

# Bagging, Random Forest, Gradient Boosting

<b>Bagging .....</b>	<b>3</b>
Principales parámetros a controlar en bagging .....	5
Bagging con caret .....	5
Parámetros básicos en bagging con caret .....	5
El Problema de la Reproducibilidad .....	6
Out of Bag .....	8
Pruebas con validación cruzada repetida .....	8
Ejemplo con variable continua dependiente .....	12
<b>Random Forest .....</b>	<b>14</b>
Principales parámetros a controlar en Random Forest .....	14
Random Forest con caret .....	15
Parámetros básicos en Random Forest con caret .....	15
Validación cruzada repetida .....	18
Ejemplo Random Forest variable continua dependiente .....	20
Validación cruzada repetida .....	21
<b>Gradient Boosting .....</b>	<b>23</b>
Parámetros a determinar en el algoritmo Gradient Boosting .....	25
Algoritmo Gradient Boosting para clasificación .....	28
Ventajas del Gradient Boosting .....	29
Desventajas del Gradient Boosting .....	29
Gradient Boosting con caret .....	30
Parámetros básicos .....	30
Ejemplo variable dependiente binaria .....	30
Importancia de variables .....	32
Ejemplo variable dependiente continua .....	34
La última moda: Xgboost .....	36
Regularización .....	36
Regularización en XGboost .....	37
Ventajas del Xgboost .....	37
Desventajas del Xgboost .....	38
Xgboost con caret .....	39
Ejemplos con validación cruzada repetida .....	42
Ejemplo con variable dependiente continua .....	45

<b>Kaggle y Gradient Boosting .....</b>	<b>49</b>
Ejemplos de Evolución en los algoritmos utilizados .....	50
Algunos comentarios sobre Kaggle y Machine Learning .....	51
<b>Bibliografía básica .....</b>	<b>53</b>

## Bagging

Recordemos las 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: toscos en los valores de predicción.

Afortunadamente, se ha conseguido mantener las ventajas de los árboles mejorando el poder predictivo mediante la combinación de muchos árboles, lo que será la base de las técnicas Bagging, Random Forest y Gradient Boosting que se verán en este capítulo.

El primer intento importante de aprovechar las ventajas de los árboles pero evitar su inestabilidad en la predicción es la técnica Bagging (Bootstrap Averaging) (Breiman, 1996)

### Algoritmo Bagging

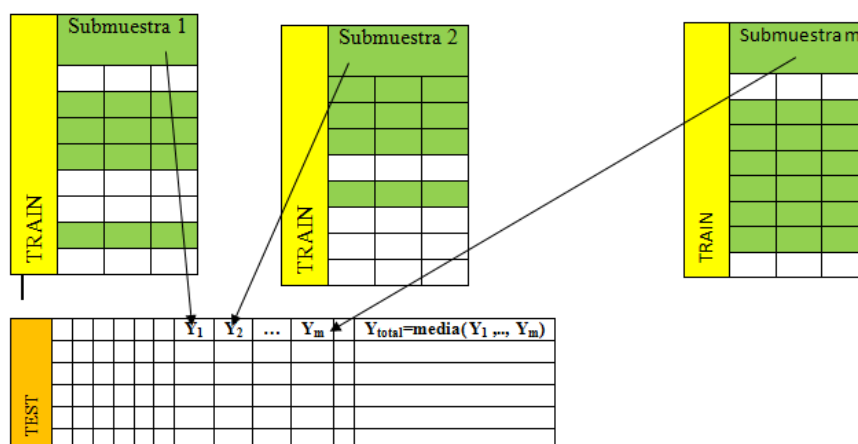
Dados los datos de tamaño  $N$ ,

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. En palabras, en lugar de predecir con un único árbol, se predice con muchos árboles, cada uno de ellos construido a partir de una submuestra, y se promedia la predicción de todos los árboles.

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.

Recordando el algoritmo:

**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. Este suele ser el mejor método y más utilizado.
  - 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

Para obtener óptimos resultados, es necesario tunear los parámetros del bagging. Los parámetros más importantes son:

- El tamaño de las muestras  $m$  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 ya 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 muchos paquetes), p-valor, etc.

## Bagging con caret

bagging1 v 5.0.R

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

### Parámetros básicos en bagging con caret

- 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. Algunas consideraciones:
  - a) Si se deja vacío (no se usa el parámetro), se toma como sampsize el total de observaciones, y hay que poner replace=TRUE para realizar muestras con reemplazamiento. Este es el proceso clásico.
  - b) Si se utiliza algún método de control (cualquier cosa diferente de method="none") hay que considerar que el sampsize máximo debería ser igual o inferior al tamaño de la muestra. Por ejemplo si hacemos train-test (method="LGOCV") con  $p=0.8$  en unos datos de 1000 observaciones, el máximo que podemos poner en sampsize es  $0.8 \cdot 1000 = 800$ . En validación cruzada igual, el máximo de datos en sampsize sería  $(k-1) \cdot n$  donde  $k$  es el número de grupos.
- replace=TRUE (con reemplazamiento o FALSE sin reemplazamiento). Si se usa sin reemplazamiento, hay que tener muy en cuenta el apartado b) anterior.

## El Problema de la Reproducibilidad

Antes de proceder a usar bagging con caret, es conveniente comentar algunos aspectos que tienen que ver con la aleatoriedad y dependencia del azar de nuestras pruebas, tuneos y evaluaciones de modelos.

Cuando se plantean modelos en machine learning se usan a menudo técnicas de remuestreo para evaluar bien los modelos (training-test, validación cruzada, etc.). Estas técnicas conllevan sorteos, ordenaciones aleatorias, etc. Para ello se utilizan semillas (seed) de aleatorización.

Si el proceso depende de una sola semilla que se suele poner al principio del código, no hay problema con la reproducibilidad del método; otro usuario con el mismo código podría tener exactamente los mismos resultados ejecutando el programa otro día en otro ordenador.

A menudo los paquetes de software libre y también los comerciales, no ofrecen un buen control de semillas y nos podemos encontrar con que el mismo código, aún con una semilla inicial declarada, da resultados diferentes en ordenadores distintos, o en ejecuciones distintas. Si estamos en un proceso de tuneo, el parámetro óptimo en unos casos da por ejemplo 5 nodos en una red, y en otra ejecución con la misma semilla da menor error con 15 nodos.

Por ello a menudo hay que cerciorarse de la reproducibilidad de nuestras pruebas, ejecutando varias veces el código y de ser posible en ordenadores distintos o tras reiniciar el ordenador.

Estos problemas son más acuciantes si se usan métodos como bagging-random forest, cuya base es la aleatorización y el remuestreo.

También hay que recordar que la semilla de aleatorización no es un parámetro del modelo, la usamos como control y podemos jugar con ella, variándola como hacemos en validación cruzada repetida, para observar la sensibilidad del modelo y sus errores ante un esquema de selección de observaciones ligeramente diferente.

En caret existe un esquema de control de semillas para que sean reproducibles nuestras pruebas.

### ¿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).

Después de controlar las semillas pasamos a usar caret para evaluar la precisión del modelo bajo validación cruzada:

### Control de la semilla para reproducibilidad en caret

En caret bastaría con **ejecutar `set.seed(semilla)`** antes de cada ejecución de control y train para obtener el mismo resultado reproducible.

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

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
0.6862256 0.2732721
```

## Out of Bag

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

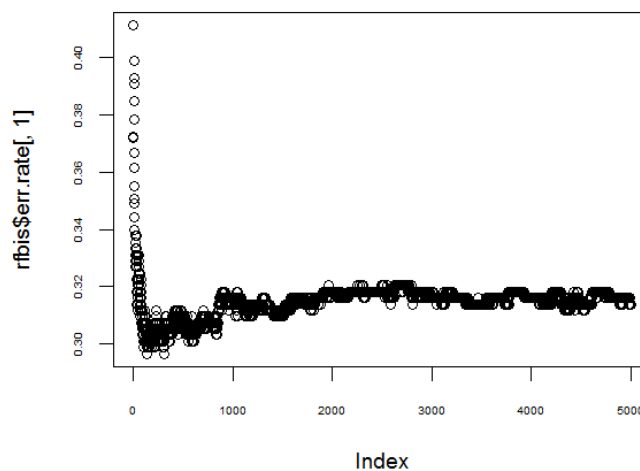
Caret no permite observar esto, pero sí el paquete randomForest

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

library(randomForest)

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

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



Se observa como el error se estabiliza a partir de menos de 1000 submuestras.

## Pruebas con validación cruzada repetida

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

```
load ("saheartbis.Rda")
source ("cruzadas avnnnet 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<-cruzadaavnnnetbin(data=saheartbis,
  vardep="chd",listconti=c("sbp", "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)

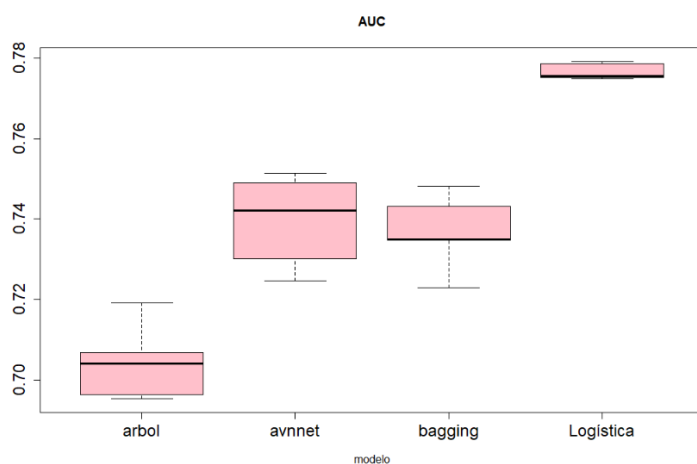
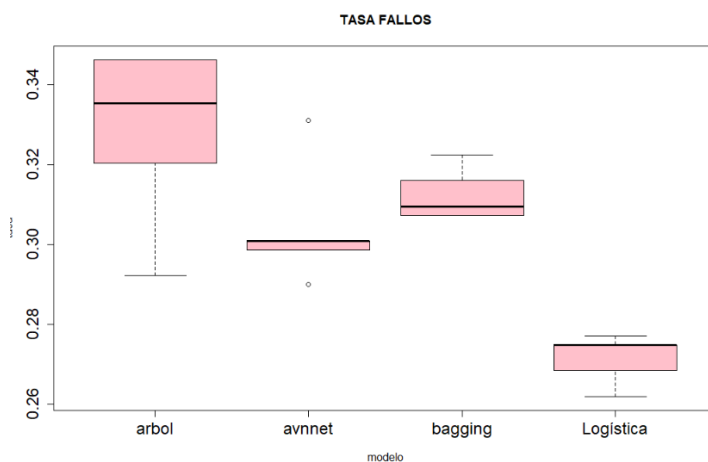
medias3$modelo="arbol"

medias4<-cruzadarfbn(data=saheartbis, vardep="chd",
listconti=c("age", "tobacco",
"ldl", "adiposity", "typea", "famhist.Absent"),
listclass=c(""),
grupos=4,sinicio=1234, repe=5,nodesize=10,
mtry=6,ntree=1000,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")

```



Podemos manipular el tamaño muestral para observar su efecto sobre el modelo bagging, incorporando el parámetro `sampsiz` en la función `cruzadarfbn`. Como en `caret` no se permite tuneado de `sampsiz`, en este caso lo hacemos con `cv` repetida y gráficos.

`Sampsiz` debe ser menor que el número de observaciones training, por lo cual si se usa por ejemplo validación cruzada de 4 grupos `sampsiz` 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 `sampsiz` que prueba por defecto el paquete `caret`.

```
medias1<-cruzadarfbn(data=saheartbis, vardep="chd",
  listconti=c("age", "tobacco",
    "ldl", "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=10,sinicio=1234, repe=20,nodesize=10,
  mtry=6,ntree=3000,replace=TRUE,sampsiz=50)

  medias1$modelo="bagging50"

  medias2<-cruzadarfbn(data=saheartbis, vardep="chd",
    listconti=c("age", "tobacco",
      "ldl", "adiposity", "typea", "famhist.Absent"),
    listclass=c(""),
    grupos=10,sinicio=1234, repe=20,nodesize=10,
    mtry=6,ntree=3000,replace=TRUE,sampsiz=100)

    medias2$modelo="bagging100"

    medias3<-cruzadarfbn(data=saheartbis, vardep="chd",
      listconti=c("age", "tobacco",
        "ldl", "adiposity", "typea", "famhist.Absent"),
      listclass=c(""),
      grupos=10,sinicio=1234, repe=20,nodesize=10,
      mtry=6,ntree=3000,replace=TRUE,sampsiz=150)

      medias3$modelo="bagging150"

      medias4<-cruzadarfbn(data=saheartbis, vardep="chd",
        listconti=c("age", "tobacco",
          "ldl", "adiposity", "typea", "famhist.Absent"),
        listclass=c(""),
        grupos=10,sinicio=1234, repe=20,nodesize=10,
        mtry=6,ntree=3000,replace=TRUE,sampsiz=200)

        medias4$modelo="bagging200"

        medias5<-cruzadarfbn(data=saheartbis, vardep="chd",
          listconti=c("age", "tobacco",
            "ldl", "adiposity", "typea", "famhist.Absent"),
          listclass=c(""),
          grupos=10,sinicio=1234, repe=20,nodesize=10,
          mtry=6,ntree=3000,replace=TRUE,sampsiz=300)
```

```

medias5$modelo="bagging300"

medias6<-cruzadarfbin(data=saheartbis, vardep="chd",
  listconti=c("age", "tobacco",
    "ldl", "adiposity", "typea", "famhist.Absent"),
  listclass=c(""),
  grupos=10,sinicio=1234,repe=20,nodesize=10,
  mtry=6,ntree=3000,replace=TRUE)

# En este modelo al no usar sampsize usa el que tiene por defecto
# caret, que son todas las observaciones con reemplazamiento (415)

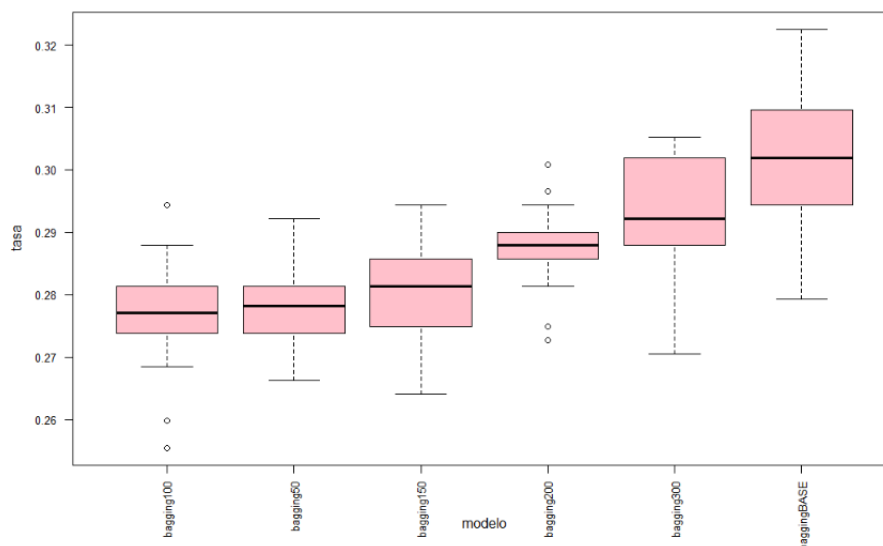
medias6$modelo="baggingBASE"

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

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

uni<-union1
uni$modelo <- with(uni,
  reorder(modelo,tasa, mean))
par(cex.axis=0.8,las=2)
boxplot(data=uni,tasa~modelo,col="pink")

```



Parece que 50 o 100 es un buen tamaño. Lo aplicamos comparando con los otros modelos con el mismo esquema de 10 grupos (no se pone el código aquí). Se ve que la logística sigue siendo el mejor modelo en este caso.

## 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

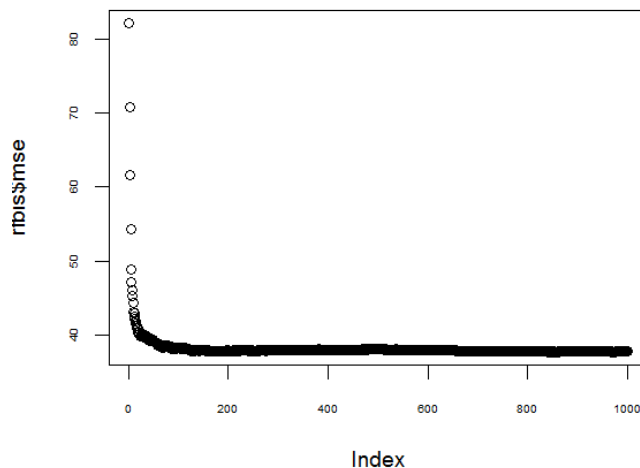
load("compressbien.Rda")

library(randomForest)

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

# En continuas este gráfico puede valer para decidir iteraciones

plot(rfbis$mse)
```



## Validación cruzada repetida

```
data<-compressbien

medias1<-cruzadaavnnnet(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="avnnnet"

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)
```

```

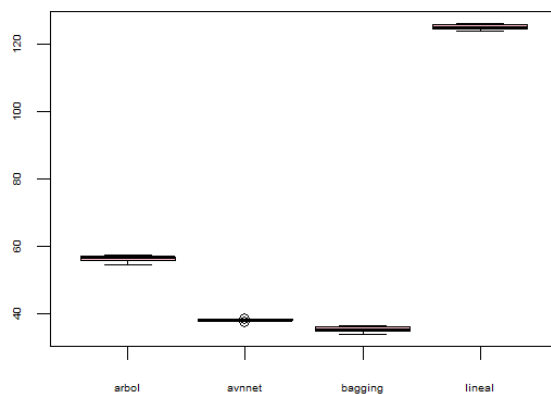
medias3$modelo="arbol"

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,col="pink")

```

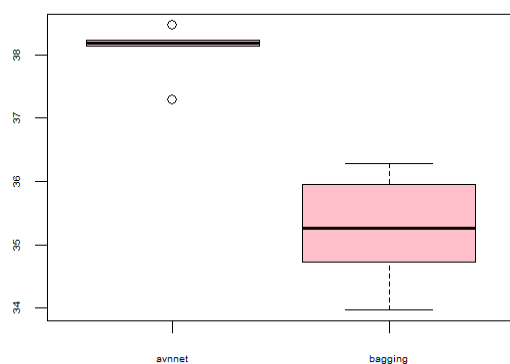


Bagging parece funcionar muy bien en este ejemplo. Para ver mejor la diferencia entre avnnet y bagging unimos solo los archivos de medias de errores de esos dos modelos.

```

union1<-rbind(medias1,medias4)
boxplot(data=union1,error~modelo,col="pink")

```



**Ejercicio:** tunear con validación cruzada repetida el tamaño de muestra sampsize

## Random Forest

Presentado por (Breiman, 2001), es una modificación del bagging que consiste en incorporar aleatoriedad en las variables utilizadas para segmentar cada nodo del árbol.

### Algoritmo Random Forest

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 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.

Obtener predicciones para todas las observaciones originales  $N$

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

### Comentarios

- 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).
- Como es un método basado en árboles, se puede elaborar una medida de importancia de variables como suma o promedio de la importancia de cada variable en los árboles creados.

## Principales parámetros a controlar en Random Forest

- El tamaño o porcentaje 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 total de variables input  $k$  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:
  - a) El número de hojas final  $o$ , en su defecto, la profundidad del árbol
  - b) El maxbranch (número de divisiones máxima en cada nodo. Por defecto se dejará en 2, árboles binarios)

- c) El p-valor para las divisiones en cada nodo. Más alto → árboles menos complejos (más sesgo, menos varianza)
- d) 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

### Random forest.R

#### Parámetros básicos en Random Forest con caret

- 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 también 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=300,nodesize=10,replace=TRUE,
  importance=TRUE)
rf
```

mtry	Accuracy	Kappa
3	0.6751874	0.2581343
4	0.6666042	0.2415272
5	0.6904048	0.3016113
6	0.6838831	0.2848640
7	0.6687031	0.2544585
8	0.6708958	0.2567905
9	0.6751874	0.2684305
10	0.6708396	0.2606668
11	0.6578336	0.2273372

Parece que mtry=5 variables está bien. Esto significa que en cada árbol creado sortearíamos en cada nodo 5 variables de las 11 variables input candidatas que tiene el archivo, y de esas 5 elegiríamos la mejor.

Con el siguiente código vemos como obtener la importancia de variables. Esto nos permite eliminar, si es razonable, aquellas que son menos importantes para obtener un modelo más

estable. El problema es que nunca sabremos donde cortar (por ej. si eliminamos la última, las dos últimas, ninguna, etc.) y tenemos que combinar esa información con otros métodos.



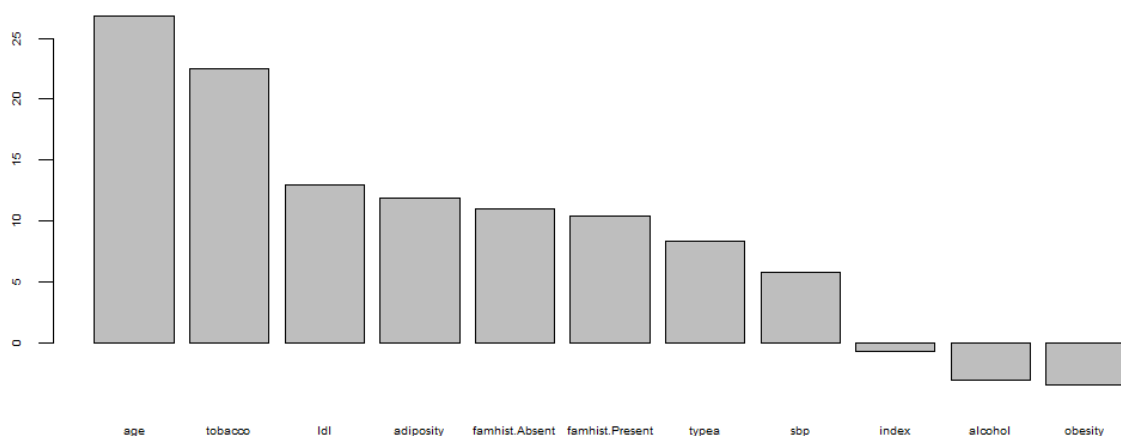
```
# IMPORTANCIA DE VARIABLES

final<-rf$finalModel

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

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

En este caso existía en nuestros datos la variable Index que es no predictiva pues es el orden de observación. En principio cualquier variable por debajo de ella (alcohol, typea) no sería útil.



En realidad el tuneado anterior lo habíamos realizado sobre todas las variables, sobre todo para observar la importancia de variables sobre todo el conjunto, pero es conveniente volver a realizar el tuneado con las variables buenas, pues no interesa en el modelo utilizar variables inservibles.

```
# EL TUNEADO ANTERIOR LO HABÍAMOS REALIZADO CON TODAS LAS VARIABLES
# PERO SABEMOS QUE SOLO 7 SON IMPORTANTES, VAMOS A REALIZAR EL TUNEADO
# UNA SEGUNDA VEZ CON SOLO LAS VARIABLES DE INTERÉS
```

```
listconti<-c("age", "tobacco", "ldl",
             "adiposity", "typea", "famhist.Absent")
paste(listconti, collapse="+")

set.seed(12345)
rfgrid<-expand.grid(mtry=c(3,4,5,6))

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

rf<-
train(factor(chd)~age+tobacco+ldl+adiposity+typea+famhist.Absent, data=
saheartbis,
      method="rf", trControl=control, tuneGrid=rfgrid,
      linout = FALSE, ntree=300, nodesize=10, replace=TRUE,
      importance=TRUE)

rf
```

```
# Recomienda mtry=3
```

Hacemos algunas comparaciones básicas con validación cruzada repetida, usando las 7 mejores variables.

### Validación cruzada repetida

```
# La función cruzadarfbin 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<-cruzadarfbin(data=saheartbis,
  vardep="chd",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<-cruzadarfbin(data=saheartbis,
  vardep="chd",listconti=c("age", "tobacco", "ldl",
    "adiposity", "typea","famhist.Absent"),
  listclass=c(""),
  grupos=10,sinicio=1234, repe=10,nodesize=10,
  mtry=3,ntree=3000,replace=TRUE,sampsize=150)

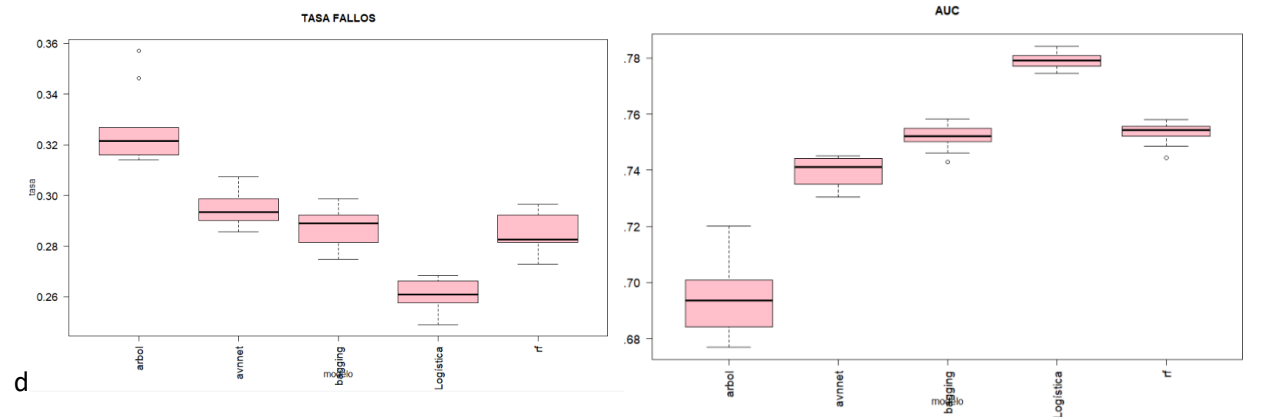
medias5$modelo="rf"

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", col="pink")
```

En este caso sigue siendo la logística mejor, y random forest no mejora con cuatro variables a meter todas las 7 (bagging).



## Ejemplo Random Forest variable continua dependiente

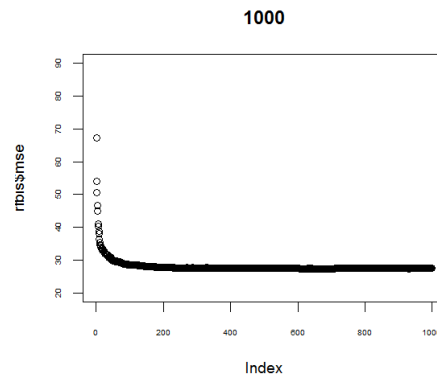
```
# EJEMPLO CON VARIABLE CONTINUA
```

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

```
library(randomForest)
```

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

```
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)
```

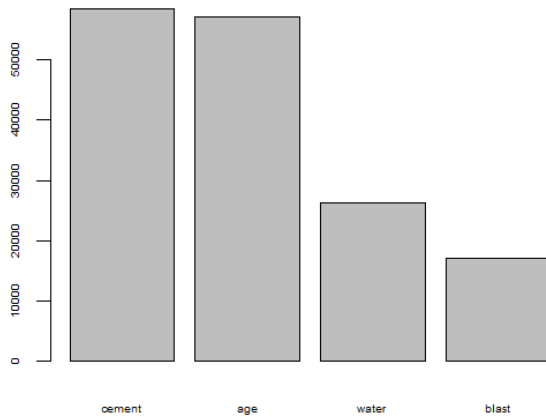
```
rf<- train(cstrength~age+water+cement+blast,
  data=compressbien,
  method="rf", trControl=control, tuneGrid=rfgrid,
  ntree=1000, sampsize=600, nodesize=10, replace=TRUE,
  importance=TRUE)
```

```
rf
```

Recomienda mtry=4, coincidiendo con bagging. Pero probaremos igualmente mtry=3 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))
```



## Validación cruzada repetida

```
source("cruzada arbol continua.R")
source("cruzadas avnnet y lin.R")
source("cruzada rf 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$modelo="avnnet"
medias2<-cruzadalineal(data=data,
vardep="cstrength",listconti=c("age","water","cement","blast"),
listclass=c(""),grupos=4,sinicio=1234,repe=5)
medias2$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=600,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"

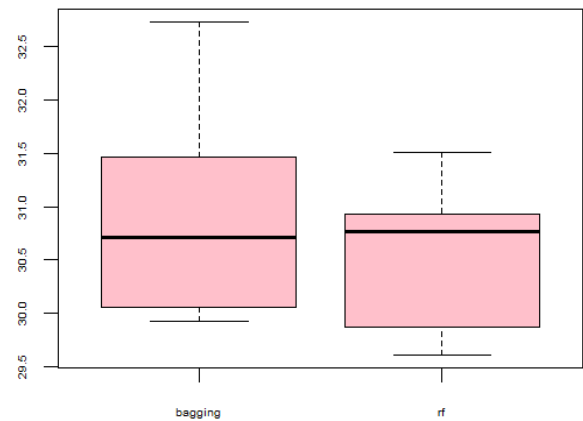
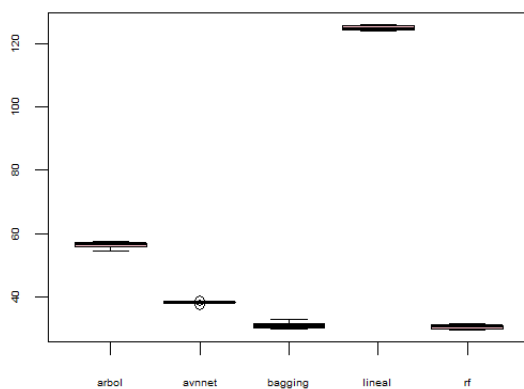
union1<-rbind(medias1,medias2,medias3,medias4,medias5)

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, sorteando  $mtry=3$  variables de las 4 en cada nodo en lugar de usar las 4 siempre como en bagging.

**Ejercicio: manipular sampsize para mejorar el modelo.**

#### Nota.

La manipulación del sampsize tiene implicaciones prácticas que lo hacen difícil de entender. Una vez decidido el modelo se construiría finalmente con todas las observaciones que tenemos en nuestros datos, digamos 1000 observaciones. Y con ese modelo se predecirían observaciones test que vendrían de fuera.

Por lo tanto nuestro límite para el sampsize es 1000 observaciones. Sin embargo, para hacer pruebas train-test o validación cruzada con nuestros datos, tendremos menos observaciones disponibles para construir el modelo. En este ejemplo de 1000 observaciones, si hacemos pruebas de validación cruzada con 10 grupos, tendríamos cada vez 900 observaciones máximo que podríamos plantear como sampsize, lo cual ya está muy cerca del valor real práctico. Por ello para obtener el sampsize óptimo hay que hacer pruebas con bastantes grupos de CV (10 está bien). Luego puede ocurrir que el sampsize óptimo para construir el modelo sea por ejemplo 300, lejos del máximo 900, pero eso depende de cada caso .

Por último, hay que recordar en Random forest y Bagging se puede dejar en blanco el parámetro sampsize, con lo cual el programa utiliza el total de datos training con reemplazamiento como sampsize, que es lo que propone el algoritmo original.

## Gradient Boosting

Es un algoritmo presentado por (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  $\hat{f}(x) = f_M(x)$ .

---

Tabla tomada del libro *Elements of Statistical Learning* (Hastie, Tibshirani)

Algunas funciones de error se presentan debajo. La función  $f(x)$  es la función de predicción o función base según el método que utilicemos (en general se utilizan árboles pero puede utilizarse cualquier método predictivo)):

**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: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Tabla tomada del libro *Elements of Statistical Learning* (Hastie, Tibshirani)

En la tabla anterior se ve que la derivada (el gradiente) de la función de error clásica en regresión es el residuo o diferencia entre el verdadero valor de la  $y$  y su predicción:  $r = y - f(x)$ .

Entonces básicamente, en regresión, el algoritmo gradient boosting consiste en modificar las predicciones en la dirección de decrecimiento del gradiente (en este caso el residuo). Si el residuo sale negativo en una observación (estamos prediciendo valores más altos que la realidad), se actualiza la predicción en la dirección de decrecimiento, es decir se reduce la predicción.

Una versión simplificada y menos general que la anterior se presenta debajo; es básicamente la que se utiliza en la práctica. El esquema de Gradient Boosting para variable dependiente continua sería:

### 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  $r_i^{(m)} = y_i - \hat{y}_i^{(m)}$

Es obvio que este residuo actual existe para los datos train, pero no para los datos test pues en estos no tenemos el valor de  $y$ .

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. Esto nos dará como resultado un residuo “predicho”  $\hat{r}_i^{(m)}$ , que no es exactamente igual que el real, pero nos sirve para actualizar las observaciones test, para las que no hay residuo real la no haber  $y$ .

De este modo se obtienen predicciones de los residuos para datos train y para datos test.

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

¿Cuánto se modifica la predicción de cada observación? Lo normal es que sea en una cantidad proporcional al valor del residuo, así si se comete mucho error se modifica mucho la predicción y si es pequeño, menos. Pero no conviene hacer grandes modificaciones, hay que dejar que el algoritmo modifique poco las predicciones en cada observación, que lo haga de forma gradual. Aquí interviene la constante de regularización  $\nu$  (letra griega nu) que toma valores pequeños (entre 0.0001 y 0.2 por ejemplo), para que en cada paso se modifique poco la predicción, en la buena dirección, pero poco.

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

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



Es conveniente señalar que los datos train convergen al verdadero valor de la  $y$ , así que no se debe tomar en ningún caso como referencia la performance del gradient boosting sobre datos train, sino solamente sobre datos test.

## Parámetros a determinar en el algoritmo Gradient Boosting

- La constante de regularización  $v$  (shrink) . Normalmente entre (0.0001 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  $v$  y muchas iteraciones.
- El número final de iteraciones-árboles **M**. A menor  $v$ , 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 reducirlo y se deja alto, 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.**

## Ejemplo

En este ejemplo se construye de forma manual el gradient boosting.

1) En la primera iteración (puntos pequeños color rojo en el gráfico) se fijan los valores iniciales de las predicciones de la variable  $y$  como su **media** (35.1), para todas las observaciones.

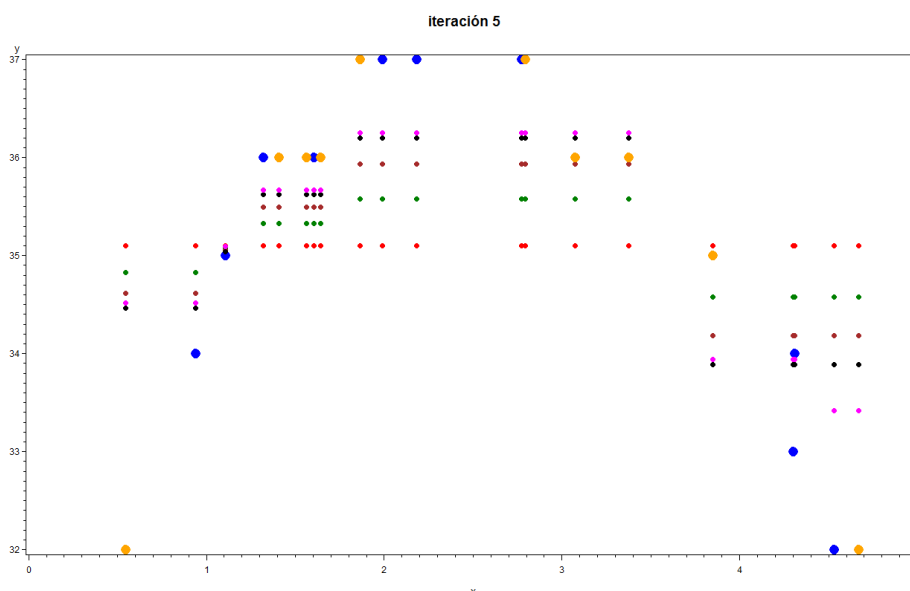
2) Se calcula el residuo real (**resi1**) que para la primera observación train toma valor -1.1 y para las observaciones test no existe al no existir la  $y$ .

3) Se construye un árbol de regresión, con **resi1** como variable objetivo, **x** como variable input. Esto da una predicción para resi1 (**resi1\_est**) que no es exactamente igual que resi1: en la observación train nº9 resi1 toma valor -1.1 y su predicción -2.1; además las observaciones test tienen valor predicho resi1\_est, al disponer de la variable predictora  $x$ .

4) Se actualiza la predicción de  $y$  (**y1**). En la primera observación train, de predecir con 35.1, se ha reducido la predicción en la buena dirección a 34.825; en las observaciones test se ha pasado a 34.825 y 34.575. En el gráfico aparece en color verde la predicción al final de esta primera iteración.

5) El proceso continúa: se calcularían los residuos, se predecirían, se actualizaría la  $y$  en cada iteración. Se observa cómo las observaciones reales train (puntos grandes azules) tienden a clavar perfectamente su predicción pero las observaciones reales test (puntos grandes naranjas), que son las que importan, también se predicen bastante bien. La quinta iteración está representada por los puntos pequeños rosa.

<b>y</b>	<b>ytest</b>	<b>x</b>	<b>media</b>	<b>resi1</b>	<b>resi1_est</b>	<b>y1</b>
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



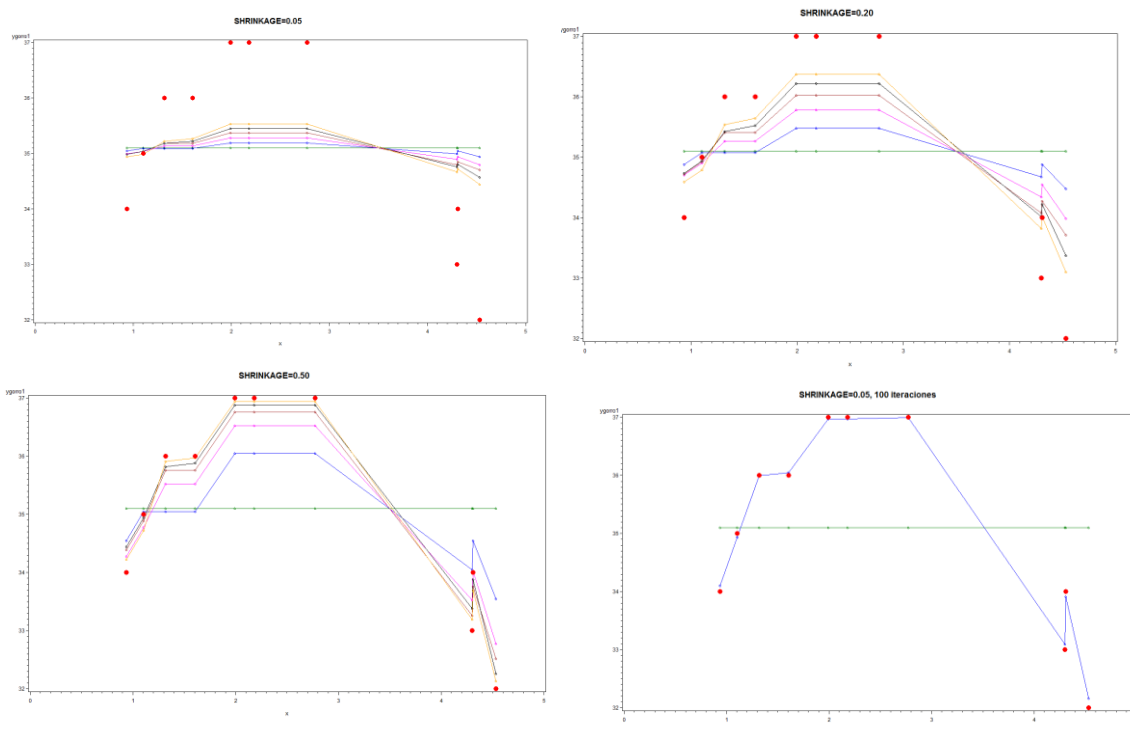
Datos train: azul

Datos test: naranja

La tabla completa para las 5 iteraciones está aquí:

				res1_				res2_				res3_				res4_			
y	ytest	x	media	res1	est	y1	res12	est	y2	res13	est	y3	res14	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				

En los gráficos siguientes se observa cómo cambiar la constante de regularización hace más rápida o lenta la convergencia. Con shrinkage (constante de regularización) bajo y 5 iteraciones (gráfica arriba izquierda) no se llega a aprovechar la potencia del gradient boosting. Es necesario poner el shrinkage alto, lo que en la práctica es poco fino en problemas complejos, o aumentar el número de iteraciones (gráfica abajo a la derecha).



## Algoritmo Gradient Boosting para clasificación

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.

### Algoritmo Gradient Boosting para clasificación binaria (función de error deviance)

La función de error en este caso es la deviance:

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

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.

## Ventajas del Gradient Boosting

- Invariante frente a transformaciones monótonas: no es necesario realizar transformaciones logarítmicas, etc.
- Buen tratamiento de missing, variables categóricas, etc. Universalidad.
- Muy fácil de implementar, relativamente pocos parámetros a monitorizar (número de hojas o profundidad del árbol, tamaño final de hojas, parámetro de regularización...).
- 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.
- Robusto respecto a variables irrelevantes. Robusto respecto a colinealidad. Detecta interacciones ocultas.
- Permite representar la importancia de variables

## 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).

No se pueden interpretar los resultados, es una técnica puramente predictiva, y es difícil desentrañar cómo construye las predicciones, más allá del cálculo de importancia de variables

## Gradient Boosting con caret

### Gradient boosting.R

#### 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.minobsinnode: 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

En este ejemplo procedemos a tunear los parámetros que permite tunear caret, que son los más importantes. interaction.depth lo dejamos en 2, árboles binarios, que funciona bien la mayoría de las veces.

```
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)

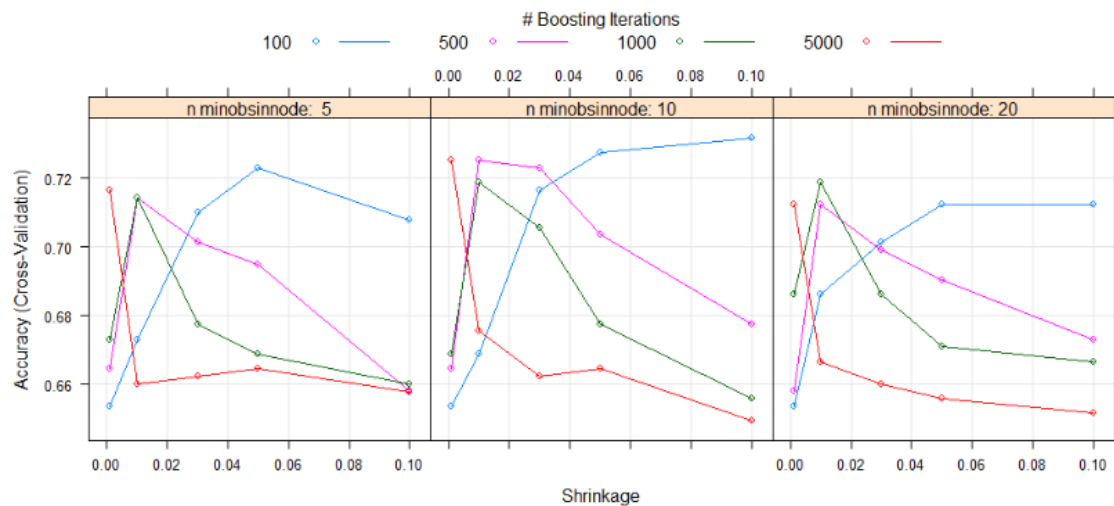
gbm<-
train(factor(chd)~age+tobacco+ldl+adiposity+typea+sbp+famhist.Absent,d
ata=saheartbis,
  method="gbm",trControl=control,tuneGrid=gbmgrid,
  distribution="bernoulli", bag.fraction=1,verbose=FALSE)

gbm

plot(gbm)
```

0.050	20	5000	0.6558471	0.22540073
0.100	5	100	0.7078336	0.32120172
0.100	5	500	0.6580022	0.22604831
0.100	5	1000	0.6602136	0.22917800
0.100	5	5000	0.6579460	0.22384346
0.100	10	100	0.7316154	0.37705682
0.100	10	500	0.6776049	0.26357022
0.100	10	1000	0.6559033	0.21717205
0.100	10	5000	0.6494190	0.21162992
0.100	20	100	0.7121627	0.32926706
0.100	20	500	0.6730885	0.25600182
0.100	20	1000	0.6666417	0.24783594
0.100	20	5000	0.6515180	0.21680697

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 = 0.1 and n.minobsinnode = 10.



Vemos en la tabla pero sobre todo en las gráficas, un comportamiento en patrones del algoritmo. Parece que la máxima accuracy se alcanza en general, o bien a) con un shrinkage entre 0.08 y 0.10 y 100 (pocos) árboles (early stopping), o bien b) muchos árboles y un valor de shrinkage muy bajo, 0.001. El número de observaciones en el nodo final debería ser `minobsinnode=10`.

Continuando en esta línea, probamos a fijar `shrinkage=0.1`, `minobsinnode=10` y observamos cuántas iteraciones-árboles sería bueno utilizar.

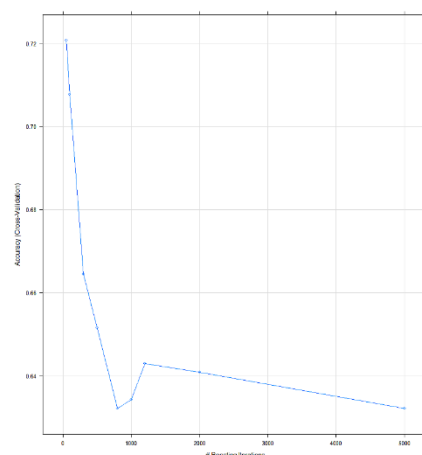
```
# ESTUDIO DE EARLY STOPPING
# Probamos a fijar algunos parámetros para ver como evoluciona
# en función de las iteraciones

gbmgrid<-expand.grid(shrinkage=c(0.1),
  n.minobsinnode=c(10),
  n.trees=c(50,100,300,500,800,1000,1200,2000,5000),
  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)

plot(gbm)
```



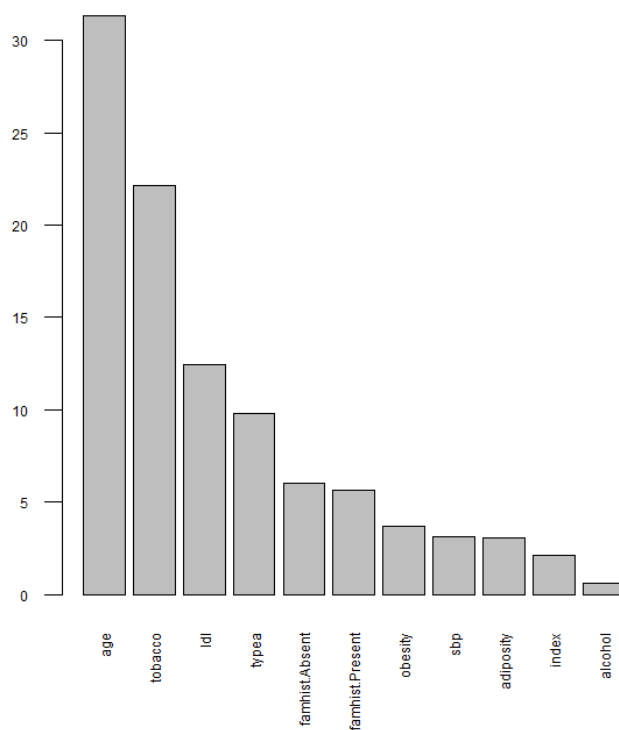
Parece que con 50 árboles es suficiente en este ejemplo.

## Importancia de variables

```
# 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
typea	typea	3.6711867
sbp	sbp	3.0982067
adiposity	adiposity	3.0428396
index	index	2.1035397
alcohol	alcohol	0.6273799





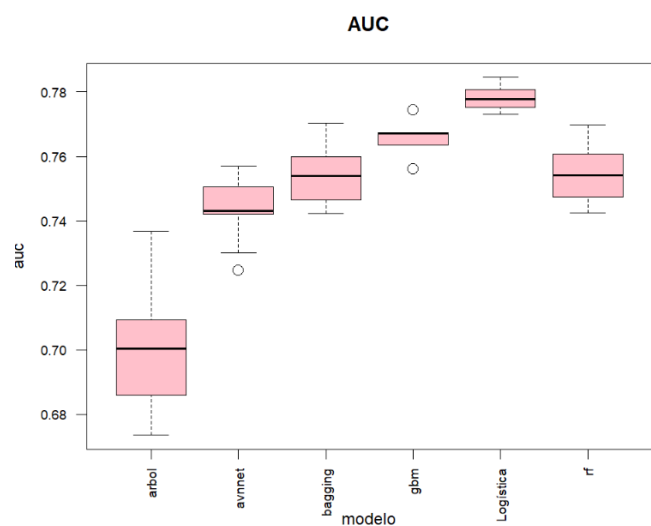
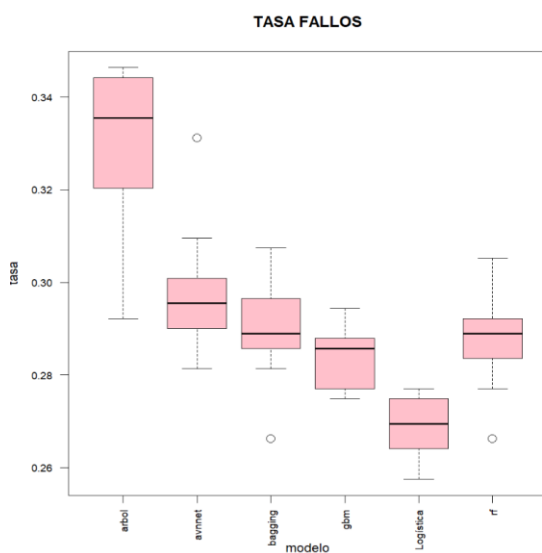
# La función `cruzadagbmbin` permite plantear gradient boosting para binarias

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

Obviamos aquí el resto del código. Solo indicar la parte relativa al gbm, donde se han fijado los parámetros elegidos en el proceso de tuneado.

```
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=0.1, n.trees=50, interaction.depth=2)

medias6$modelo="gbm"
```



Como se ve, aunque la logística es lo más apropiado para estos datos, el gbm es un algoritmo muy competitivo aún en datos simples como éstos.

## 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

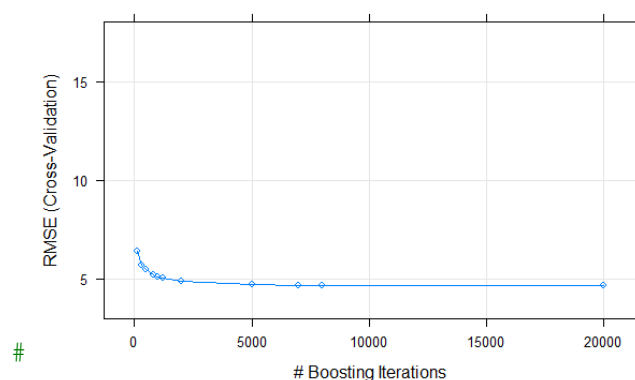
gbmgrid<-expand.grid(shrinkage=c(0.1),
  n.minobsinnode=c(10),
  n.trees=c(100,300,500,800,1000,1200,2000,5000,7000,8000,20000),
  interaction.depth=c(2))

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

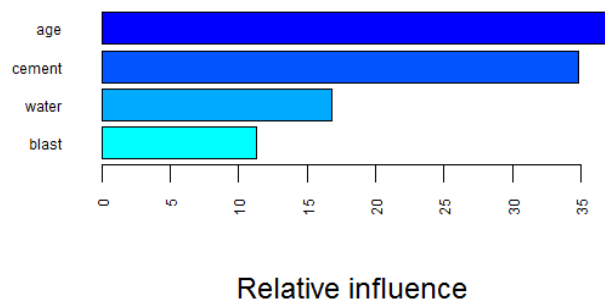
gbm<- train(cstrength~age+water+cement+blast,data=compressbien,
  method="gbm",trControl=control,tuneGrid=gbmgrid,
  distribution="gaussian", bag.fraction=1,verbose=FALSE)

gbm

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



```
summary(gbm)
```



```
source("cruzada rf continua.R")
source("cruzada gbm continua.R")
load("compressbien.Rda")
data<-compressbien
```

(Se omite aquí el resto del código)

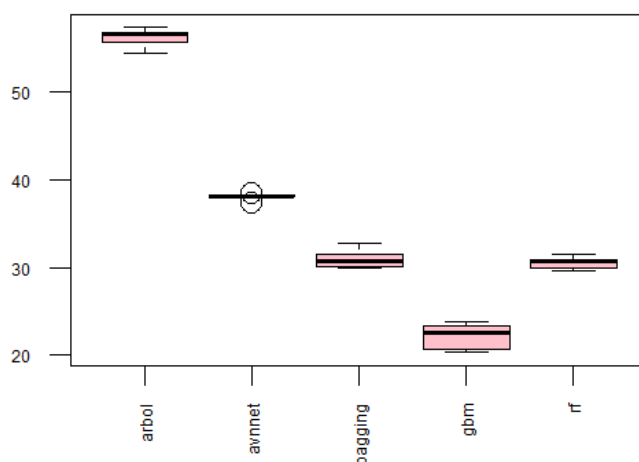
...

```
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")
```



Se observa cómo el gradient boosting se adapta mejor que otros modelos a esa relación no lineal entre la variable dependiente y las inputs.

## La última moda: Xgboost

En los concursos de Kaggle, desde hace pocos 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\_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 más, 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

### Archivo Xgboost 2.0.R

```
# TUNEADO DE XGBOOST CON CARET
#
# nrounds (# Boosting Iterations)=número de iteraciones
# max_depth (Max Tree Depth)=profundidad máxima de los árboles
# eta (Shrinkage)=parámetro  $\eta$  de gradient boosting
# gamma (Minimum Loss Reduction)=gamma
#   # cte regularización. Dejar a 0 por defecto
# colsample_bytree (Subsample Ratio of Columns)
#   # % Sorteo variables antes de cada árbol ,
#   # al estilo de random forest pero antes del árbol, no en cada nodo. Dejar
#   # a 1 por defecto.
# min_child_weight (Minimum Sum of Instance Weight).
#   # observaciones mínimas en el nodo final. Similar al minobsinnode del gbm.
# subsample (Subsample Percentage)
#   # % Sorteo de observaciones antes de cada árbol , al estilo de random forest.
#   # Dejar a 1 por defecto.

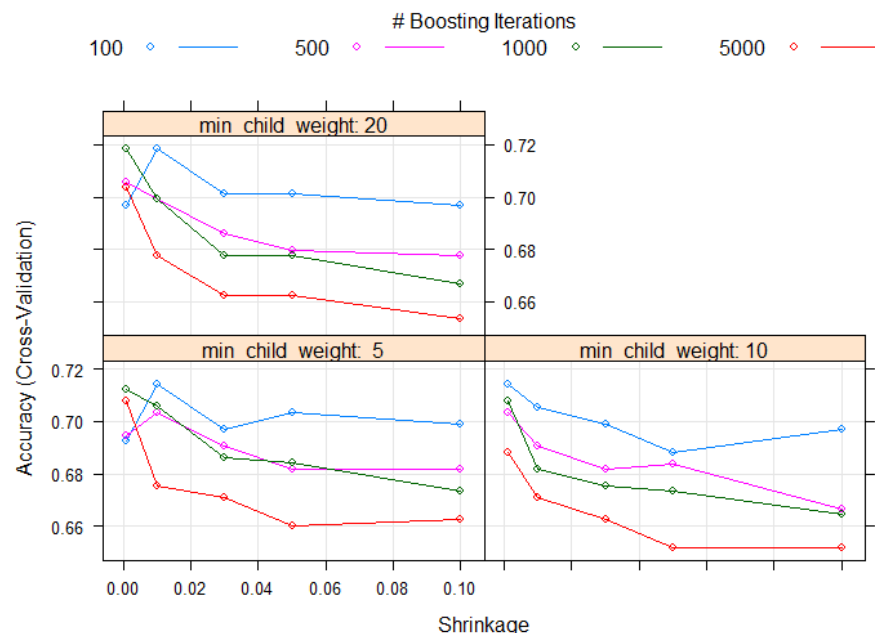
library(caret)
set.seed(12345)
xgbmgrid<-expand.grid( min_child_weight=c(5,10,20),
eta=c(0.1,0.05,0.03,0.01,0.001), nrounds=c(100,500,1000,5000),
max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

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)
```



De los gráficos anteriores se deduce que se debe de tomar un shrinkage bajo y un número de iteraciones alto, 1000, 5000. Pero podría ser un número de iteraciones menor también si se fija min child weight algo alto. Por ejemplo, caret recomienda nrounds = 100, max\_depth = 6, eta = 0.01, min child weight=20. Si las diferencias son pequeñas es siempre mejor tomar un minchild weight alto (pues hace modelos más simples y menos sobreajustados).

```

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

xgbmgrid<-expand.grid(eta=c(0.01),
  min_child_weight=c(20),
  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

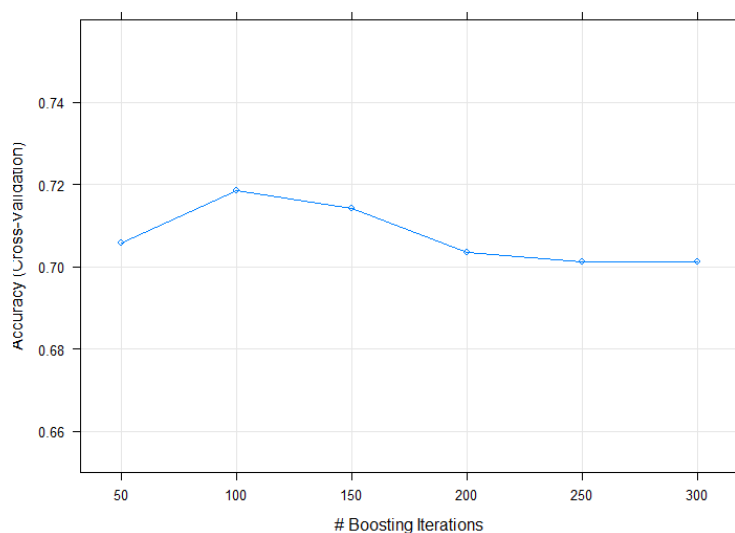
set.seed(12367)
control<-trainControl(method = "cv",number=4,savePredictions = "all",
  classProbs=TRUE)

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 o 100 iteraciones es suficiente (early stopping). Se puede probar variando la semilla en `set.seed` para ver si estos resultados son estables o erráticos. Se observa que 100 iteraciones está bien en términos generales (aunque el valor concreto de la accuracy varía por supuesto, siempre es mejor con iteraciones =100).



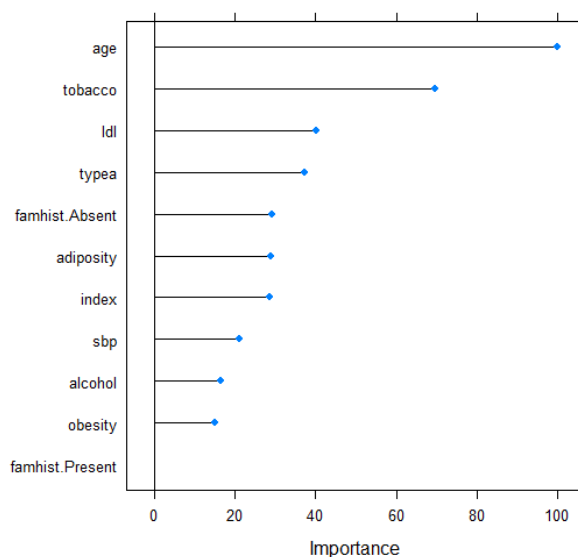


## # IMPORTANCIA DE VARIABLES

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

Por probar hemos metido todas las variables en el modelo xgbm. Lo normal es hacer una preselección antes de tunear los parámetros, pero se ha hecho para comprobar el funcionamiento de la importancia de variables en xgboost.

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

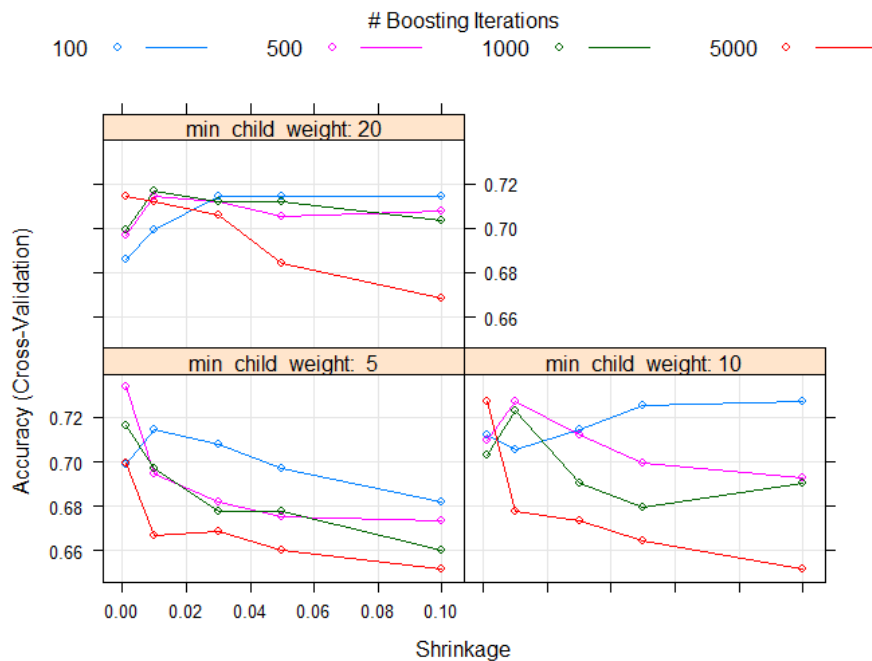
```
xgbmgrid<-expand.grid(
  min_child_weight=c(5,10,20),
  eta=c(0.1,0.05,0.03,0.01,0.001),
  nrounds=c(100,500,1000,5000),
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

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

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

xgbm
plot(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.



A pesar de la recomendación de caret parece más fiable usar minchild weight =10, con shrinkage alto=0.10 y iteraciones=100. Se puede probar también la opción recomendada por caret (500 iteraciones, v=0.001, minchild weight =10) y comparar con validación cruzada repetida.

De momento no utilizamos los parámetros de regularización, al no tunearlos caret y a menudo no ser muy estables. Podemos después hacer pequeñas modificaciones en nuestras pruebas de validación cruzada repetida al final de las comparaciones modelo.

### Ejemplos con validación cruzada repetida

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

Omitimos el resto del código...

```
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=0.10, 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,
```

```

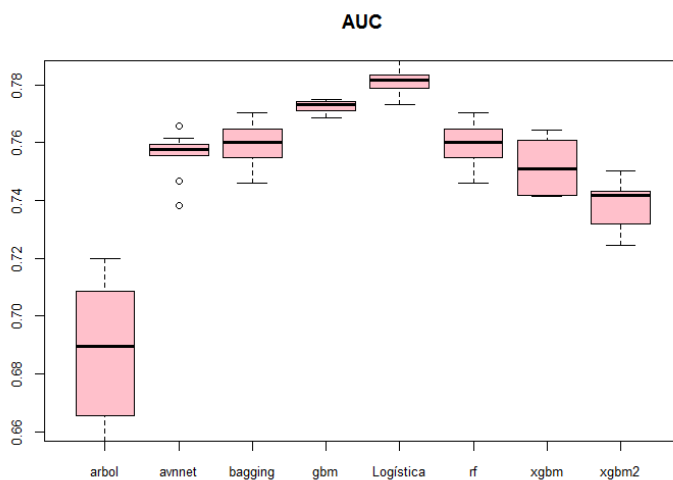
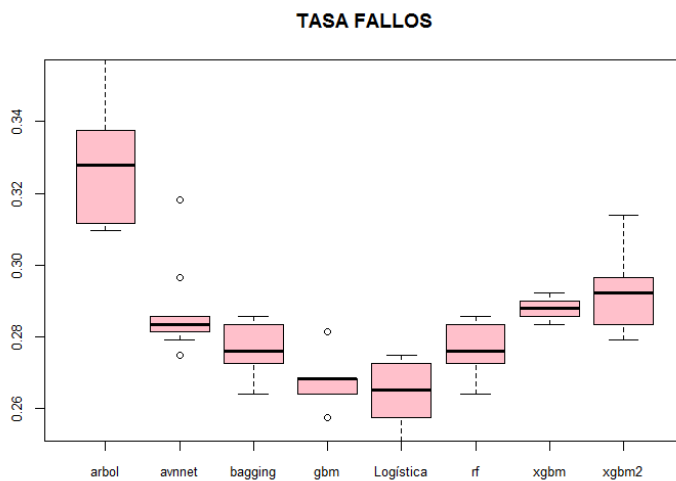
min_child_weight=10,eta=0.001,nrounds=500,max_depth=6,
gamma=0,colsample_bytree=1,subsample=1,
alpha=0,lambda=0,lambda_bias=0)

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")

```



Como se observa en este caso el paquete xgboost no mejora al gbm, como se ha dicho esto puede ocurrir pues la manera de construir los arboles es diferente en ambos métodos.

Sin embargo el paquete xgboost tiene la ventaja de poder disponer de sorteo de observaciones y de variables ; aunque esto serviría más bien para reducir la varianza y no el sesgo, que es el problema en este caso, hacemos algunas pruebas.

Al sortear variables u observaciones, previsiblemente habrá que aumentar el número de árboles para obtener un sesgo similar; probamos con 200 iteraciones en lugar de 100, aunque habría que tunearlo en serio.

```
# 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("tobacco", "ldl", "age", "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)

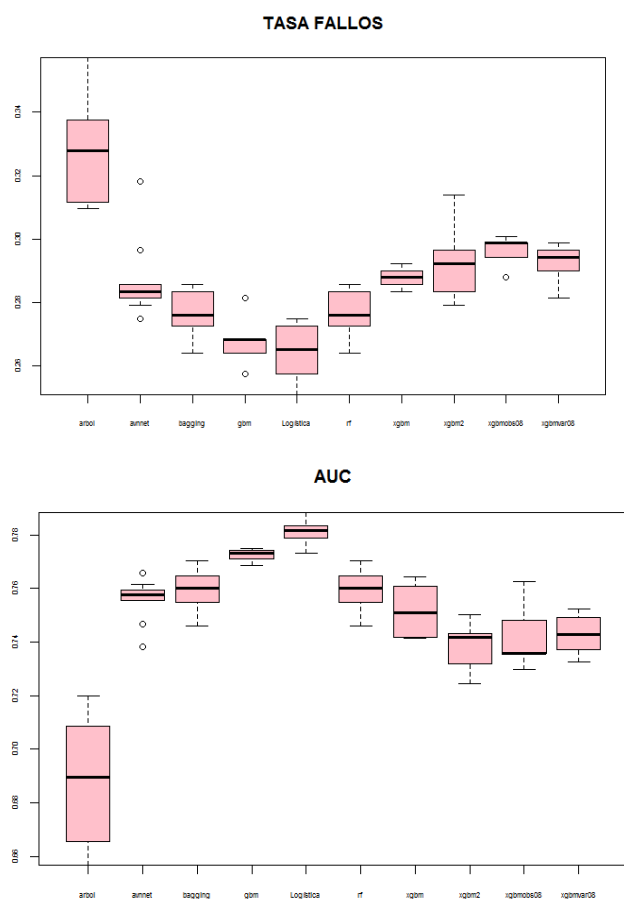
medias9$modelo="xgbmvar08"

medias10<-cruzadaxgbmbin(data=saheartbis, vardep="chd",
  listconti=c("tobacco", "ldl", "age", "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")
```



No se observa mejora alguna. Al ser datos tan sencillos y el problema estar en el sesgo no se pueden aprovechar bien las opciones del xgboost.

### Ejemplo con variable dependiente continua.

Con variable 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")

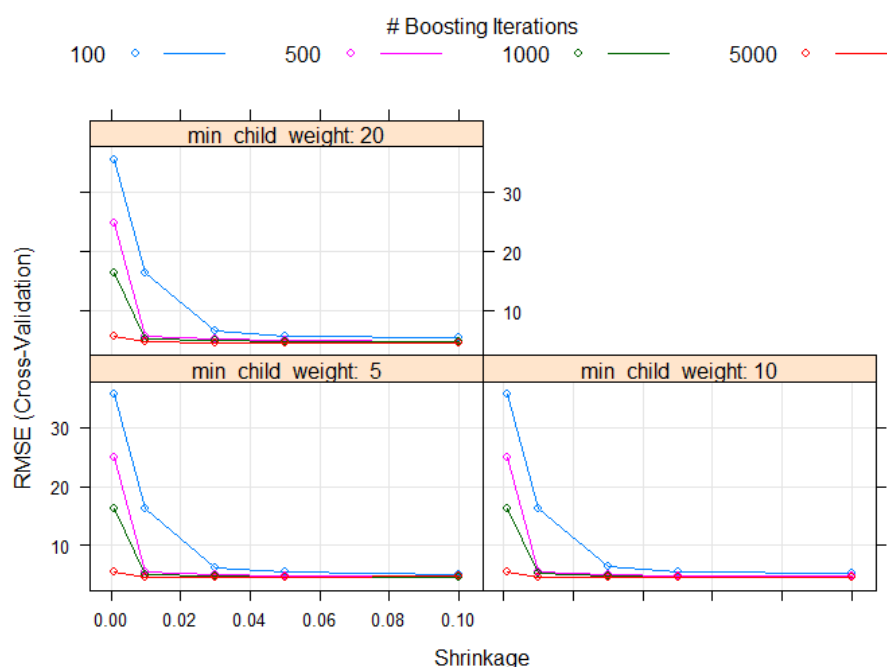
comp<-compressbien[,c("cstrength","age","water","cement","blast")]

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)
```



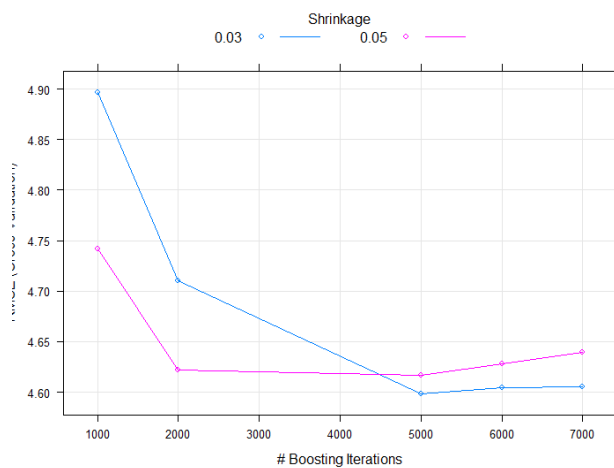
```
# ESTUDIO DE EARLY STOPPING
# Probamos a fijar algunos parámetros para ver como evoluciona
# en función de las iteraciones
(Atención, puede tardar bastante)

xgbmgrid<-expand.grid(eta=c(0.03,0.05),
  min_child_weight=c(10),
  nrounds=c(1000,2000,5000,6000,7000),
  max_depth=6,gamma=0,colsample_bytree=1,subsample=1)

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

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

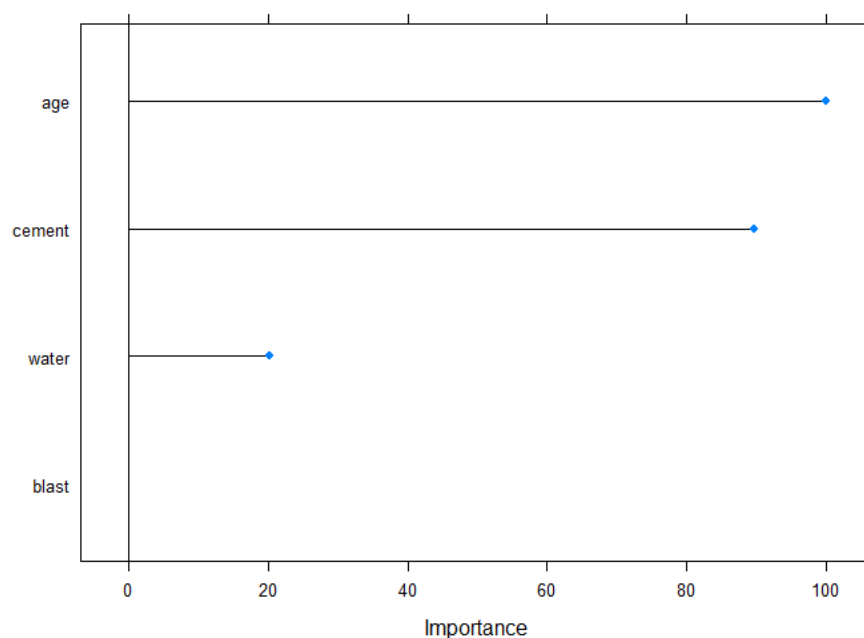
plot(xgbm)
```



Parece que con  $v=0.03$  y 5000 iteraciones está bien.

```
# IMPORTANCIA DE VARIABLES
```

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



### 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")
```

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

Omitimos el resto del código...

```
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)
```

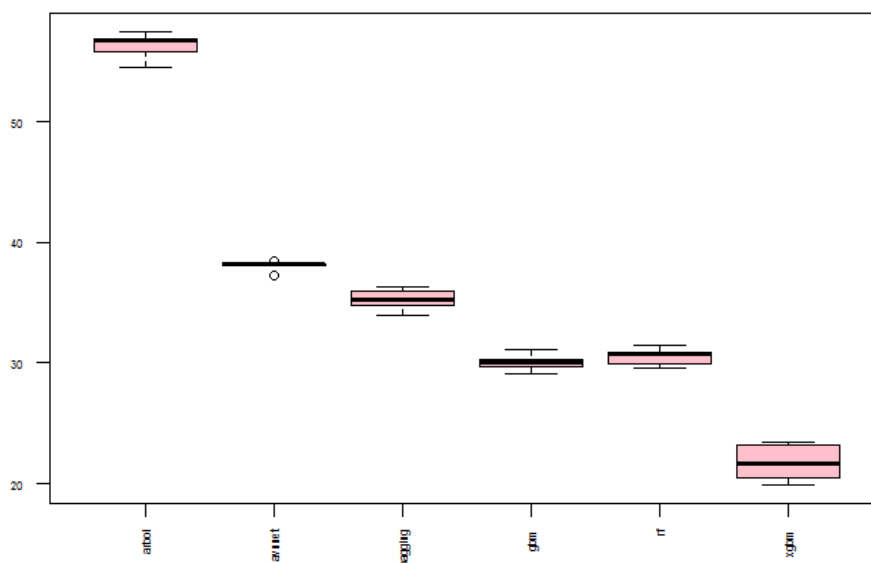
```
medias7$modelo="xgbm"
```

```
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")
```



En este caso es obvio que el xgbm mejora al gbm. Si fuera necesario se probarían técnicas para reducir la varianza, lo hacemos como ejercicio, aunque no parezca necesario en este caso, probando sorteo de observaciones (de variables no va a ser útil, al ser tan pocas) y tocando algo el parámetro alpha de regularización.

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

```
medias8<-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=0.8)
```

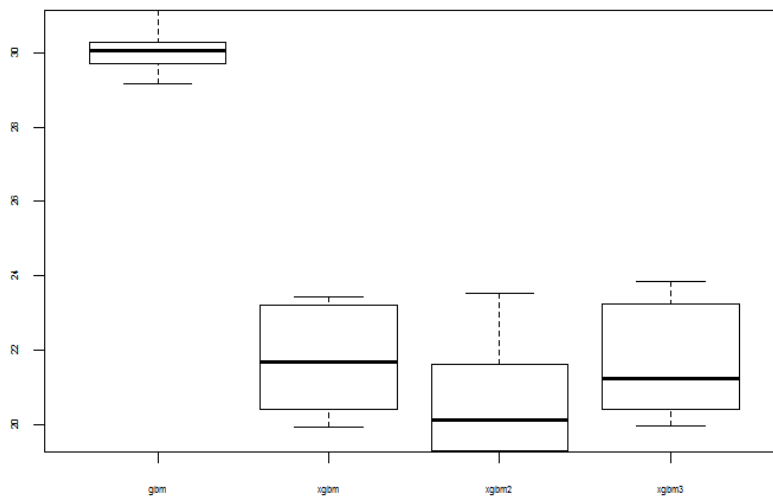
```
medias8$modelo="xgbm2"
```

```
medias9<-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,alpha=0.5)
```

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

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

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




Se observa que no hay mucha mejora bajo ninguna de las dos modificaciones.













## Kaggle y Gradient Boosting

La plataforma de concursos de predicción online Kaggle ofrece desde 2010 la posibilidad de concursar planteando modelos predictivos sobre datos reales. Las empresas ofrecen datos de entrenamiento y esponsorizan el concurso, ofreciendo premios monetarios a aquellos concursantes que obtienen menor error objetivo sobre un conjunto de datos test.



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

Los temas y problemas predictivos a tratar son de gran variedad; hay problemas de clasificación o de regresión, de clasificación de imágenes o texto, etc. Suelen ser problemas complejos, pues las empresas no pagarían de otro modo por su solución.

Desde 2010 hasta octubre del 2019 ha habido 356 competiciones. En cada momento hay aproximadamente entre 15 y 20 competiciones activas. Algunas de ellas son académicas o simplemente de prueba, sin premio. Desde 2017 hay más de 1 millón de usuarios registrados en Kaggle. La tabla siguiente presenta el ranking de los mejores primeros concursantes en 2019.

Competitions

Kernels

Discussion

Learn more about rankings >

162

Grandmasters

1,318

Masters

4,709




































Experts

50,609

Contributors

66,466

Novices

Rank	Tier	User		Medals	Points
1		 <b>bestfitting</b>	joined 3 years ago	 23  4  0	248,913
2		 <b>Guanshuo Xu</b>	joined 4 years ago	 9  14  2	166,033
3		 <b>Giba</b>	joined 7 years ago	 47  36  26	164,691
4		 <b>Pavel Pleskov</b>	joined 4 years ago	 14  25  12	159,965
5		 <b>Μαριος Μιχαηλιδης KazAnova</b>	joined 6 years ago	 36  44  30	147,150
6		 <b>n01z3</b>	joined 4 years ago	 12  18  6	136,687
7		 <b>ZFTurbo</b>	joined 4 years ago	 21  22  9	124,095

Debido a que en Kaggle se compite por dinero por obtener el menor error en datos test, es una buena referencia para observar el estado del arte en machine learning. Muchas modificaciones o creaciones de algoritmos nuevos, como xgboost, han surgido de competiciones de Kaggle, mejorando el conocimiento científico en el sector. Nuevas herramientas informáticas (paquetes en R y Python) van adaptándose y enriqueciéndose con las aportaciones de los concursantes.

Es interesante ver cómo el algoritmo estrella en Kaggle ha ido deveniendo en xgboost. Vemos ejemplos de la evolución de los algoritmos utilizados por los concursantes. No es una información completa pues los concursantes, incluidos los ganadores, no siempre dicen públicamente qué algoritmos utilizan.

## 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 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.

**Algunos comentarios sobre Kaggle y Machine Learning****1) No universalidad de los datos de Kaggle**

Los problemas planteados por dinero en Kaggle suelen ser complejos: muchas variables, muchas observaciones, ruido (errores de medición, datos missing, categorías mal clasificadas, etc.). Los métodos clásicos (regresión, regresión logística) suelen ser insuficientes, pues no hay relaciones o separación lineal, y la complejidad de la no linealidad, muchas variables categóricas, missings, etc. hacen que sean más eficaces los algoritmos basados en árboles para este tipo de datos.

Sin embargo en el mundo real (banca, seguros, medicina, etc.) las relaciones lineales, (o separación lineal en caso de clasificación), y también las relaciones sencillas, existen con cierta frecuencia. A menudo con pequeñas transformaciones, feature engineering y un modelo clásico es suficiente, y siempre serán preferibles estos modelos por su explicabilidad y solidez teórica. Por lo tanto considerar xgboost como una panacea para todo debido a su uso indiscriminado en Kaggle es un error, pues no estamos viendo la población de todos los problemas reales, sino solo los más complejos, aquellos que se prestan a un concurso de predicción y las empresas están dispuestas a sponsorizar.

**2) Evaluación incompleta a nivel práctico**

Las medidas de evaluación de Kaggle son simples, (MSE, Accuracy, etc.) pero en la práctica el decisor o especialista tiene que tener en cuenta factores como el coste de obtener información de ciertas variables input, si tiene sentido meter variables input poco fiables o extrañas, si la ganancia en error compensa un modelo más complicado, el tener que explicar el modelo a los jefes o clientes, la falta de razonamientos probabilísticos, etc.

### **3) Diferencias no tan evidentes entre los modelos**

A menudo (si no siempre) las diferencias entre la performance de unos modelos o concursantes sobre otros es debida al azar, a pesar de los esfuerzos en Kaggle por hacer varias evaluaciones, usar muchos datos test, etc.

Suele ocurrir que el software utilizado (paquetes en R, Python, C, etc.) marque la diferencia, no el algoritmo en sí. Además el disponer de hardware potente permite un tuneado más exhaustivo, y también establece diferencias entre concursantes.

### **4) Fuente de aprendizaje y experiencia**

A pesar de los comentarios anteriores Kaggle permite aprender cómo se trabaja con machine learning, las herramientas de moda, trucos de feature engineering y modelado, ensamblados, comparación de modelos, etc. Se pueden encontrar en el Foro de Kaggle scrips completos en R o Python para abordar problemas, ideas interesantes y aprender de las prácticas de los demás.

## Bibliografía básica

### Libros disponibles en PDF

Hastie, Tibshirani: The Elements of Statistical Learning (PDF)

(En la web hay más información)

<http://statweb.stanford.edu/~tibs/ElemStatLearn/>

Hastie, Tibshirani: An Introduction to Statistical Learning with Applications in R (PDF)

(básicamente el mismo que el anterior, pero para R)

<http://www-bcf.usc.edu/~gareth/ISL/data.html/>

### Paquetes R

<https://cran.r-project.org/web/packages/randomForest/index.html>

<https://cran.r-project.org/web/packages/gbm/index.html>

<https://cran.r-project.org/web/packages/xgboost/index.html>