para decidir tamaño e iteraciones
```

```
plot(rfbis$mse)
```



## Tuneado del número de variables mtry

# TUNEADO CON CARET DEL NÚMERO DE VARIABLES A SORTEAR EN CADA NODO

```
set.seed(12345)
rfgrid<-expand.grid(mtry=c(2,3,4))

control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)
```

```
rf<- train(cstrength~age+water+cement+blast,
  data=compressbien,
  method="rf",trControl=control,tuneGrid=rfgrid,
  ntree=1000,samplesize=600,nodesize=10,replace=TRUE,
  importance=TRUE)
```

rf

# Recomienda bagging (mtry=4). A pesar de ello, comprobaremos vía validación  
# cruzada repetida

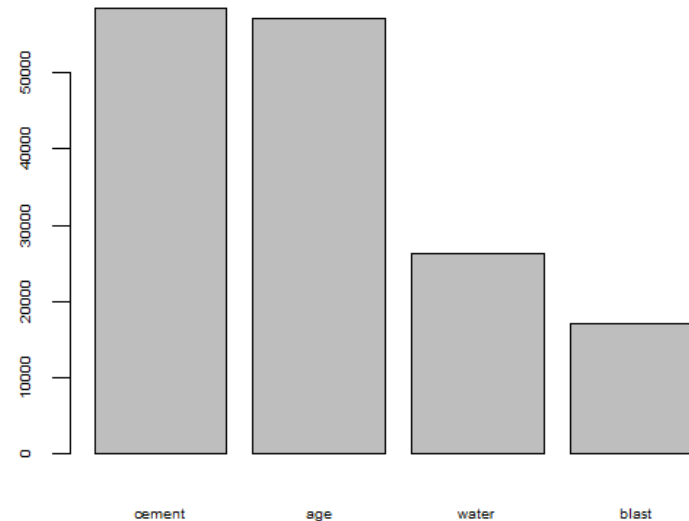
Recomienda mtry=4, coincidiendo con bagging. Pero probaremos igualmente mtry=2 en cv repetida

# IMPORTANCIA DE VARIABLES

```
final<-rf$finalModel
```

```
tabla<-as.data.frame(importance(final))
tabla<-tabla[order(-tabla$IncNodePurity),]
tabla
```

```
barplot(tabla$IncNodePurity,names.arg=rownames(tabla))
```



```
data<-compressbien
```

```
medias1<-cruzadaavnnet (data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),grupos=4,sinicio=1234,repe=5,
size=c(15),decay=c(0.01),repeticiones=5,itera=100)
medias1$modelo="avnnet"
medias2<-cruzadalín (data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),grupos=4,sinicio=1234,repe=5)
medias2$modelo="lineal"
medias3<-cruzadaarbol (data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),
grupos=4,sinicio=1234,repe=5,cp=0,minbucket=5)
medias3$modelo="arbol"
```

```
medias4<-cruzadarf (data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),
grupos=4,sinicio=1234,repe=5,
nodesize=10,replace=TRUE,ntree=200,mtry=4)
medias4$modelo="bagging"
```

```
medias5<-cruzadarf (data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),
grupos=4,sinicio=1234,repe=5,
nodesize=10,replace=TRUE,ntree=600,mtry=3)
```

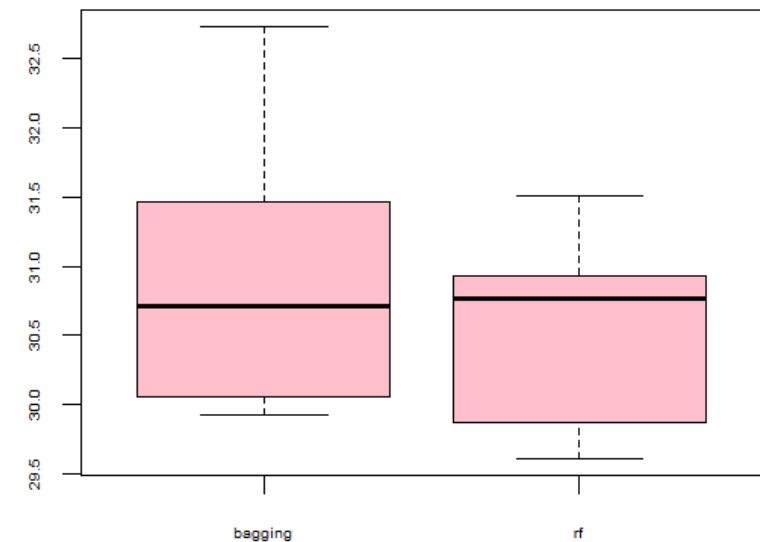
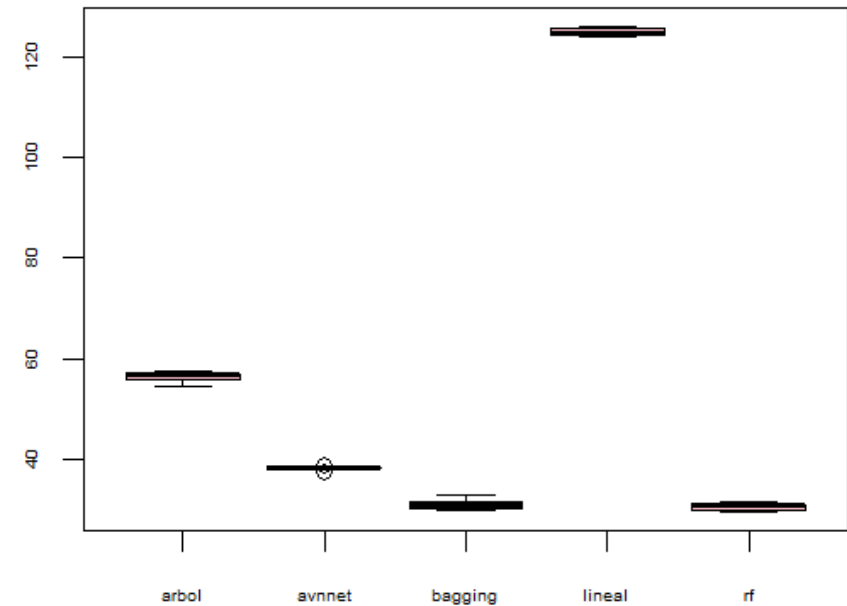
```
medias5$modelo="rf"
```

```
union1<-rbind(medias1,medias2,medias3,medias4,medias5)
```

```
par(cex.axis=0.5)
boxplot (data=union1,error~modelo,col="pink")
```

```
union1<-rbind(medias4,medias5)
```

```
par(cex.axis=0.5)
boxplot (data=union1,error~modelo,col="pink")
```



Se ve como randomForest reduce la varianza del modelo respecto a bagging. Ejercicio: manipular sampsize.



## Tercer intento importante: **Gradient Boosting (Friedman, 2001)**

Se basa en ir actualizando las predicciones en la **dirección de decrecimiento dada por el negativo del gradiente, de la función de error  $L(y_i, f(x_i))$**  (paso (a)). La función  $f(x_i)$  es la función de predicción de  $y_i$  basada en los valores  $x_i$ .

---

**Algorithm 10.3** *Gradient Tree Boosting Algorithm.*

---

1. Initialize  $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$ .

2. For  $m = 1$  to  $M$ :

(a) For  $i = 1, 2, \dots, N$  compute

$$r_{im} = - \left[ \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}.$$

(b) Fit a regression tree to the targets  $r_{im}$  giving terminal regions  $R_{jm}$ ,  $j = 1, 2, \dots, J_m$ .

(c) For  $j = 1, 2, \dots, J_m$  compute

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma).$$

(d) Update  $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$ .

3. Output  $\hat{f}(x) = f_M(x)$ .

---

Algunas funciones de error (la función  $f(x)$  es la función de predicción o función base según el método que utilicemos (en general, árboles pero puede utilizarse cualquiera)):

**TABLE 10.2.** *Gradients for commonly used loss functions.*

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$ y_i - f(x_i) $	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $ y_i - f(x_i)  \leq \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $ y_i - f(x_i)  > \delta_m$ where $\delta_m = \alpha \text{th-quantile}\{ y_i - f(x_i) \}$
Classification	Deviance	$k$ th component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

# Algoritmo Gradient Boosting para regresión (función de error SCE)

1) Dar como valor predictivo de la variable  $y$ , para cada observación, la media de los valores de la variable  $y$ . Este será el punto de partida, y en cada iteración del algoritmo la predicción de  $y$  para cada observación será actualizada de manera individual.

$$\hat{y}_i^{(0)} = \bar{y}$$

2) Repetir los pasos siguientes para cada iteración  $m$ :

i) Calcular el residuo actual  $r_i^{(m)} = y_i - \hat{y}_i^{(m)}$

ii) Construir un árbol de regresión para predecir los residuos, tomando  $r_i^{(m)}$  como variable dependiente u objetivo, y el conjunto de las variables X input como independientes.

iii) Actualizar la predicción de  $y$  para cada observación (incluidas las observaciones de datos test), en la dirección de decrecimiento.

$$\hat{y}_i^{(m+1)} = \hat{y}_i^{(m)} + v \cdot \hat{r}_i^{(m)}$$

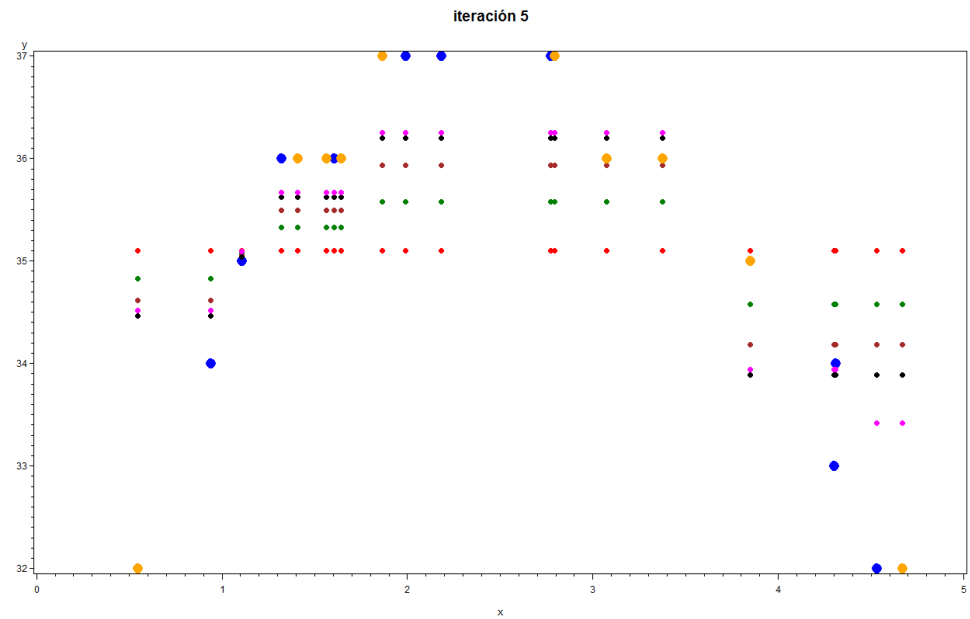
3) El proceso se detiene cuando se llega al número de iteraciones final deseado.

A determinar:

- \* Especificaciones generales para los árboles (número de hojas finales, ... )
- \* Parámetro de regularización  $v$
- \* Número de iteraciones

# Ejemplo

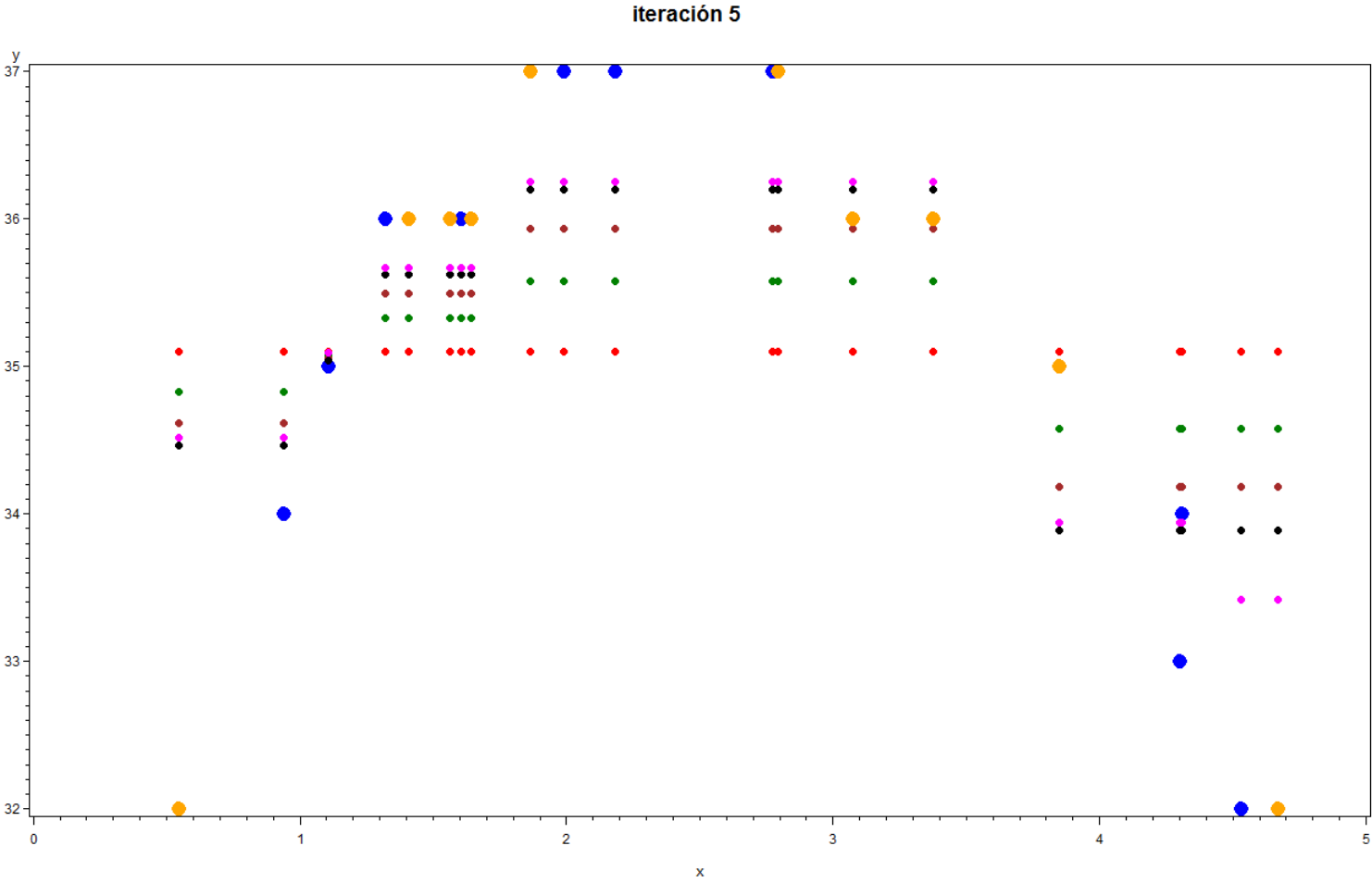
y	ytest	x	media	resi1	resi1_ est	y1	resi2	resi2_ est	y2	resi3	resi3_ est	y3	resi4	resi4_ est	y4
34	.	0.93843	35.1	-1.1	-1.1	34.825	-0.825	-0.825	34.6188	-0.61875	-0.61875	34.4641	-0.46406	0.20955	34.5164
35	.	1.10557	35.1	-0.1	-0.1	35.075	-0.075	-0.075	35.0563	-0.05625	-0.05625	35.0422	-0.04219	0.20955	35.0946
36	.	1.31851	35.1	0.9	0.9	35.325	0.675	0.675	35.4938	0.50625	0.50625	35.6203	0.37969	0.20955	35.6727
36	.	1.60456	35.1	0.9	0.9	35.325	0.675	0.675	35.4938	0.50625	0.50625	35.6203	0.37969	0.20955	35.6727
37	.	1.99041	35.1	1.9	1.9	35.575	1.425	1.425	35.9313	1.06875	1.06875	36.1984	0.80156	0.20955	36.2508
37	.	2.18037	35.1	1.9	1.9	35.575	1.425	1.425	35.9313	1.06875	1.06875	36.1984	0.80156	0.20955	36.2508
37	.	2.77429	35.1	1.9	1.9	35.575	1.425	1.425	35.9313	1.06875	1.06875	36.1984	0.80156	0.20955	36.2508
33	.	4.30212	35.1	-2.1	-2.1	34.575	-1.575	-1.575	34.1813	-1.18125	-1.18125	33.8859	-0.88594	0.20955	33.9383
34	.	4.30672	35.1	-1.1	-2.1	34.575	-0.575	-1.575	34.1813	-0.18125	-1.18125	33.8859	0.11406	0.20955	33.9383
32	.	4.53017	35.1	-3.1	-2.1	34.575	-2.575	-1.575	34.1813	-2.18125	-1.18125	33.8859	-1.88594	-1.88594	33.4145
.	32	0.54394	35.1	.	-1.1	34.825	.	-0.825	34.6188	.	-0.61875	34.4641	.	0.20955	34.5164
.	32	4.66699	35.1	.	-2.1	34.575	.	-1.575	34.1813	.	-1.18125	33.8859	.	-1.88594	33.4145
.	35	3.84978	35.1	.	-2.1	34.575	.	-1.575	34.1813	.	-1.18125	33.8859	.	0.20955	33.9383
.	36	1.40883	35.1	.	0.9	35.325	.	0.675	35.4938	.	0.50625	35.6203	.	0.20955	35.6727
.	36	1.56117	35.1	.	0.9	35.325	.	0.675	35.4938	.	0.50625	35.6203	.	0.20955	35.6727
.	36	1.64375	35.1	.	0.9	35.325	.	0.675	35.4938	.	0.50625	35.6203	.	0.20955	35.6727
.	36	3.07427	35.1	.	1.9	35.575	.	1.425	35.9313	.	1.06875	36.1984	.	0.20955	36.2508
.	36	3.37434	35.1	.	1.9	35.575	.	1.425	35.9313	.	1.06875	36.1984	.	0.20955	36.2508
.	37	1.86322	35.1	.	1.9	35.575	.	1.425	35.9313	.	1.06875	36.1984	.	0.20955	36.2508
.	37	2.79205	35.1	.	1.9	35.575	.	1.425	35.9313	.	1.06875	36.1984	.	0.20955	36.2508



Datos train: azul  
Datos test: naranja

La primera iteración son los puntos rojos, se predice y por la media,  $y_0 = \text{media}$   
 Para la segunda iteración:  
 a) Se calcula el residuo  $\text{resi1}$ .  
 b) Se predice  $\text{resi1}$  con un árbol con  $\text{resi1}$  dependiente,  $x$  independiente dando lugar a la predicción  $\text{resi1\_est}$   
 c) Se predice  $y$  como  $y_1 = y_0 + v * \text{resi1\_est}$   
 Para iteraciones sucesivas se vuelve a calcular el residuo y se continúa con el mismo proceso.

y	ytest	x	media	resi1	resi1_est	y1
34	.	0.93843	35.1	-1.1	-1.1	34.825
35	.	1.10557	35.1	-0.1	-0.1	35.075
36	.	1.31851	35.1	0.9	0.9	35.325
36	.	1.60456	35.1	0.9	0.9	35.325
37	.	1.99041	35.1	1.9	1.9	35.575
37	.	2.18037	35.1	1.9	1.9	35.575
37	.	2.77429	35.1	1.9	1.9	35.575
33	.	4.30212	35.1	-2.1	-2.1	34.575
34	.	4.30672	35.1	-1.1	-2.1	34.575
32	.	4.53017	35.1	-3.1	-2.1	34.575
.	32	0.54394	35.1	.	-1.1	34.825
.	32	4.66699	35.1	.	-2.1	34.575

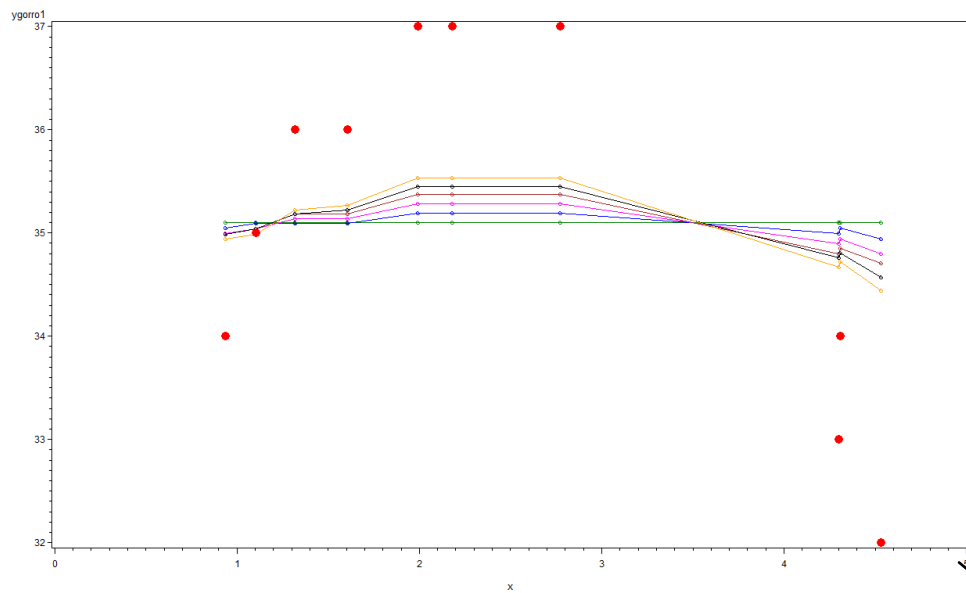


Datos train: azul  
 Datos test: naranja

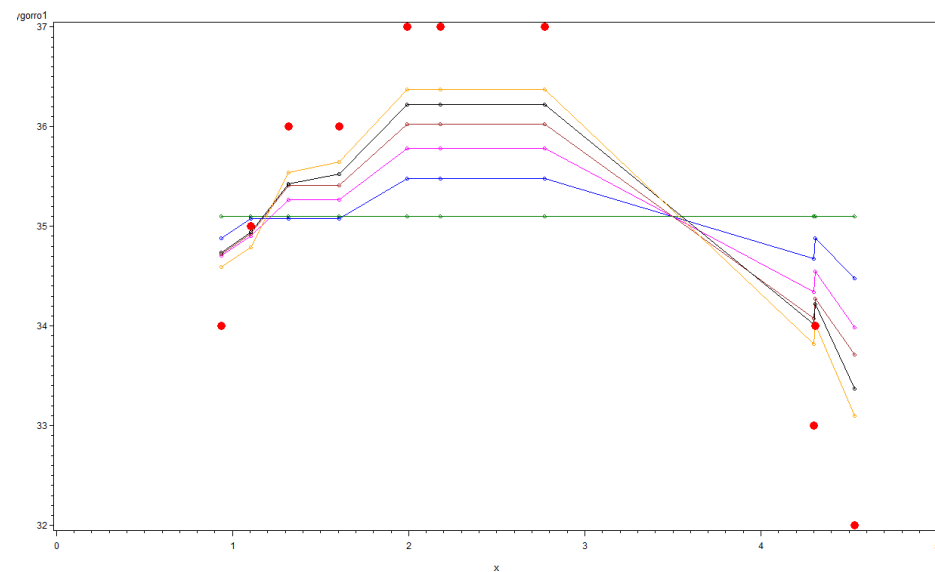
A medida que aumentan las iteraciones, los datos train convergen al valor real de  $y$ , con error cero (las medidas de “ajuste” no tienen sentido aquí).

Lo importante es qué tal se comportan las predicciones sobre datos test.

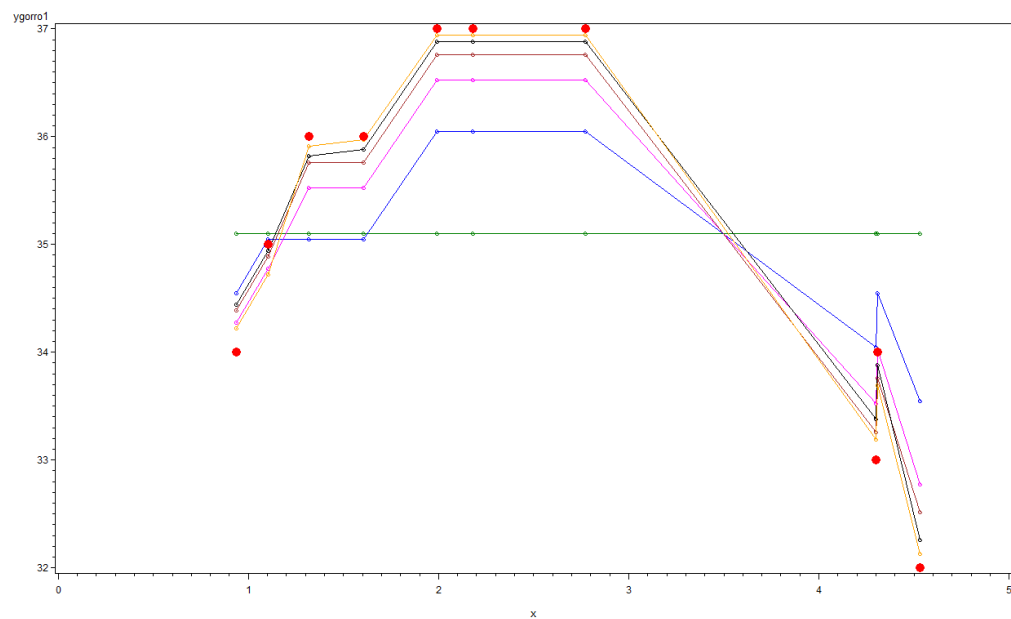
SHRINKAGE=0.05



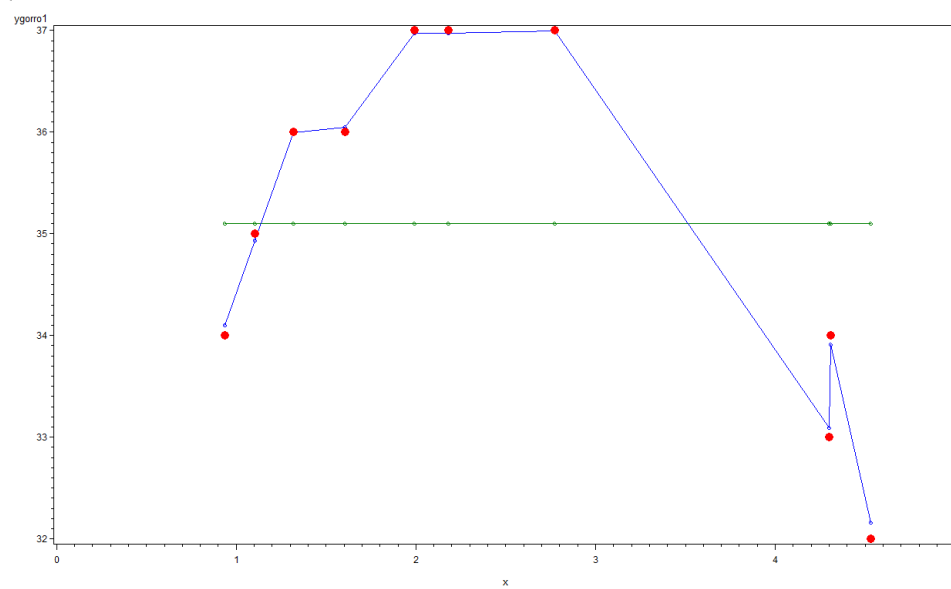
SHRINKAGE=0.20



SHRINKAGE=0.50



SHRINKAGE=0.05, 100 iteraciones



# Algoritmo Gradient Boosting para clasificación binaria

En este caso se utiliza la función logit como función base, y la Deviance como función de error. El objetivo es ir retocando la función logit y en cada paso del algoritmo se actualizan las probabilidades predichas y los residuos.

La función de error en este caso es:

$$L(y_i, f(x_i)) = \log(1 + e^{-2y_i f(x_i)})$$

donde la variable dependiente es binaria:  $y_i = 1, 0$

La función  $f(x_i)$  es la función logit y se define como  $f(x_i) = \frac{1}{2} \log \left( \frac{\hat{p}_i^{(m)}}{1 - \hat{p}_i^{(m)}} \right)$  con  $p_i = P(y_i = 1)$

1) Se toma como valor inicial para la probabilidad predicha de 1 en todas las observaciones el porcentaje de 1 en la muestra:

$$\hat{p}_i^{(0)} = \% \text{ de observaciones con } y = 1$$

$$\hat{f}_i^{(0)} = \frac{1}{2} \log \left( \frac{\hat{p}_i^{(0)}}{1 - \hat{p}_i^{(0)}} \right)$$

2) Repetir los pasos siguientes para cada iteración  $m$ :

i) Calcular el residuo actual  $r_i^{(m)} = y_i - \hat{p}_i^{(m)}$

ii) Construir un árbol de regresión para predecir los residuos, tomando  $r_i^{(m)}$  como variable dependiente u objetivo, y el conjunto de las variables X input como independientes.

iii) Actualizar la predicción de la función logit  $f$  para cada observación de la siguiente manera:

$$\hat{f}_i^{(m+1)} = \hat{f}_i^{(m)} + v \cdot \hat{r}_i^{(m)} = \frac{1}{2} \log \left( \frac{\hat{p}_i^{(m)}}{1 - \hat{p}_i^{(m)}} \right) + v \cdot \hat{r}_i^{(m)}$$

iv) Actualizar la predicción de las probabilidades mediante

$$\hat{p}_i^{(m+1)} = \frac{1}{1 + e^{-2\hat{f}_i^{(m+1)}}}$$

3) El proceso se detiene cuando se llega al número de iteraciones final deseado.



# Stochastic Gradient Boosting

Es una pequeña modificación para luchar contra el sobreajuste y alta varianza, en la línea de bagging-random forest: Se seleccionaría en cada iteración (cada árbol creado), una muestra diferente de los datos de entrenamiento para construir el árbol.

## En resumen:

El algoritmo gradient boosting consiste en repetir la construcción de árboles de regresión/clasificación, modificando ligeramente las predicciones iniciales cada vez, intentando ir minimizando los residuos en la dirección de decrecimiento.

Al plantear diferentes árboles cada vez, el proceso va ajustando las predicciones cada vez más a los datos, y de alguna manera unos árboles corrigen a otros con lo cual la flexibilidad y adaptación del método mejora respecto a la construcción de un único árbol.

La convergencia suele ser lenta y van a la par los errores en training y validación (no sobreajuste).

# Principales parámetros a controlar en Gradient Boosting

- La constante de regularización  $\nu$  (shrink) . Normalmente entre (0.001 y 0.2). Cuanto más alta, más rápido converge pero demasiado alta es poco fino. Si se pone muy baja (la recomendación teórica) hay que poner muchas iteraciones para que converja. En la práctica se comienza con valores altos para observar resultados básicos y cuando se controla bien el proceso el modelo final se realiza con valores bajos de  $\nu$  y muchas iteraciones.
- El número de iteraciones **m**. A menor  $\nu$ , serán necesarias más iteraciones  $m$ . Es un parámetro a monitorizar (con validación cruzada y gráficos preferentemente), pues teoría y práctica coinciden en que a partir de un punto se puede producir sobreajuste. La decisión sobre el valor  $m$  se denomina **early stopping**. En muchos casos prácticos no es necesario, pero es conveniente estudiarlo.
- Características de los árboles. Son bastante influyentes:
  - El número de hojas final  $o$ , en su defecto, la profundidad del árbol
  - El maxbranch (número de divisiones máxima en cada nodo. Por defecto se dejará en 2, árboles binarios)
  - El número de observaciones mínimo en una rama-nodo. Se puede ampliar para evitar sobreajuste (reducir varianza) o reducir para ajustar mejor (reducir sesgo).

**Los parámetros a utilizar en gradient boosting son interdependientes, conviene tener esto en cuenta**

## **Ventajas del Gradient Boosting:**

- 1) Invariante frente a transformaciones monótonas: no es necesario realizar transformaciones logarítmicas, etc.
- 2) Buen tratamiento de missing, variables categóricas, etc. Universalidad.
- 3) Muy fácil de implementar, relativamente pocos parámetros a monitorizar (número de hojas o profundidad del árbol, tamaño final de hojas, parámetro de regularización...).
- 4) Gran eficacia predictiva, algoritmo muy competitivo. En Kaggle, aproximadamente el 80% de los concursantes lo usa, exclusivamente o combinado con otras técnicas. Supera a menudo al algoritmo Random Forest.
- 5) Robusto respecto a variables irrelevantes. Robusto respecto a colinealidad. Detecta interacciones ocultas.

## Desventajas del Gradient Boosting:

Como todos los métodos basados en árboles, dependiendo de los datos puede ser superado por otras técnicas más sencillas.

- A mayor **complejidad** de los datos (interacciones, missing, no linealidad, muchas variables categóricas, muchas variables en general), es más posible que el algoritmo gradient boosting supere a otros algoritmos (por ello en Kaggle se utiliza más, puesto que suelen ser problemas complejos).
- Por el contrario, en datos relativamente **sencillos** (pocas variables, no missing, no interacciones, linealidad (regresión) o separabilidad lineal (clasificación)), el gradient boosting (o random forest) no tiene nada nuevo que aportar y pueden ser preferibles modelos sencillos (regresión, regresión logística, discriminante) o modelos ad-hoc que adapten aspectos concretos como la no linealidad (redes por ejemplo).

# Gradient Boosting con caret

## Parámetros básicos:

- shrinkage (parámetro  $\nu$  de regularización, mide la velocidad de ajuste, a menor  $\nu$ , más lento y necesita más iteraciones, pero es más fino en el ajuste)
- n.minobsnode: tamaño máximo de nodos finales (el principal parámetro que mide la complejidad)
- n.trees=el número de iteraciones (árboles)
- interaction.depth (2 para árboles binarios)

## Ejemplo variable dependiente binaria

```
library(caret)

set.seed(12345)

gbmgrid<-expand.grid(shrinkage=c(0.1,0.05,0.03,0.01,0.001),
  n.minobsinnode=c(5,10,20),
  n.trees=c(100,500,1000,5000),
  interaction.depth=c(2))

control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)

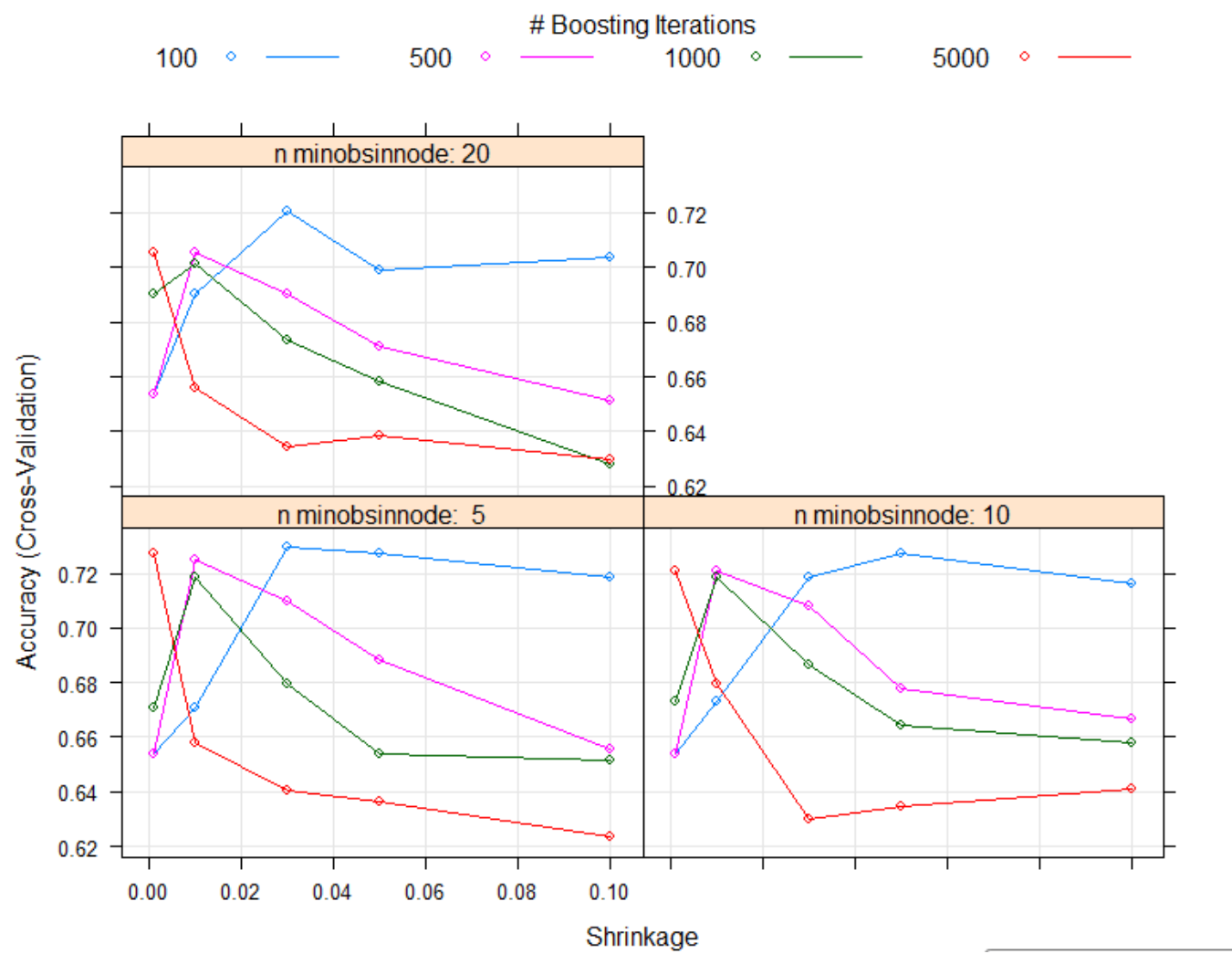
gbm<- train(factor(chd)~.,data=saheartbis,
  method="gbm",trControl=control,tuneGrid=gbmgrid,
  distribution="bernoulli", bag.fraction=1,verbose=FALSE)

gbm
plot(gbm)
```

shrinkage	n.minobsinnode	n.trees	Accuracy	Kappa
0.001	5	100	0.6536732	0.000000000
0.001	5	500	0.6536732	0.000000000
0.001	5	1000	0.6709895	0.088046491
0.001	5	5000	0.7272864	0.349343532
0.001	10	100	0.6536732	0.000000000
0.001	10	500	0.6536732	0.000000000
0.001	10	1000	0.6731634	0.095732268
0.001	10	5000	0.7207646	0.328810148
0.001	20	100	0.6536732	0.000000000
0.001	20	500	0.6536732	0.007383966
0.001	20	1000	0.6905360	0.166746708
0.001	20	5000	0.7055847	0.300499330
0.010	5	100	0.6709895	0.088046491
0.010	5	500	0.7251124	0.343471846
0.010	5	1000	0.7186094	0.340219890
0.010	5	5000	0.6579085	0.205789769
0.010	10	100	0.6731634	0.095732268
0.010	10	500	0.7207834	0.330415347
0.010	10	1000	0.7186094	0.338187545
0.010	10	5000	0.6797601	0.267649859
0.010	20	100	0.6905360	0.166746708
0.010	20	500	0.7055847	0.300499330
0.010	20	1000	0.7012556	0.302232854
0.010	20	5000	0.6559220	0.205433524
0.030	5	100	0.7294228	0.336125192
0.030	5	500	0.7099700	0.326386902
0.030	5	1000	0.6796477	0.266693801
0.030	5	5000	0.6406672	0.169074854
0.030	10	100	0.7186094	0.313015068
0.030	10	500	0.7078148	0.317972004
0.030	10	1000	0.6862069	0.277568997
0.030	10	5000	0.6299663	0.162100618
0.030	20	100	0.7207834	0.316335222
0.030	20	500	0.6904985	0.279802542
0.030	20	1000	0.6731634	0.246165849
0.030	20	5000	0.6343141	0.159764818
0.050	5	100	0.7272489	0.348416320
0.050	5	500	0.6883621	0.280858752
0.050	5	1000	0.6535795	0.197993734
0.050	5	5000	0.6364318	0.170145910
0.050	10	100	0.7272676	0.348602370
0.050	10	500	0.6775487	0.254496143
0.050	10	1000	0.6645615	0.233599581
0.050	10	5000	0.6343328	0.173034137
0.050	20	100	0.6991004	0.281753284
0.050	20	500	0.6710082	0.249823367

shrinkage	n.minobsinnode	n.trees	Accuracy	Kappa
0.050	20	1000	0.6580960	0.211164963
0.050	20	5000	0.6385870	0.173543102
0.100	5	100	0.7185532	0.338391308
0.100	5	500	0.6557909	0.204592627
0.100	5	1000	0.6514993	0.203389984
0.100	5	5000	0.6234633	0.138804302
0.100	10	100	0.7164730	0.331909854
0.100	10	500	0.6667354	0.234249839
0.100	10	1000	0.6580960	0.210956777
0.100	10	5000	0.6407984	0.192749956
0.100	20	100	0.7034670	0.309913460
0.100	20	500	0.6515555	0.190934530
0.100	20	1000	0.6277361	0.139825291
0.100	20	5000	0.6299663	0.162885869

Tuning parameter 'interaction.depth' was held constant at a value of 2  
Accuracy was used to select the optimal model using the largest value.  
The final values used for the model were n.trees = 100, interaction.depth  
= 2, shrinkage = 0.03 and n.minobsinnode = 5.



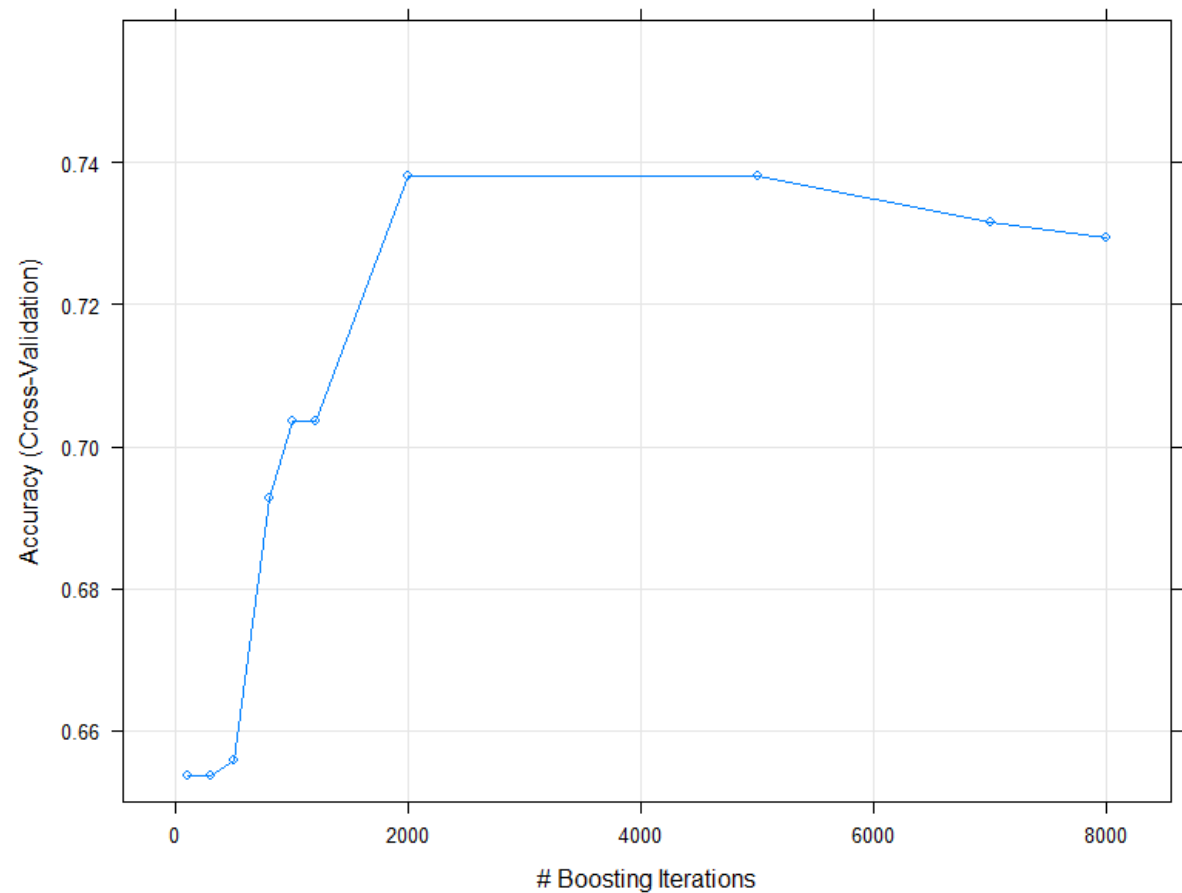
```
# ESTUDIO DE EARLY STOPPING
# Probamos a fijar algunos parámetros para ver como evoluciona
# en función de las iteraciones
```

```
gbmgrid<-expand.grid(shrinkage=c(0.001),
  n.minobsinnode=c(10),
  n.trees=c(100,300,500,800,1000,1200,2000,5000,7000,8000),
  interaction.depth=c(2))
```

```
control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)
```

```
gbm<- train(factor(chd)~.,data=saheartbis,
  method="gbm",trControl=control,tuneGrid=gbmgrid,
  distribution="bernoulli", bag.fraction=1,verbose=FA
```

```
plot(gbm,ylim=c(0.65,0.76))
```

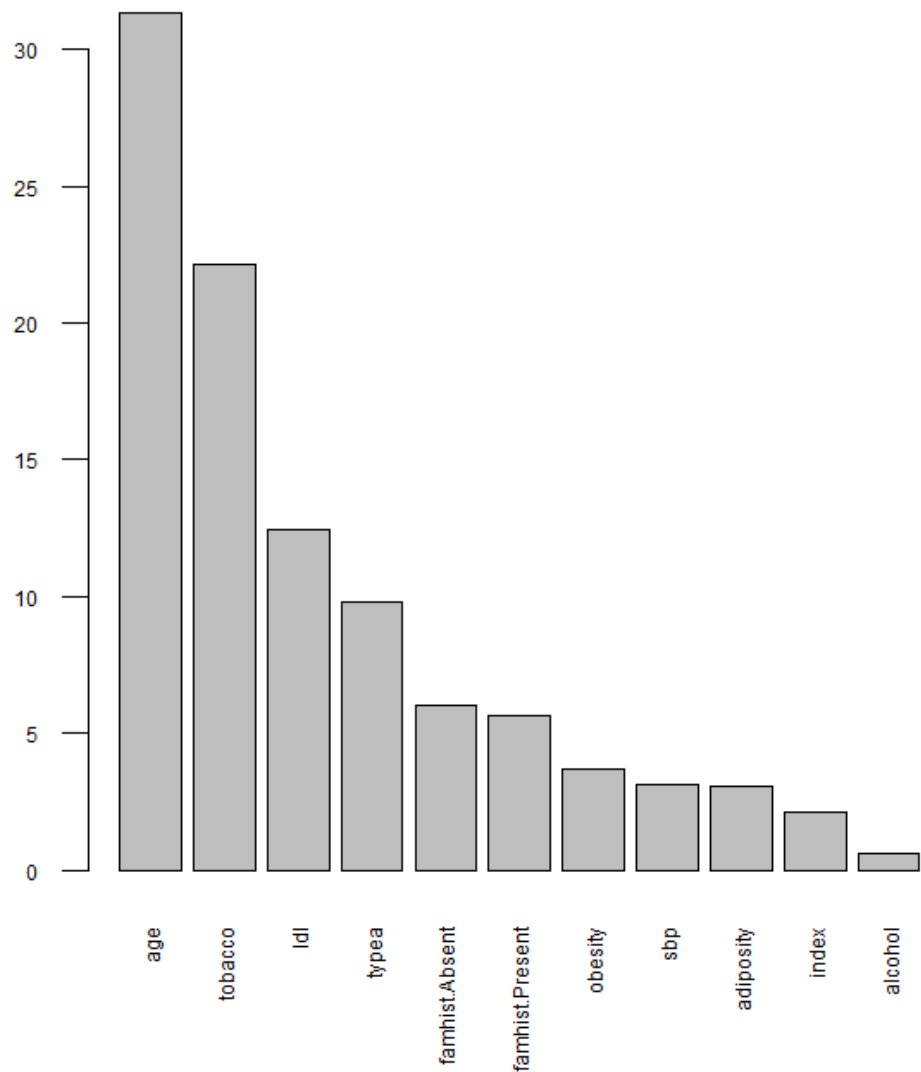




# IMPORTANCIA DE VARIABLES

```
summary(gbm)
tabla<-summary(gbm)
par(cex=1.5,las=2)
barplot(tabla$rel.inf,names.arg=row.names(tabla))
```

	var	rel.inf
age	age	31.3572318
tobacco	tobacco	22.1651237
ldl	ldl	12.4591794
typea	typea	9.7732005
famhist.Absent	famhist.Absent	6.0332718
famhist.Present	famhist.Present	5.6688400
obesity	obesity	3.6711867
sbp	sbp	3.0982067
adiposity	adiposity	3.0428396
index	index	2.1035397
alcohol	alcohol	0.6273799



```
# La función cruzadagbmbin permite plantear gradient boosting para binarias
load ("saheartbis.Rda")
source ("cruzadas avnnet y log binaria.R")
source ("cruzada arbolbin.R")
source ("cruzada rf binaria.R")
source ("cruzada gbm binaria.R")

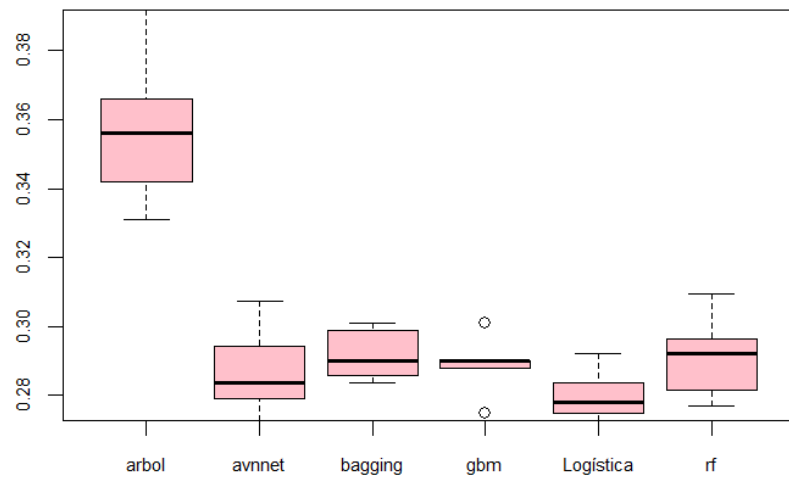
medias6<-cruzadagbmbin(data=saheartbis, vardep="chd",
  listconti=listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  n.minobsinnode=10, shrinkage=0.001, n.trees=2000, interaction.depth=2)

medias6$modelo="gbm"

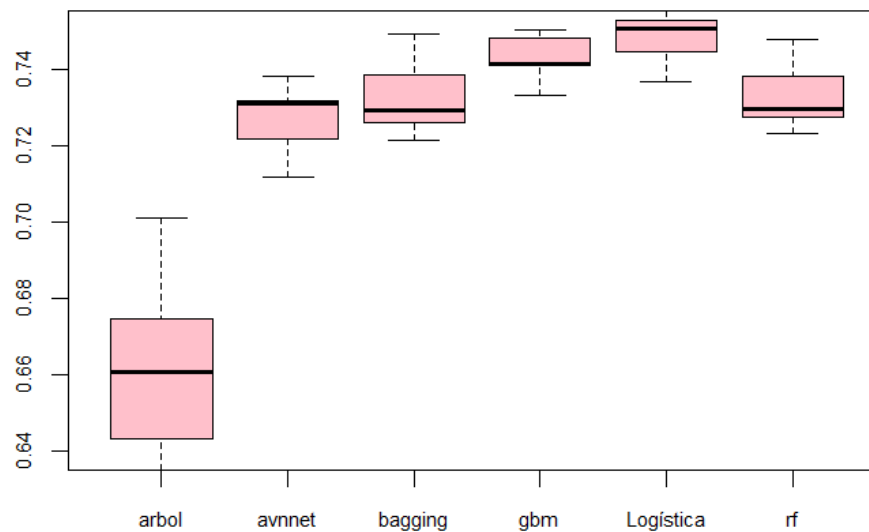
union1<-rbind(medias1,medias2,medias3,medias4,medias5,medias6)

par(cex.axis=0.8)
boxplot(data=union1,tasa~modelo,main="TASA FALLOS",col="pink")
boxplot(data=union1, auc~modelo,main="AUC")
```

**TASA FALLOS**



**AUC**



# Ejemplo variable dependiente continua

```
# EJEMPLO CON VARIABLE CONTINUA
```

```
load("compressbien.Rda")
```

```
gbmgrid<-expand.grid(shrinkage=c(0.1,0.05,0.03,0.01,0.001),  
  n.minobsinnode=c(5,10,20),  
  n.trees=c(100,500,1000,5000),  
  interaction.depth=c(2))
```

```
control<-trainControl(method = "cv",number=4,savePredictions = "all",  
  classProbs=TRUE)
```

```
gbm<- train(cstrength~age+water+cement+blast,data=compressbien,  
  method="gbm",trControl=control,tuneGrid=gbmgrid,  
  distribution="gaussian", bag.fraction=1,verbose=FALSE)
```

```
gbm
```

shrinkage	n.minobsinnode	n.trees	RMSE	Rsquared	MAE
...					
0.100	10	5000	4.673020	0.9224754	3.247772
...					

```
plot(gbm)
```

```
# ESTUDIO DE EARLY STOPPING
```

```
# Probamos a fijar algunos parámetros para ver como evoluciona  
# en función de las iteraciones
```

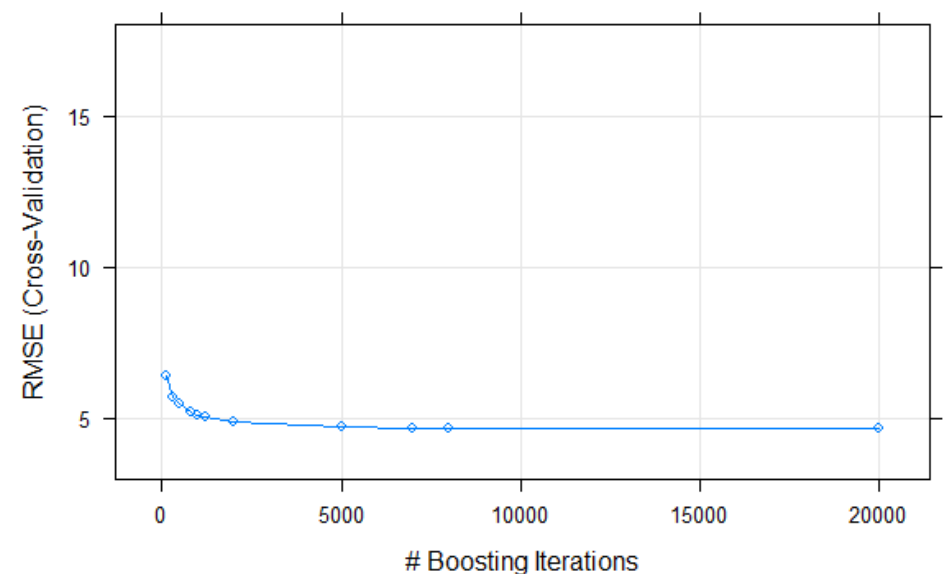
```
gbmgrid<-expand.grid(shrinkage=c(0.1),  
  n.minobsinnode=c(10),  
  n.trees=c(100,300,500,800,1000,1200,2000,5000,7000,8000,20000),  
  interaction.depth=c(2))
```

```
control<-trainControl(method = "cv",number=4,savePredictions = "all")
```

```
gbm<- train(cstrength~age+water+cement+blast,data=compressbien,  
  method="gbm",trControl=control,tuneGrid=gbmgrid,  
  distribution="gaussian", bag.fraction=1,verbose=FALSE)
```

```
gbm
```

```
plot(gbm,ylim=c(3,18))
```



## # IMPORTANCIA DE VARIABLES

```
summary(gbm)
```

## # ESTUDIO CV REPETIDA

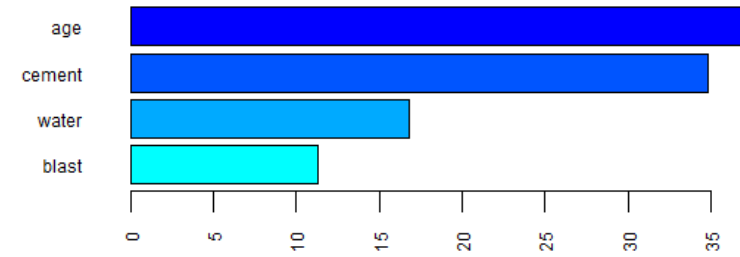
```
source("cruzada arbol continua.R")
source("cruzadas avnnet y lin.R")
source("cruzada rf continua.R")
source("cruzada gbm continua.R")
load("compressbien.Rda")
data<-compressbien
medias1<-cruzadaavnnet(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),grupos=4,sinicio=1234, repe=5,
  size=c(15),decay=c(0.01),repeticiones=5,itera=100)
medias1$modelo="avnnet"
medias2<-cruzadalin(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),grupos=4,sinicio=1234, repe=5)
medias2$modelo="lineal"
medias3<-cruzadaarbol(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5, cp=0, minbucket=5)
medias3$modelo="arbol"
medias4<-cruzadarf(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  nodesize=10, replace=TRUE, ntree=200, mtry=4)
medias4$modelo="bagging"
medias5<-cruzadarf(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  nodesize=10, replace=TRUE, ntree=600, mtry=3)
medias5$modelo="rf"

medias6<-cruzadagbm(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  n.minobsinnode=10, shrinkage=0.10, n.trees=20000, interaction.depth=2)

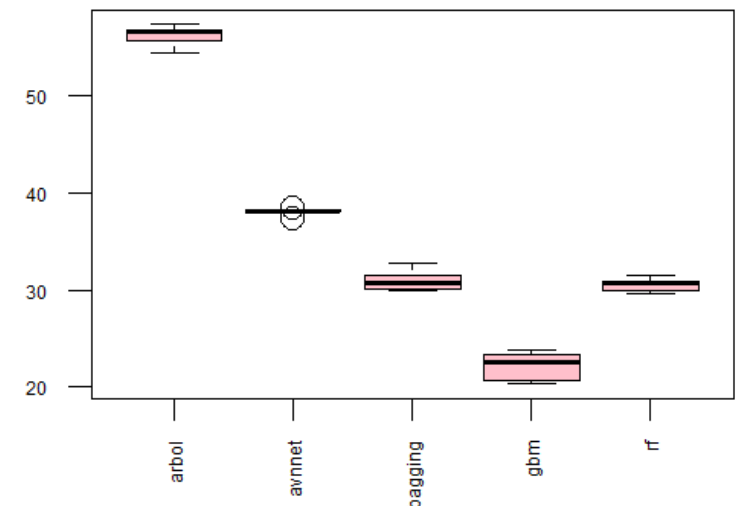
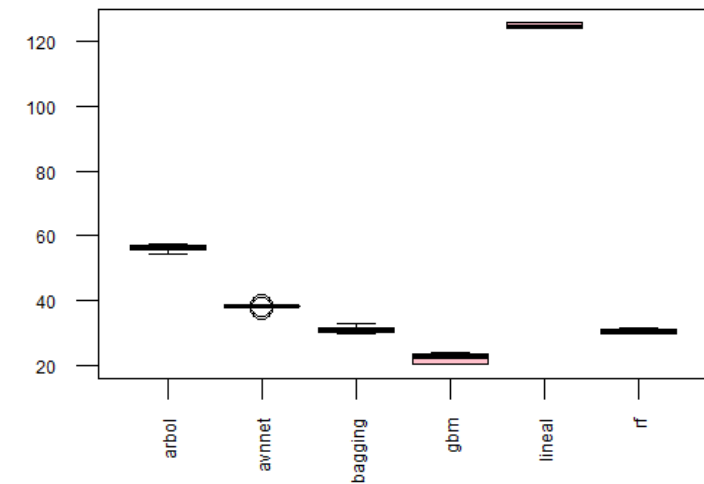
medias6$modelo="gbm"

union1<-rbind(medias1,medias2,medias3,medias4,medias5,medias6)

par(cex.axis=0.5)
boxplot(data=union1,error~modelo,col="pink")
union1<-rbind(medias1,medias3,medias4,medias5,medias6)
par(cex.axis=0.5)
boxplot(data=union1,error~modelo,col="pink")
```



Relative influence



## La última moda: Xgboost

En los concursos de Kaggle, desde hace 2-3 años el paquete-algoritmo más utilizado es el gbm (Gradient boosting machine), basado en el algoritmo gradient boosting básico.

Desde Febrero de 2016 existe el paquete **Xgboost**. Está basado en una modificación del Gradient boosting realizada ex profeso para ganar el concurso de Kaggle “High Boson Challenge”, un problema de clasificación (no ganaron pero quedaron en el 2% superior del leaderboard).

La principal corrección del algoritmo consiste en la utilización de la **regularización**.

# Regularización

Es una técnica orientada a la reducción de varianza de los errores (sobreajuste).

Para reducir el sobreajuste existen otros métodos:

- seleccionar modelos más sencillos
- early stopping
- utilizar validación cruzada para controlar la varianza del error
- ensamblados

La diferencia con la **regularización** es que ésta **interviene en la optimización** interna del algoritmo (en el proceso de estimación de parámetros).

El caso más conocido y antiguo de regularización es la **regresión Ridge**.

Cuando hay colinealidad (v. independientes muy correladas entre ellas) la estimación de los parámetros puede ser muy errática.

Por ello se introduce un término de penalización en la función objetivo que hace que estimar parámetros con valores más altos en conjunto sea peor:

En regresión normal:

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

En regresión Ridge se fija primero un parámetro lambda y se penalizan los valores altos de la suma de parámetros al cuadrado :

$$\hat{\beta}^{ridge} = \operatorname{argmin}_{\beta} \left\{ \sum_{i=1}^n \left( y_i - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p \beta_j^2 \right\}$$

De este modo, el carácter errático de los parámetros se ve corregido, obteniendo valores bajos y **más estables** para los betas. Otras extensiones de la regresión ridge son los métodos **Lasso** y **Elastic Net** (ambos tienen paquetes en R).

## Regularización en XGboost

En este algoritmo se modifica el gradient boosting a la hora de construir cada árbol con una función de penalización basada en el número de hojas y el score-predicción en cada hoja (que sería análogo al valor del parámetro en regresión):

Árboles más complejos= más hojas, más suma de cuadrados.

El algoritmo Xgboost prefija **dos parámetros principales de regularización**, lambda y alpha, que penalizan por los pesos w, score-predicción en cada hoja. Un tercero gamma (llamado lambda\_bias en el paquete) penaliza por el número de hojas Q.

$$\Omega(f_t) = \frac{1}{2} \lambda \left( \sum_{j=1}^Q w_j^2 \right) + \alpha \left( \sum_{i=1}^Q |w_i| \right) + \gamma Q$$



A la hora de construir cada árbol del gradient boosting, se tiene en cuenta esa penalización .

Se utiliza un algoritmo secuencial para evaluar como se hace cada división de manera que la función objetivo sea la habitual, pero penalizada por la función anterior.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

El algoritmo se programó en C++ y es muy rápido. El control del sobreajuste es algo que faltaba al Gradient Boosting.

## Ventajas del Xgboost:

- 1) Regularización. Una gran novedad, aunque hay que monitorizar los parámetros  $\lambda$ ,  $\alpha$ ,  $\lambda_{\text{bias}}$ . Sirven sobre todo para corregir la varianza del modelo. A mayores valores, más conservador: más sesgo, menos varianza.
- 2) El paquete está elaborado más universalmente y permite utilizar diferentes funciones objetivo. Es muy rápido y esa es otra razón por su uso en grandes bases de datos.
- 3) El algoritmo implementado sigue dividiendo un nodo aunque parezca malo, y después evalúa el árbol final. El gradient boosting normal se para en un nodo si es malo. Esto puede significar una gran diferencia.
- 4) El programa Xgboost incorpora también, aparte de la regularización, el control del sobreajuste utilizando las ideas de remuestreo del randomforest: tiene un parámetro de % de sorteo de observaciones y otro de sorteo de variables (como en randomforest pero no en cada nodo, sino antes de construir el árbol). Por esta flexibilidad es tan utilizado en concursos, a menudo una buena combinación de parámetros da muy buenos resultados.

## Desventajas del Xgboost:

- 1) Es otra manera de construir los árboles, no hay mucha teoría al respecto.
- 2) Es necesario monitorizar los parámetros de regularización. Hay riesgo de gran dependencia de éstos de los datos utilizados. Los resultados de utilizar regularización a menudo no tienen efecto o son demasiado erráticos.

Es posible que la raíz del éxito del xgboost no esté en la regularización en sí sino en el proceso de generación de árboles. El parámetro alpha parece más útil en general (pruebas mías, no suele tampoco ayudar demasiado).

- 3) Como consecuencia del punto anterior, es conveniente comparar vía CV el modelo sin regularización (los parámetros de regularización puestos a cero) y el modelo con parámetros de regularización si estos parecen útiles.
- 4) No siempre xgboost es mejor que el gbm al crear los árboles de otra manera.

El éxito de Xgboost está llevando al surgimiento de otras versiones: lightGBM (Microsoft) y Catboost (Yandex). Ambas tienen paquetes en R.

Muy buena explicación de xgboost y estas últimas alternativas:

<https://www.youtube.com/watch?v=5CWwwtEM2TA&t=18s>

Las últimas versiones de Xgboost incorporan el método “histograma” usado en lightGBM para acelerar el proceso. En realidad eso lo hace más rápido, pero no más preciso. No están en caret todavía pero sí en el paquete básico.

<https://github.com/dmlc/xgboost/issues/1950>

# Xgboost con caret

```
# TUNEADO DE XGBOOST CON CARET
```

```
# Caret permite tunear estos parámetros básicos:
```

```
#  
# nrounds (# Boosting Iterations)  
# max_depth (Max Tree Depth)  
# eta (Shrinkage)  
# gamma (Minimum Loss Reduction)  
# colsample_bytree (Subsample Ratio of Columns)  
# min_child_weight (Minimum Sum of Instance Weight)  
# subsample (Subsample Percentage)
```

```
library(caret)
```

```
set.seed(12345)
```

```
xgbmgrid<-expand.grid( min_child_weight=c(5,10,20), eta=c(0.1,0.05,0.03,0.01,0.001), nrounds=c(100,500,1000,5000),  
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)
```

```
control<-trainControl(method = "cv",number=4,savePredictions = "all", classProbs=TRUE)
```

```
xgbm<- train(factor(chd)~.,data=saheartbis, method="xgbTree",trControl=control,tuneGrid=xgbmgrid,verbose=FALSE)
```

```
xgbm
```

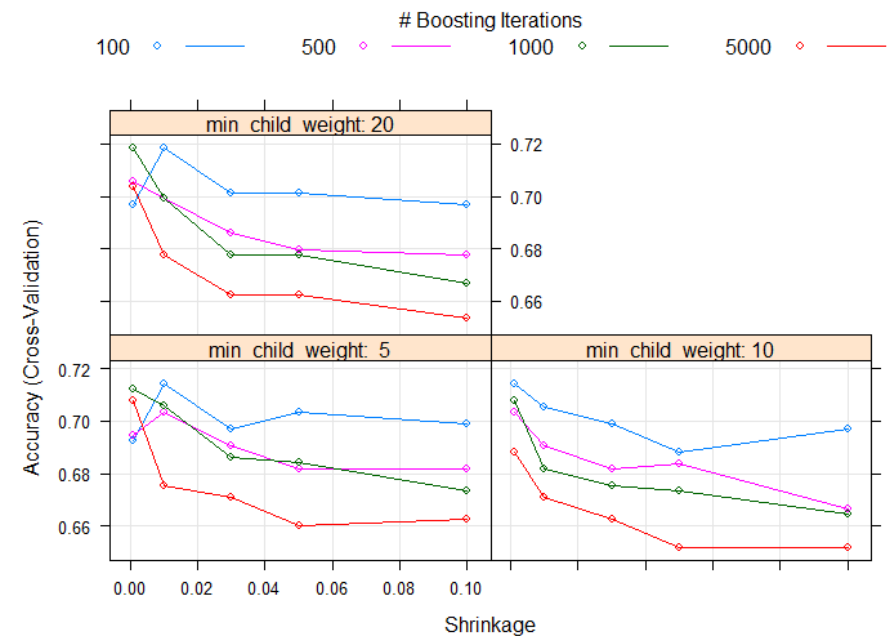
```
plot(xgbm)
```

Tuning parameter 'colsample\_bytree' was held constant at a value of 1

Tuning parameter 'subsample' was held constant at a value of 1

Accuracy was used to select the optimal model using the largest value.

The final values used for the model were nrounds = 100, max\_depth = 6, eta = 0.01, gamma = 0, colsample\_bytree = 1, min\_child\_weight = 20 and subsample = 1.



```
# ESTUDIO DE EARLY STOPPING
# Probamos a fijar algunos parámetros para ver como evoluciona
# en función de las iteraciones

xgbmgrid<-expand.grid(eta=c(0.03),
  min_child_weight=c(5),
  nrounds=c(50,100,150,200,250,300),
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

set.seed(12345)
control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)

xgbm<- train(factor(chd)~.,data=saheartbis,
  method="xgbTree",trControl=control,
  tuneGrid=xgbmgrid,verbose=FALSE)

plot(xgbm,ylim=c(0.65,0.76))

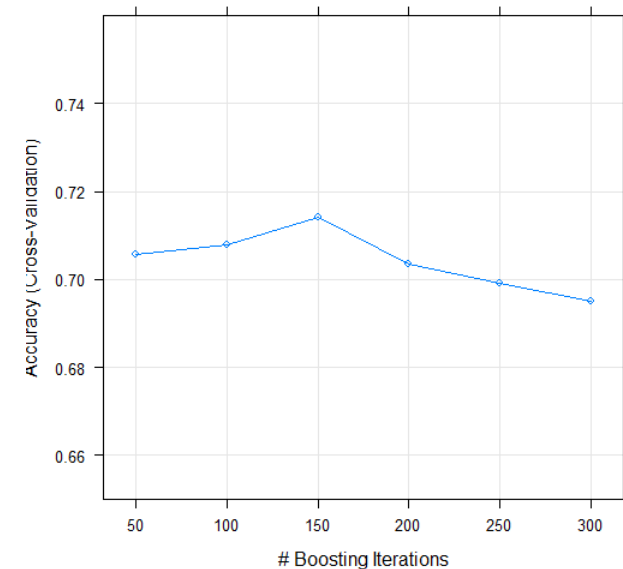
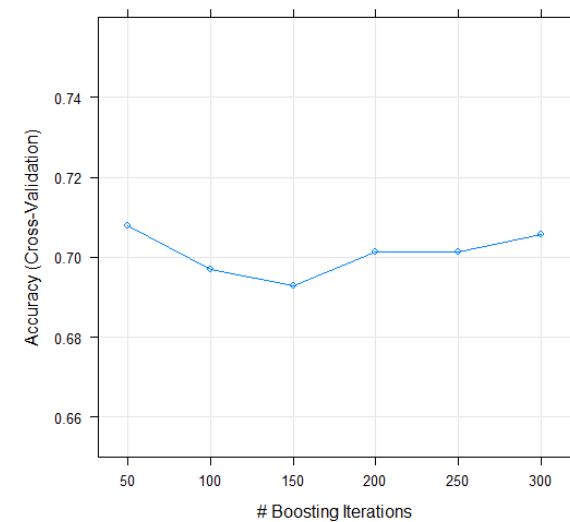
# Probamos con otras semillas para la validación cruzada

set.seed(12367)
control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)

xgbm<- train(factor(chd)~.,data=saheartbis,
  method="xgbTree",trControl=control,
  tuneGrid=xgbmgrid,verbose=FALSE)

plot(xgbm,ylim=c(0.65,0.76))
```

Parece que con 50 o 100 iteraciones es suficiente (early stopping).



## # IMPORTANCIA DE VARIABLES

```
varImp(xgbm)
plot(varImp(xgbm))
```

La variable index no tiene relación con la dependiente. Todas las que tienen importancia menor deberían estar fuera.

Volvemos a probar tuneado de parámetros con solo las variables buenas

## # PRUEBO PARÁMETROS CON VARIABLES SELECCIONADAS

```
xgbmgrid<-expand.grid(
  min_child_weight=c(5,10,20),
  eta=c(0.1,0.05,0.03,0.01,0.001),
  nrounds=c(100,500,1000,5000),
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)

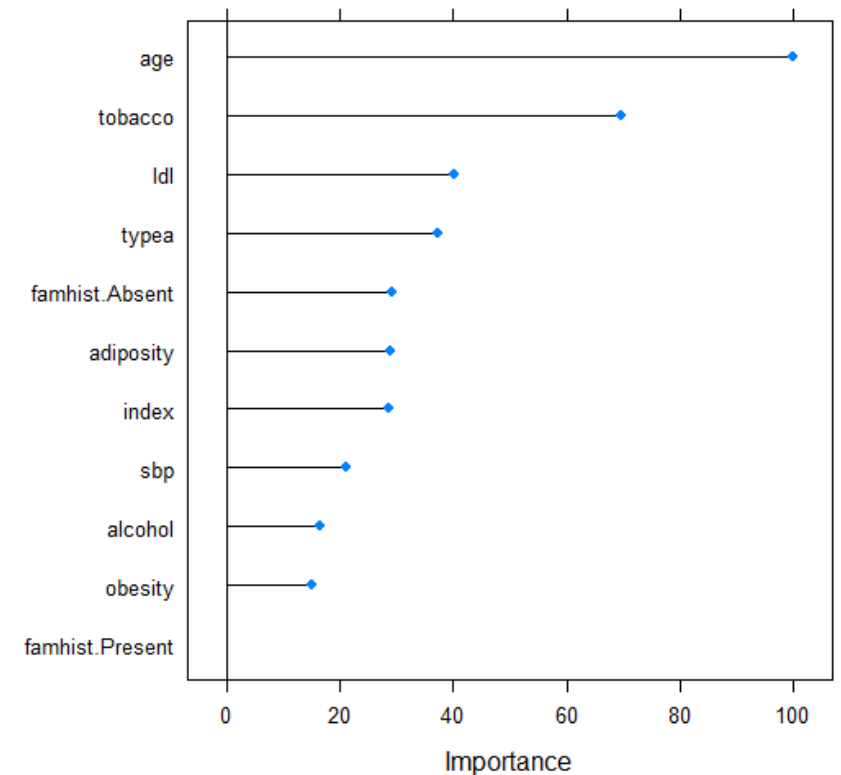
xgbm<- train(factor(chd)~age+tobacco+ldl+adiposity+typea+famhist.Absent,
  data=saheartbis,
  method="xgbTree",trControl=control,
  tuneGrid=xgbmgrid,verbose=FALSE)
```

xgbm

Accuracy was used to select the optimal model using the largest value.  
The final values used for the model were nrounds = 500, max\_depth = 6, eta = 0.001, gamma = 0, colsample\_bytree = 1, min\_child\_weight = 5 and subsample = 1.

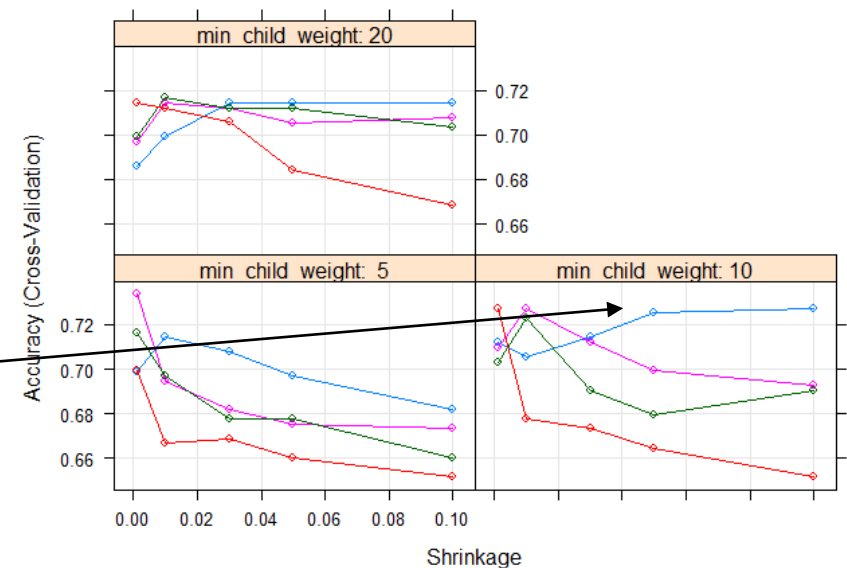
```
plot(xgbm)
```

Parece más fiable usar n=100 con shrinkage alto



# Boosting Iterations

100    500    1000    5000



```
load ("saheartbis.Rda")
...
medias6<-cruzadagbmbin(data=saheartbis, vardep="chd",
listconti=c("age", "tobacco", "ldl",
"adiposity", "typea", "famhist.Absent"),
listclass=c(""),
grupos=4,sinicio=1234, repe=5,
n.minobsinnode=10, shrinkage=0.001, n.trees=5000, interaction.depth=2)
medias6$modelo="gbm"
```

```
medias7<-cruzadaxgbmbin(data=saheartbis, vardep="chd",
listconti=c("age", "tobacco", "ldl",
"adiposity", "typea", "famhist.Absent"),
listclass=c(""),
grupos=4,sinicio=1234, repe=5,
min_child_weight=10, eta=0.08, nrounds=100, max_depth=6,
gamma=0, colsample_bytree=1, subsample=1,
alpha=0, lambda=0, lambda_bias=0)
```

```
medias7$modelo="xgbm"
```

```
# La otra opción recomendada por caret
```

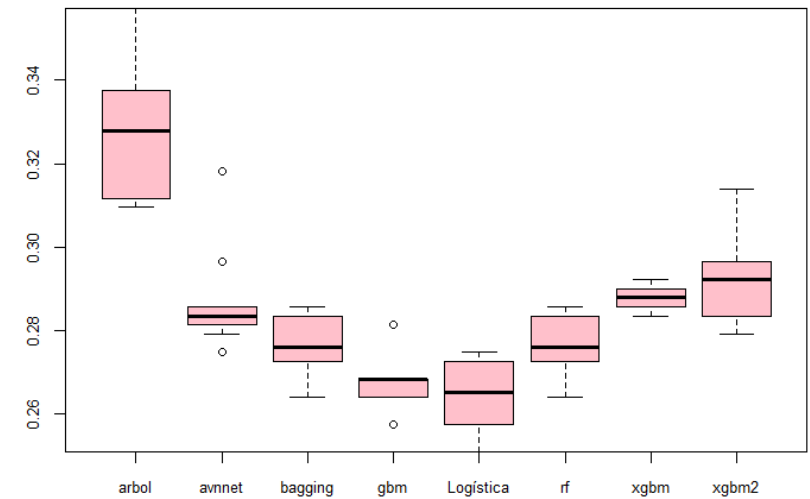
```
medias8<-cruzadaxgbmbin(data=saheartbis, vardep="chd",
listconti=c("age", "tobacco", "ldl",
"adiposity", "typea", "famhist.Absent"),
listclass=c(""),
grupos=4,sinicio=1234, repe=5,
min_child_weight=10, eta=0.001, nrounds=500, max_depth=6,
gamma=0, colsample_bytree=1, subsample=1,
alpha=0, lambda=0, lambda_bias=0)
```

```
medias8$modelo="xgbm2"
```

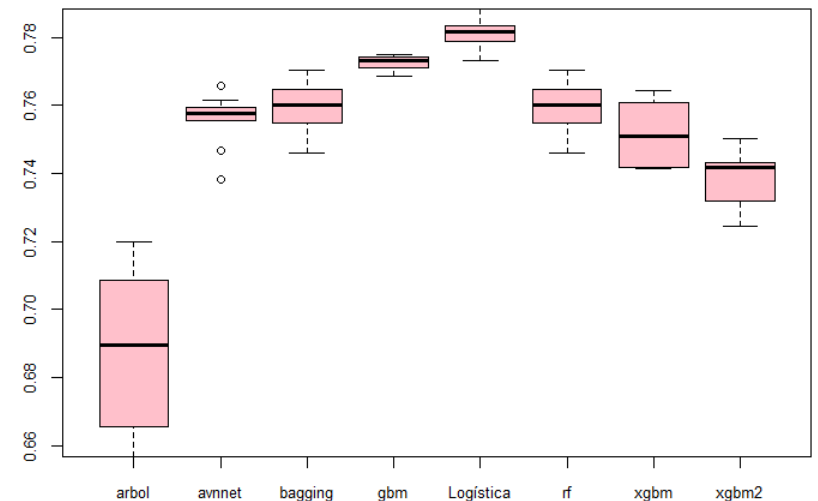
```
union1<-rbind(medias1,medias2,medias3,medias4,medias5,medias6,
medias7,medias8)
```

```
par(cex.axis=0.8, cex=1)
boxplot(data=union1, tasa~modelo, main="TASA FALLOS", col="pink")
boxplot(data=union1, auc~modelo, main="AUC", col="pink")
```

**TASA FALLOS**



**AUC**





En xgboost se pueden intentar incorporar técnicas para reducir la varianza: sortear observaciones (en plan bagging-gbm estocástico) o bien variables (como Random Forest, pero antes de cada árbol).

```
# PARA REDUCIR LA VARIANZA SE PUEDE RECURRIR A SORTEAR VARIABLES
# (ESTILO RANDOMFOREST PERO ANTES DEL ARBOL) Y/O
# A SORTEAR OBSERVACIONES (BAGGING)
# EN AMBOS CASOS HAY QUE AUMENTAR EL NÚMERO DE ÁRBOLES
medias9<-cruzadaxgbmbin(data=saheartbis, vardep="chd",
  listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  min_child_weight=10, eta=0.10, nrounds=200, max_depth=6,
  gamma=0, colsample_bytree=0.8, subsample=1,
  alpha=0, lambda=0, lambda_bias=0)
```

```
medias9$modelo="xgbmvar08"
```

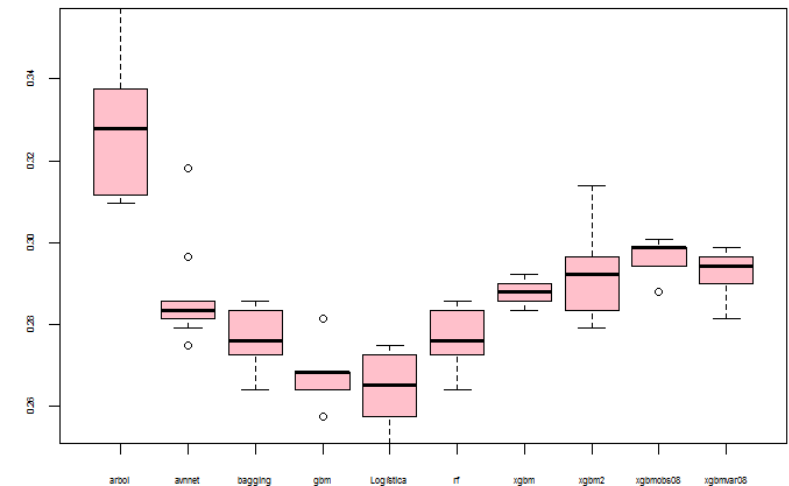
```
medias10<-cruzadaxgbmbin(data=saheartbis, vardep="chd",
  listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  min_child_weight=10, eta=0.10, nrounds=200, max_depth=6,
  gamma=0, colsample_bytree=1, subsample=0.8,
  alpha=0, lambda=0, lambda_bias=0)
```

```
medias10$modelo="xgbmobs08"
```

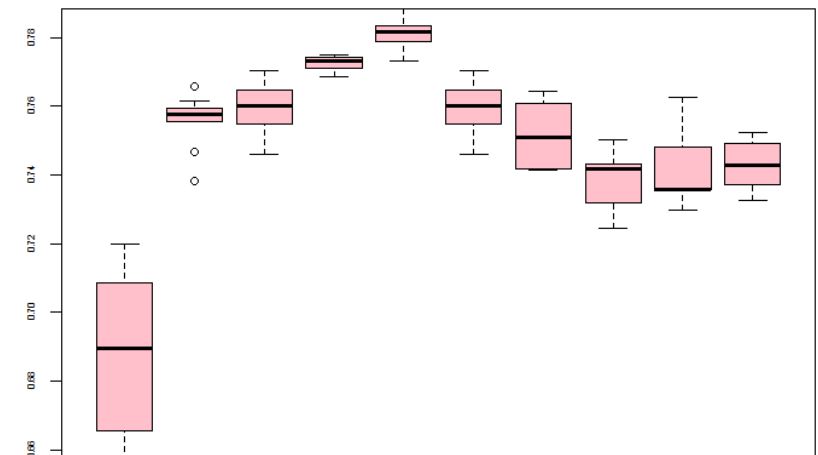
```
union1<-rbind(medias1,medias2,medias3,medias4,medias5,medias6,medias7,
  medias8,medias9,medias10)
```

```
par(cex.axis=0.5,cex=1)
boxplot(data=union1,tasa~modelo,main="TASA FALLOS",col="pink")
boxplot(data=union1,auc~modelo,main="AUC",col="pink")
```

**TASA FALLOS**



**AUC**



## Ejemplo con variable dependiente continua. El criterio es RMSE, cuanto más bajo mejor.

```
# EJEMPLO CON VARIABLE CONTINUA
```

```
load("compressbien.Rda")
```

```
library(caret)
```

```
set.seed(12345)
```

```
xgbmgrid<-expand.grid(  
  min_child_weight=c(5,10,20),  
  eta=c(0.1,0.05,0.03,0.01,0.001),  
  nrounds=c(100,500,1000,5000),  
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)
```

```
control<-trainControl(method = "cv",number=4,savePredictions = "all")
```

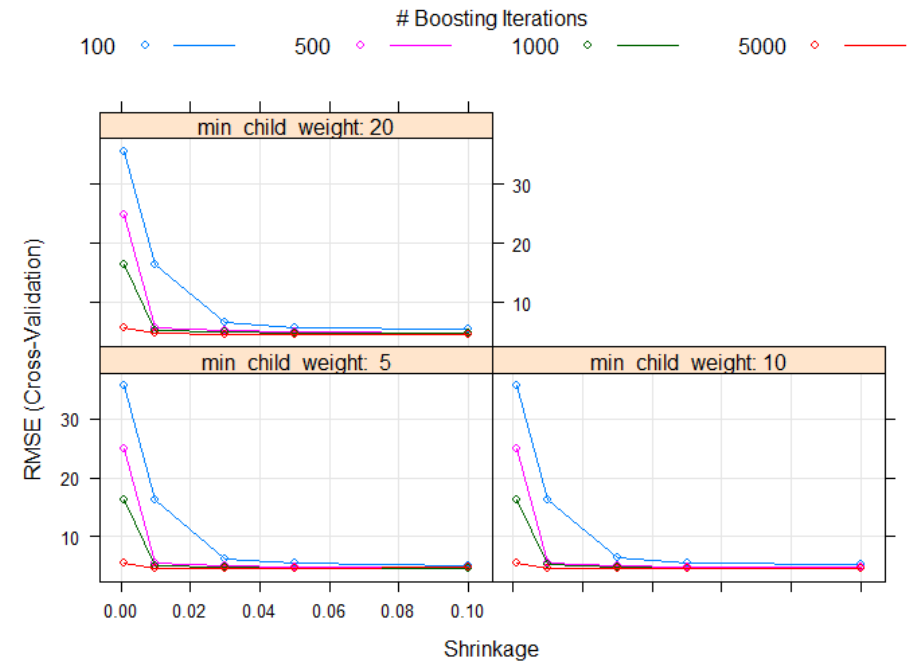
```
comp<-compressbien[,c("cstrength","age","water","cement","blast")]
```

```
xgbm<- train(cstrength~.,data=comp,  
  method="xgbTree",trControl=control,  
  tuneGrid=xgbmgrid,verbose=FALSE)
```

```
xgbm
```

```
# Tuning parameter 'subsample' was held constant at a value of 1  
# RMSE was used to select the optimal model using the smallest value.  
# The final values used for the model were nrounds = 5000, max_depth = 6,  
# eta = 0.03, gamma = 0, colsample_bytree = 1, min_child_weight = 10  
# and subsample = 1.
```

```
plot(xgbm)
```



## Early stopping

```
# ESTUDIO DE EARLY STOPPING
# Probamos a fijar algunos parámetros para ver como evoluciona
# en función de las iteraciones

xgbmgrid<-expand.grid(eta=c(0.03,0.05),
  min_child_weight=c(10),
  nrounds=c(1000,2000,5000),
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

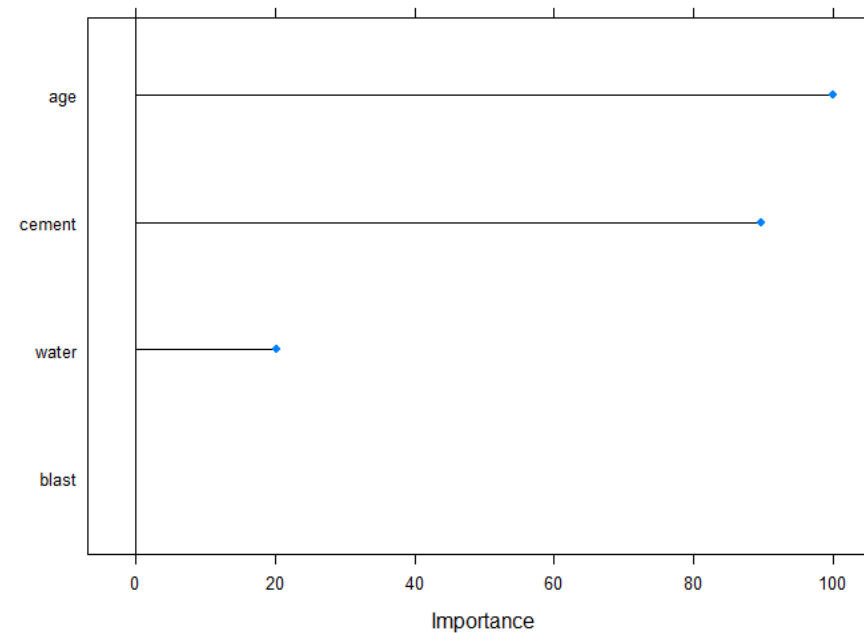
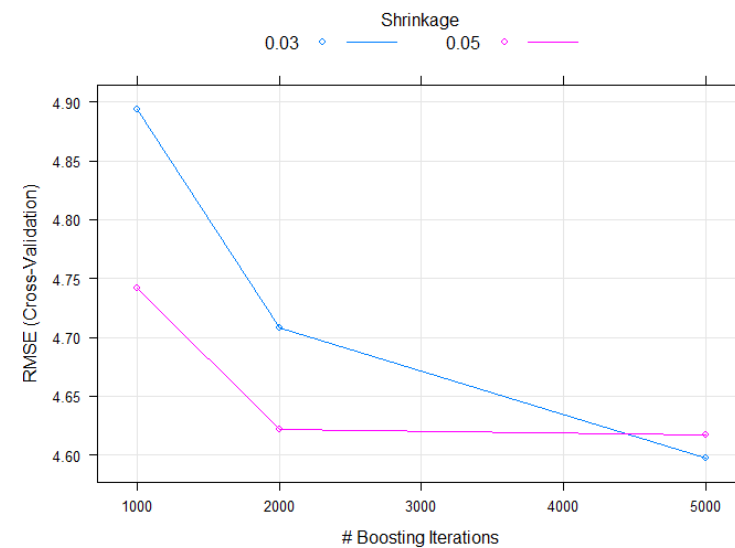
set.seed(12345)
control<-trainControl(method = "cv",number=4,savePredictions = "all")

xgbm<- train(cstrength~.,data=comp,
  method="xgbTree",trControl=control,
  tuneGrid=xgbmgrid,verbose=FALSE)

plot(xgbm)

# IMPORTANCIA DE VARIABLES

varImp(xgbm)
plot(varImp(xgbm))
```



## Pruebas con Validación cruzada repetida

```
source("cruzada arbol continua.R")
source("cruzadas avnnet y lin.R")
source("cruzada rf continua.R")
source("cruzada gbm continua.R")
source("cruzada xgboost continua.R")
```

```
medias7<-cruzadaxgbm(data=data,
  vardep="cstrength",listconti=c("age","water","cement","blast"),
  listclass=c(""),
  grupos=4,sinicio=1234, repe=5,
  min_child_weight=10,eta=0.03,nrounds=5000,max_depth=6,
  gamma=0,colsample_bytree=1,subsample=1)
```

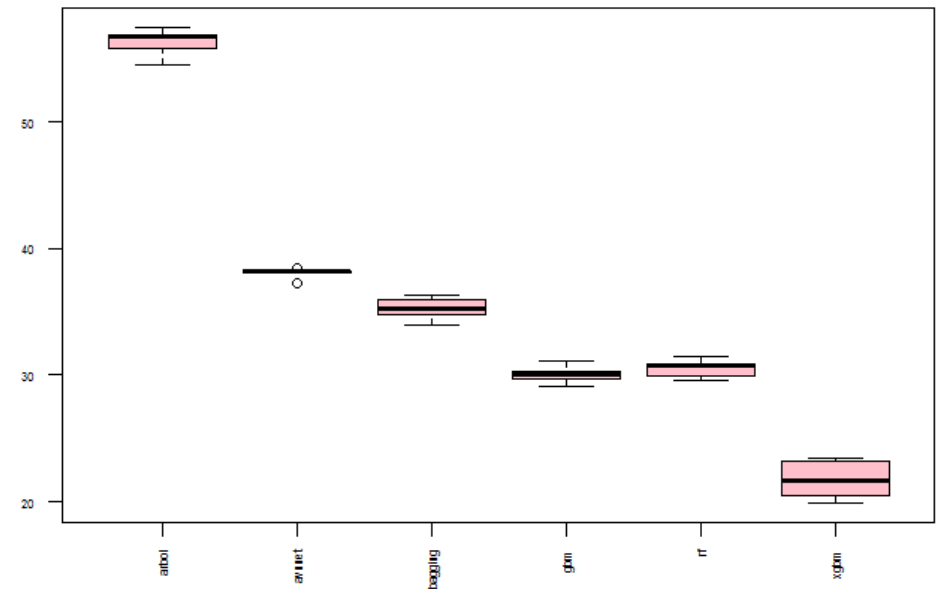
```
medias7$modelo="xgbm"
```

```
union1<-rbind(medias1,medias2,medias3,medias4,
  medias5,medias6,medias7)
```

```
par(cex.axis=0.5)
boxplot(data=union1,error~modelo)
```

```
union1<-rbind(medias1,medias3,medias4,medias5,medias6,medias7)
```

```
par(cex.axis=0.5)
boxplot(data=union1,error~modelo,col="pink")
```



```
# PRUEBAS SORTEANDO OBSERVACIONES Y TAMBIÉN VARIANDO EL PARÁMETRO ALPHA DE REGULARIZACIÓN
```

```
medias8<-cruzadaxgbm(data=data,  
  vardep="cstrengh",listconti=c("age","water","cement","blast"),  
listclass=c(""),  
  grupos=4,sinicio=1234, repe=5,  
  min_child_weight=10,eta=0.03,nrounds=5000,max_depth=6,  
  gamma=0,colsample_bytree=1,subsample=0.8)
```

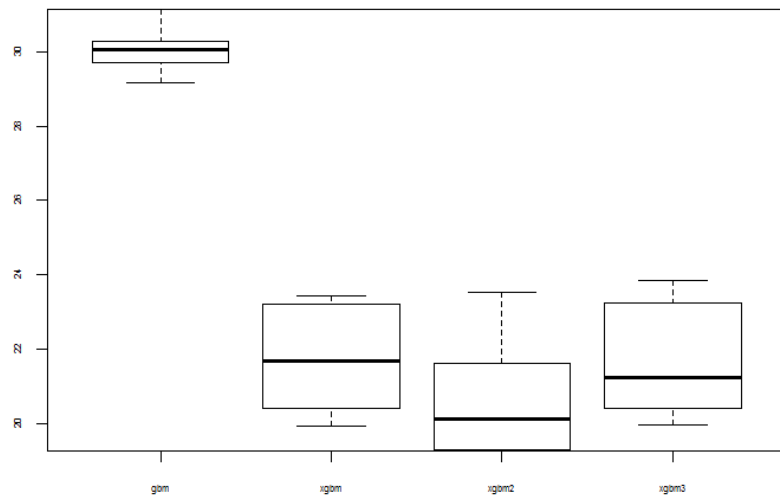
```
medias8$modelo="xgbm2"
```

```
medias9<-cruzadaxgbm(data=data,  
  vardep="cstrengh",listconti=c("age","water","cement","blast"),  
listclass=c(""),  
  grupos=4,sinicio=1234, repe=5,  
  min_child_weight=10,eta=0.03,nrounds=5000,max_depth=6,  
  gamma=0,colsample_bytree=1,subsample=1,alpha=0.5)
```

```
medias9$modelo="xgbm3"
```











```
union1<-rbind(medias6,medias7,medias8,medias9)
```

```
par(cex.axis=0.5)  
boxplot(data=union1,error~modelo)
```



# Kaggle y Gradient Boosting:


























































18 Active Competitions		
	<b>2018 Data Science Bowl</b> Find the nuclei in divergent images to advance medical discovery <i>Featured</i> · 11 days to go ·  biology	\$100,000 3,425 teams
	<b>TalkingData AdTracking Fraud Detection Challenge</b> Can you detect fraudulent click traffic for mobile app ads? <i>Featured</i> · a month to go ·	\$25,000 2,276 teams
	<b>CVPR 2018 WAD Video Segmentation Challenge</b> Can you segment each objects within image frames captured by vehicles? <i>Research</i> · 2 months to go ·	\$2,500 7 teams
	<b>iMaterialist Challenge (Fashion) at FGVC5</b> Image classification of fashion products. <i>Research</i> · 2 months to go ·	\$2,500 8 teams
	<b>iMaterialist Challenge (Furniture) at FGVC5</b> Image Classification of Furniture & Home Goods. <i>Research</i> · 2 months to go ·	\$2,500 155 teams
	<b>Google Landmark Retrieval Challenge</b> Given an image, can you find all of the same landmarks in a dataset? <i>Research</i> · 2 months to go ·  image data	\$2,500 108 teams
	<b>Google Landmark Recognition Challenge</b> Label famous (and not-so-famous) landmarks in images <i>Research</i> · 2 months to go ·  image data	\$2,500 206 teams

# Kaggle nació en 2010

En octubre 2019:

- 18 competiciones activas
- 356 competiciones finalizadas

Competitions		Kernels	Discussion	Learn more about rankings ›	
 108 Grandmasters	 985 Masters	 3,213 Experts	 47,370 Contributors	 29,401 Novices	
Rank	Tier	User		Medals	Points
1		 <b>bestfitting</b>	joined 2 years ago	 9  3  0	171,845
2		 <b>Giba</b>	joined 6 years ago	 38  28  21	167,962
3		 Μανος Μιχαηλιδης <b>KazAnova</b>	joined 5 years ago	 28  28  24	149,053
4		 <b>raddar</b>	joined 3 years ago	 10  7  3	145,773
5		 <b>Eureka</b>	joined 4 years ago	 18  17  5	133,257
6		 <b>weiwei</b>	joined 2 years ago	 7  6  4	123,987
7		 <b>Stanislav Semenov</b>	joined 4 years ago	 28  9  0	119,929
8		 <b>idle_speculation</b>	joined 5 years ago	 8  8  7	115,058
9		 <b>ZFTurbo</b>	joined 2 years ago	 9  14  7	114,815
10		 <b>Little Boat</b>	joined 4 years ago	 13  16  6	108,851

# Ejemplos de Evolución en los algoritmos utilizados:

2011:



## Predict Grant Applications

This task requires participants to predict the outcome of grant applications for the University of Melbour...

Featured · 7 years ago ·

\$5,000

204 teams

El ganador usó Random Forest

2012:



## Give Me Some Credit

Improve on the state of the art in credit scoring by predicting the probability that somebody will experien...

Featured · 6 years ago ·

\$5,000

924 teams

El ganador usó redes neuronales  
El segundo usó gbm

2013:



## U.S. Census Return Rate Challenge

Predict census mail return rates.

Featured · 5 years ago ·

\$25,000

243 teams

(V. dependiente continua)  
Los primeros usaron todos gbm



2014:



### Loan Default Prediction - Imperial College London

Constructing an optimal portfolio of loans  
\$10,000 · 675 teams · 4 years ago

Todos los primeros usaron gbm

2015:



### Restaurant Revenue Prediction

Predict annual restaurant sales based on objective measurements  
Featured · 3 years ago · tabular data, regression

\$30,000  
2,257 teams

(V. dependiente continua)

El ganador usó gbm (y posiblemente los primeros restantes también)

2016:



### Santander Customer Satisfaction

Which customers are happy customers?  
\$60,000 · 5,123 teams · 2 years ago

El ganador usó xgboost (y posiblemente los primeros restantes también)

2017:



**BOSCH**

**Bosch Production Line Performance**

Reduce manufacturing failures

\$30,000 · 1,373 teams · a year ago

El ganador usó xgboost

Algunos de los primeros xgboost combinado con Random Forest

# Explicaciones de solución del ganador del concurso:



## Restaurant Revenue Prediction

Predict annual restaurant sales based on objective measurements

Featured · 3 years ago · 📊 tabular data, regression

\$30,000

2,257 teams

Thanks a lot!

Here are the important steps of my solution:

### 1) Feature Engineering

i) Square root transformation was applied to the obfuscated P variables with maximum value  $\geq 10$ , to make them into the same scale, as well as the target variable “revenue”.

ii) Random assignments of uncommon city levels to the common city levels in both training and test set, which I believe, diversified the geo location information contained in the city variable and in some of the obfuscated P variables.

Note: I discovered this to be helpful **by chance**. My intention was to assign uncommon city levels to their nearest common city levels. But it read the city levels differently on my laptop and on the server. It performed significantly better on the server. I am not 100% sure, but my explanation were given above.

iii) Missing value indicator for multiple P variables, i.e. P14 to P18, P24 to P27, and P30 to P37 was created to help differentiate synthetic and real test data.

Note: These variables were all zeroes on 88 out of 137 rows in the training set. The proportion was much less on the test set, i.e. those rows on which these variables were not zeroes at the same time has higher probability to be synthetic.

iv) Type “MB”, which did not occur in training set, was changed to Type “DT” in test set.

v) Time / Age related information was also extracted, including open day, week, month and lasting years and days.

vi) Zeroes were treated as missing values and mice imputation was applied on training and test set separately.

## 2) Modelling and Selection Criteria

i ) Gradient boosting models were trained on the feature-engineered training set. I used R caret package and 10-fold cv repeated 10 times (default setting) to train the gbm models. The parameters grid used was simple to reduce over-fitting:

```
gbmGrid <- expand.grid(interaction.depth = c(6, 7, 8, 9),  
n.trees = (3:7) * 10,  
shrinkage = 0.05)
```

ii ) **Two statistics were used to determine the model(s) to choose: training error and training error with outliers removed. The error limits are  $3.7 * 10^{12}$  and  $1.4 * 10^{12}$ , respectively.**

Note: I tested this strategy **post-deadline**, it was very effective choosing the "right" models. Around one in 15-20 models trained in step i) satisfied the two constraints. i.e. I trained about 200 models (using different seed) and 11 of them had training error and training error with outliers removed both lower than the limit I set. I randomly averaged 4 of them to make it more robust as a final model. These final models scored ~1718 to ~1735 privately and ~1675 to ~1707 publicly. (I guess taking average was more effective on the public data.)

## 3) Possible Improvements

**I read from the forum that dealing with outliers properly could improve scores, although I did not try it out myself. And in this situation, my strategy in 2) ii) might need modification.**

## 4) Conclusion

**I got lucky on this competition, while my true intention was to stabilize my performance on top 5%. As you guys can see, I am trying hard to stay in top 5% on two other competitions near its end. The techniques or methods I used here might not be a good strategy for other problems/competitions, and vice versa. I am learning a lot from the Kaggle forum and by participating in Kaggle competitions and gathering experience throughout these practices. Cheers!**

PS: I am also from a mathematical background, and I personally am very impressed by team BAYZ's work. It seems to me that hundreds of submissions would do the job, but 110, it's just amazing and it must have involved really smart strategies.

PS2: I wrote about my story of this competition and submitted to Kaggle's winners blog. This post has already covered all the details in my solution, so reading that blog will help you know a little bit about me. I will post the link here once it's ready. You guys are welcome to take a look.