

Lec7: Random Forest y XGB

Isidoro Garcia Urquieta

2023

Agenda

- ▶ Mini Repaso Trees
- ▶ Bootrapping
- ▶ Bagging
- ▶ Random Forests
- ▶ Boosting
- ▶ Gradient Boosting Machines

Resumen Trees

- ▶ Los árboles de decisión (Regresión y Clasificación) son algoritmos no paramétricos
- ▶ Son fáciles de interpretar!
- ▶ Detectan no linealidades de manera automática
- ▶ Se estiman de manera **greedy**, donde se buscan splits que maximicen la varianza entre nodos (inter-varianza) y se minimice la varianza dentro de cada nodo (intra-varianza)
- ▶ Hay dos maneras de parar la estimación de los arboles: observaciones mínimas en los nodos terminales o disminución en el deviance mínima.
- ▶ El greediness es miope. Esto es, un split malo puede seguirle un gran split.
- ▶ Es difícil evitar el overfitting en los árboles.

Cost complexity pruning

Cost complexity pruning (Weakest link pruning) es la manera más común de reducir el número de modelos candidatos y aplicar k-fold CV.

El proceso funciona así:

1. Estima árbol de manera greedy hasta llegar a `min obs`.
2. Aplica un parámetro de complejidad α que castigue la complejidad del árbol. Esto te dará una secuencia de subárboles como función de α .
3. Aplica k-fold CV para escoger α

Cost complexity pruning

Veamos la formula de cost complexity pruning:

$$\sum_{j=1}^J \sum_{R_j} (y_i - \hat{y}_i)^2 + \alpha |T|$$

Donde T es el número de nodos terminales. Se ve familiar?

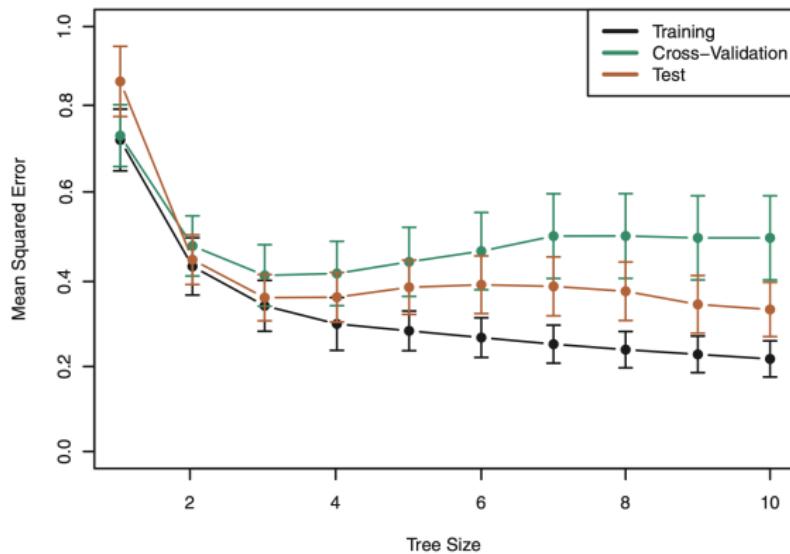
Resulta que si un α muy grande, tendremos un árbol con $T_0 \subset T$.

Mientras α tiende a cero $T_0 \rightarrow T$.

α nos ayuda a tener un set de candidatos de modelo estimable vía K-fold CV!!

Grafica de desempeño de CV tree

Noten como el CV es un buen estimador del desempeño fuera de la muestra (3 nodos terminales). Adicionalmente, noten como en el training set, el árbol siempre parece que va a estimar mejor mientras más grande haga los árboles.



Resumen Tree

- ▶ Para ayudar al overfitting, podemos hacer Tree pruning con Cross Validation
- ▶ Esto hará que estimemos greedy y luego tengamos un set de árboles candidatos dependiendo de cuánto castigamos la complejidad de los mismos.
- ▶ Los prune.trees son más poderosos (predicción) que los árboles normales. Por ende, preferibles.
- ▶ Aún así, los árboles son estimadores de alta varianza. Esto es, varían mucho dependiendo de la muestra en la que nos encontramos (no paramétrico). Por ende, necesitamos alguna manera de estabilizar las predicciones de los árboles para lograr mejor poder predictivo OOS.

Boostrapping

Llegamos al segundo método de **remuestreo** (El primero fue CV)!

El boostrapping es muy útil para conseguir errores estandares e intervalos de confianza cuando la distribución de algún estimador es desconocida, tenemos pocas observaciones o es muy difícil de estimar.

Por ejemplo, en el caso de la econometría, podemos inferir la distribución de nuestro estimador $\hat{\beta}$ al remuestrear la base **con reemplazo** y estimar β con cada muestra.

Lo bonito del Boostrapping es que tiene otras aplicaciones además de la generación de errores standar. Se puede aplicar a mejorar el poder predictivo de nuestros modelos estadísticos. Podemos usarlo para disminuir la varianza de los árboles.

Boostrapping

Pasos:

Dada una base de datos $\{z_i\}_{i=1}^n$, para $b \in \{1, 2, 3, \dots, B\}$ muestras:

1. Tomas una muestra **con reemplazo** $\{z_i^b\}_{i=1}^n$
2. Estimas β_b usando la base de (1)

Luego, $\{\beta_b\}_{b=1}^B$ es una aproximación de la distribución de $\hat{\beta}$

El vector de tamaño B $\{\beta_b\}_{b=1}^B$ puede ser usado como la distribución teorica que aprendimos en la clase 2 (inferencia!) que usaba el Teorema del Límite Central.

$$\beta \in \hat{\beta} \pm 2sd(\hat{\beta}_b)$$

$$se(\hat{\beta}) \simeq sd(\beta_b) = \sqrt{\frac{1}{B} \sum_b (\hat{\beta}_b - \hat{\beta})^2}$$

Donde $\hat{\beta}$ es el estimador original usando toda la base de datos.

Bootstrapping

Como podríamos usar Bootstrapping para mejorar un modelo predictivo usando bootstrapping? Cómo se combinaría con Cross-Validation?

Bagging

Los árboles de decisión sufren alta varianza. Esto es, cuando dividimos la muestra en entrenamiento y validación y entrenamos el árbol (c/ CV pruning), los resultados pueden ser muy distintos en cada base.

En el caso del LASSO, esto no era así. Porqué?

- ▶ Por que teníamos **estimadores** paramétricos que se estabilizan con cierta facilidad ($n > p$).

El **Boosting Aggregation (Bagging!)** es un procedimiento muy utilizado para estabilizar la varianza en aprendizaje estadístico.

Bagging

El Bagging utiliza el poder de las medias para crear estimadores estables.

Recuerden que si tenemos un set de n observaciones independientes Z_1, Z_2, \dots, Z_n y *tenemos un procedimiento insesgado*:

Promediamos las Z_i : \bar{Z}

Sabemos que:

$$\bar{Z}_n \xrightarrow{D} N(\mu_Z, \sqrt{\sigma^2/n})$$

En otras palabras, la varianza original de los datos σ^2 se reduce cuando sacamos promedios a σ^2/n !

Bagging en Árboles de Decisión

Pasos para aplicar Bagging en árboles de decisión:

1. Construye B muestras con reemplazo
2. Construyes un árbol en cada muestra B : $\hat{f}^b(X)$ $b \in \{1, \dots, B\}$
3. Construye el Bag estimate:
 - ▶ Para árboles de regresión: $\hat{f}_{bag}(X) = \frac{1}{B} \sum_{b=1}^B \hat{f}^b(X)$
 - ▶ Para árboles de clasificación: Majority vote. La clase más vota es a predicción.

Queda pendiente algo: Cuántas submuestras B tomar? Con B alto caemos en overfitting?

Bagging

Cuantas B tomar?

- ▶ En la práctica, necesitamos B suficientemente grandes para estabilizar la varianza. Esto implica prueba y error en la práctica.
- ▶ Otro punto importante es que un número grande B no induce overfitting, al contrario.
- ▶ Con: Estimar los árboles con demasiadas submuestras puede ser computacionalmente complejo. No obstante es alcanzable en una buena computadora (Parallel Computing!)

Bagging

Cómo decidirían cuantos árboles estimar usando Bagging?

Out-of-Bag

Resulta que, como en Cross-Validation, hay manera de estimar el error OOS desde la muestra de entrenamiento.

Recuerden que en cada muestra B , dejamos algunas observaciones sin tomar. Estas observaciones pueden servir como estimador del error. Su nombre es out of bag.

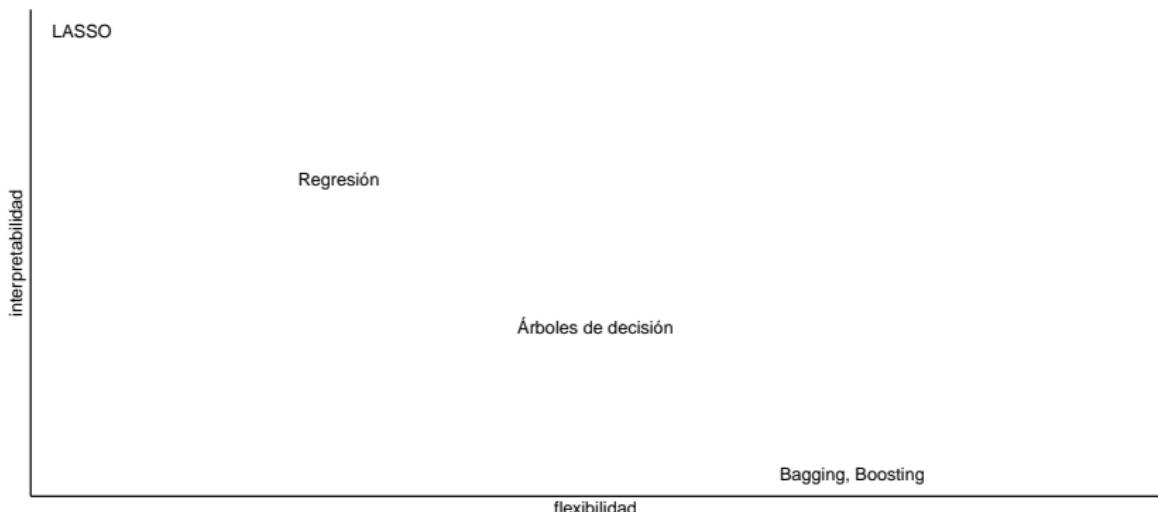
Los resultados muestran que en la práctica, los bagging trees superan en poder de predicción a los trees y pruned trees.

Terminamos? que cosas se les ocurren que pueden seguir pasando con los Bagging Estimators?

Interpretabilidad

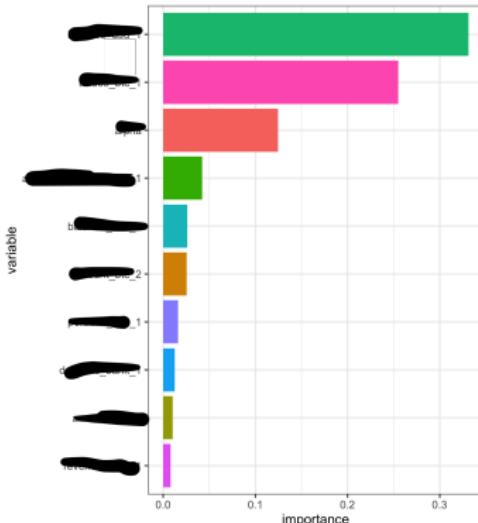
El bagging nos permite estabilizar las predicciones de los árboles y ganar poder predictivo.

El costo asociado es la **interpretabilidad**



Variable Importance

Una manera de interpretar los resultados es contar para cada variable cuántos splits de hicieron. Un valor grande es un indicativo de que esa variable es muy útil para predecir y .



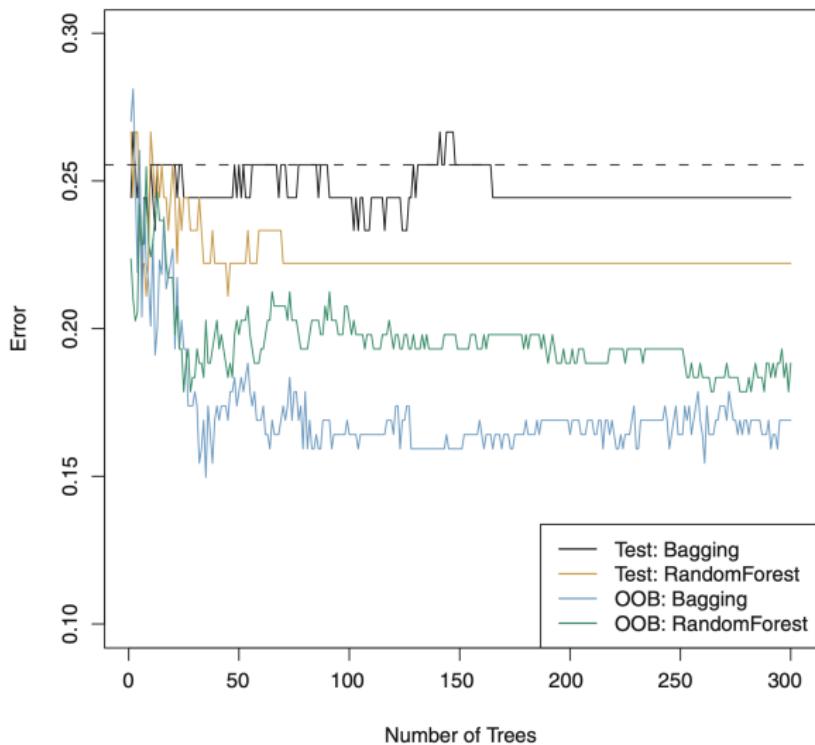
Random Forests

Un problema de los Bagging Trees es que incluso con el remuestreo, los árboles tienen mucha probabilidad de ser muy parecidos entre ellos. Muy **correlacionados**.

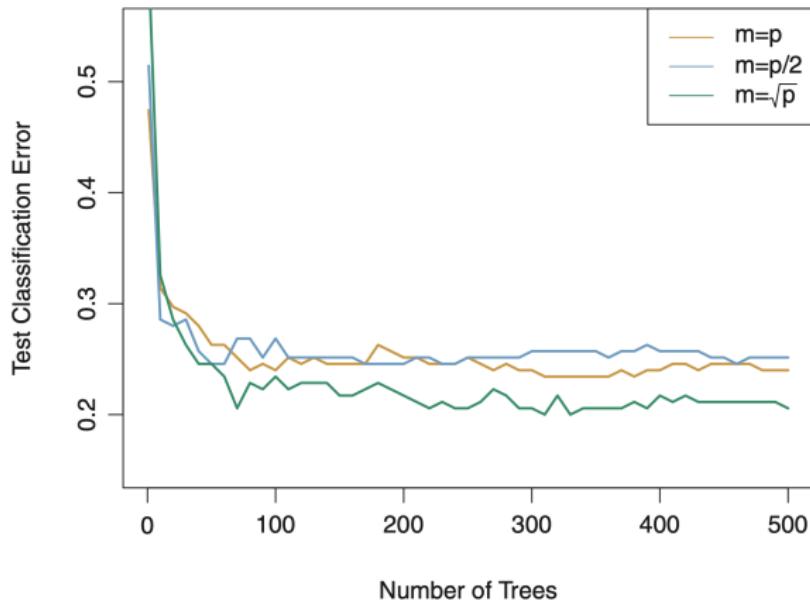
Un Random Forest resuelve esto así (para una base (n, p)):

1. Construye B muestras con reemplazo
2. **Toma una muestra de las columnas** $m = \sqrt{p}$
3. Construyes un árbol en cada muestra B : $\hat{f}^b(X) \ b \in \{1, \dots, B\}$
4. Construye el Bag estimate:
 - ▶ Para árboles de regresión: $\hat{f}_{bag}(X) = \frac{1}{B} \sum_{b=1}^B \hat{f}^b(X)$
 - ▶ Para árboles de clasificación: Majority vote. La clase más vota es a predicción.

Random Forest vs Bagging Trees



Cuántas columnas m



Random Forest: Hiperparámetros

Noten como los hiperparámetros a estimar van aumentando:

- ▶ En regularización L1 o L2 (LASSO, Ridge) sólo teníamos a λ estimable via CV
- ▶ En los árboles teníamos `min.obs` y `min.dev`. Como se estimaban?

En el Random Forest tengo:

- ▶ `min.obs`
- ▶ `min.dev`
- ▶ `m`
- ▶ `B` o `num.trees`

Acá necesitamos estimar a prueba y error. Optimizamos el algoritmo como una optimización numérica.

Random Forests en R

library(ranger) y library(randomForest) son las librerías más comunes en R. La primera es lo mismo que la segunda pero mucho más rápida en estimar.

```
detectCores()  
cl<-makeCluster(12)  
cl  
  
# Estimation  
a<-Sys.time()  
rf<-ranger(y~., data = x, classification = T, num.trees = 750)  
  
Sys.time() -a  
  
save(rf, file = 'Modelos/rf_1000.Rdata')  
stopCluster(cl)
```

ranger prediction

```
# Out of Bag
oob<-rf$predictions

# prediction vectors
pred_val<-predict(rf, data = validation)$predictions
```

Ranger



ranger (<ranger>)

R Documentation

Ranger

Description

Ranger is a fast implementation of random forests (Breiman 2001) or recursive partitioning, particularly suited for high dimensional data. Classification, regression, and survival forests are supported. Classification and regression forests are implemented as in the original Random Forest (Breiman 2001), survival forests as in Random Survival Forests (Ishwaran et al. 2008). Includes implementations of extremely randomized trees (Geurts et al. 2006) and quantile regression forests (Meinshausen 2006).

Usage

```
ranger:  
formula = NULL,  
data = NULL,  
num.trees = 500,  
mtry = 3,  
importance = "none",  
write.forest = TRUE,  
probability = FALSE,  
min.node.size = NULL,  
max.depth = NULL,  
replace = TRUE,  
sample.fraction = ifelse(replace, 1, 0.632),  
case.weights = NULL,  
class.weights = NULL,  
weights = NULL,  
num.random.splits = 1,  
alpha = 0.5,  
min.node.size = 0.1,  
split.select.weights = NULL,  
always.split.variables = NULL,  
respect.unordered.factors = NULL,  
scale.permutation.importance = FALSE,  
local.improvement = FALSE,  
quantile.permutation.factor = 1,  
regularization.used.edepth = FALSE,  
keep.inbag = FALSE,  
inbag = NULL,  
allow.trap = FALSE,  
quantreg = FALSE,  
oob.error = TRUE,  
num.threads = NULL,  
save.memory = FALSE,  
prob.above = TRUE,  
seed = NULL,  
dependent.variable.name = NULL,  
dependent.variable.name = NULL,  
classification = NULL,  
x = NULL,  
y = NULL  
)
```

Ranger



Arguments

<code>formula</code>	Object of class <code>formula</code> or <code>character</code> describing the model to fit. Interaction terms supported only for numerical variables.
<code>data</code>	Training data of class <code>data.frame</code> , <code>matrix</code> , <code>dgCMatrix</code> (<code>Matrix</code>) or <code>gwaa.data</code> (<code>GwABEL</code>).
<code>num.trees</code>	Number of trees.
<code>mtry</code>	Number of variables to possibly split at in each node. Default is the (rounded down) square root of the number variables. Alternatively, a single argument function returning an integer, given the number of independent variables.
<code>importance</code>	Variable importance mode, one of 'none', 'impurity', 'impurity_corrected', 'permutation'. The 'impurity' measure is the Gini index for classification, the variance of the responses for regression and the sum of test statistics (see <code>splitrule</code>) for survival.
<code>write.forest</code>	Save <code>ranger.forest</code> object, required for prediction. Set to <code>FALSE</code> to reduce memory usage if no prediction intended.
<code>probability</code>	Grow a probability forest as in Malfay et al. (2012).
<code>min.node.size</code>	Minimal node size. Default 1 for classification, 5 for regression, 3 for survival, and 10 for probability.
<code>max.depth</code>	Maximal tree depth. A value of <code>NULL</code> or 0 (the default) corresponds to unlimited depth, 1 to tree stumps (1 split per tree).
<code>replace</code>	Sample with replacement.
<code>sample.fraction</code>	Fraction of observations to sample. Default is 1 for sampling with replacement and 0.632 for sampling without replacement. For classification, this can be a vector of class-specific values.
<code>case.weights</code>	Weights for sampling of training observations. Observations with larger weights will be selected with higher probability in the bootstrap (or subsampled) samples for the trees.
<code>class.weights</code>	Weights for the outcome classes (in order of the factor levels) in the splitting rule (cost sensitive learning). Classification and probability prediction only. For classification the <code>weights</code> are also applied in the majority vote in terminal nodes.
<code>splitrule</code>	Splitting rule. For classification and probability estimation "gini", "extratrees" or "hollinger" with default "gini". For regression "variance", "extratrees", "maxstat" or "beta" with default "variance". For survival "logrank", "C" or "maxstat" with default "logrank".
<code>num.random.splits</code>	For "extratrees" splitrule: Number of random splits to consider for each candidate splitting variable.
<code>alpha</code>	For "maxstat" splitrule: Significance threshold to allow splitting.
<code>minprop</code>	For "maxstat" splitrule: Lower quantile of covariate distribution to be considered for splitting.
<code>split.select.weights</code>	Numeric vector with weights between 0 and 1, representing the probability to select variables for splitting. Alternatively, a list of size <code>num.trees</code> , containing split select weight vectors for each tree can be used.
<code>always.split.variables</code>	Character vector with variable names to be always selected in addition to the <code>mtry</code> variables tried for splitting.
<code>respect.unordered.factors</code>	Handling of unordered factor covariates. One of 'ignore', 'order' and 'partition'. For the "extratrees" splitrule the default is 'partition' for all other splitrules 'ignore'. Alternatively TRUE (=order) or FALSE (=ignore) can be used. See below for details.
<code>scale.permutation.importance</code>	Scale permutation importance by standard error as in (Breiman 2001). Only applicable if <code>importance</code> is set to 'permutation'.
<code>local.importance</code>	Calculate and return local importance values as in (Breiman 2001). Only applicable if <code>importance</code> is set to 'permutation'.
<code>regularization.factor</code>	Regularization factor (gain penalization), either a vector of length <code>p</code> or one value for all variables.
<code>regularization.useddepth</code>	Consider the depth in regularization.
<code>keep.inbag</code>	Show how often observations are in-bag in each tree.
<code>inbag</code>	Manually set observations per tree. List of size <code>num.trees</code> , containing inbag counts for each observation. Can be used for stratified sampling.
<code>holdout</code>	Hold-out mode. Hold-out all samples with case weight 0 and use these for variable importance and prediction error.
<code>quantreg</code>	Prepare quantile prediction as in quantile regression forests (Menshawen 2006). Regression only. Set keep.inbag = TRUE to prepare out-of-bag quantile prediction.
<code>oob.error</code>	Compute OOB prediction error. Set to <code>FALSE</code> to save computation time, e.g. for large survival forests.
<code>num.threads</code>	Number of threads. Default is number of CPUs available.
<code>save.memory</code>	Use memory saving (but slower) splitting mode. No effect for survival and GWAS data. Warning: This option slows down the tree growing, use only if you encounter memory problems.
<code>verbose</code>	Show computation status and estimated runtime.
<code>seed</code>	Random seed. Default is <code>NULL</code> , which generates the seed from <code>pi</code> . Set to 0 to ignore the <code>it</code> seed.
<code>dependent.variable.name</code>	Name of dependent variable, needed if no formula given. For survival forests this is the time variable.
<code>status.variable.name</code>	Name of status variable, only applicable to survival data and needed if no formula given. Use 1 for event and 0 for censoring.
<code>classification</code>	Set to <code>TRUE</code> to grow a classification forest. Only needed if the data is a matrix or the response numeric.
<code>x</code>	Predictor data (independent variables), alternative interface to data with formula or <code>dependent.variable.name</code> .
<code>y</code>	Response vector (dependent variable), alternative interface to data with formula or <code>dependent.variable.name</code> . For survival use a <code>Surv()</code> object or a matrix with time and status.
<code>Details</code>	

Ranger



Value

Object of class `ranger` with elements

<code>forest</code>	Saved forest (If <code>write.forest</code> set to TRUE). Note that the variable IDs in the <code>split.varIDs</code> object do not necessarily represent the column number in R.
<code>predictions</code>	Predicted classes/values, based on out of bag samples (classification and regression only).
<code>variable.importance</code>	Variable importance for each independent variable.
<code>variable.importance.local</code>	Variable importance for each independent variable and each sample, if <code>local.importance</code> is set to TRUE and <code>importance</code> is set to 'permutation'.
<code>prediction.error</code>	Overall out of bag prediction error. For classification this is the fraction of misclassified samples, for probability estimation the Brier score, for regression the mean squared error and for survival one minus Harrell's C-index.
<code>r.squared</code>	R squared. Also called explained variance or coefficient of determination (regression only). Computed on out of bag data.
<code>confusion.matrix</code>	Contingency table for classes and predictions based on out of bag samples (classification only).
<code>unique.death.times</code>	Unique death times (survival only).
<code>chf</code>	Estimated cumulative hazard function for each sample (survival only).
<code>survival</code>	Estimated survival function for each sample (survival only).
<code>call</code>	Function call.
<code>num.trees</code>	Number of trees.
<code>num.independent.variables</code>	Number of independent variables.
<code>ntry</code>	Value of mtry used.
<code>min.node.size</code>	Value of minimal node size used.
<code>tree.type</code>	Type of forest/tree, classification, regression or survival.
<code>importance.mode</code>	Importance mode used.
<code>num.samples</code>	Number of samples.
<code>inbag.counts</code>	Number of times the observations are in-bag in the trees.

Ejemplo: California Housing data

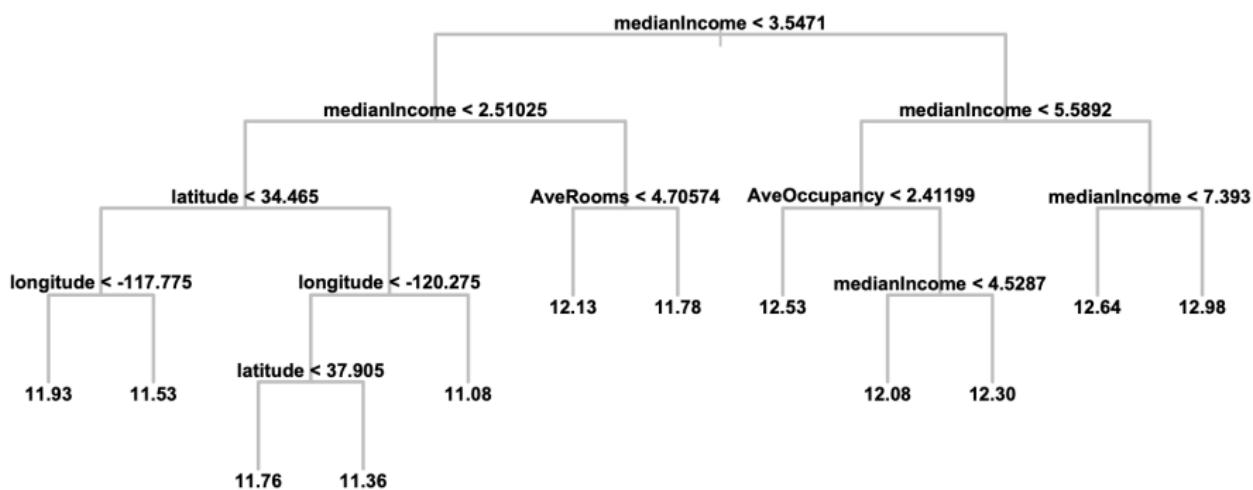
Tenemos una base de precios de casas en California que contiene, por código censal:

- ▶ Precios medianos de las casas
- ▶ Población e ingreso
- ▶ Características del hogar

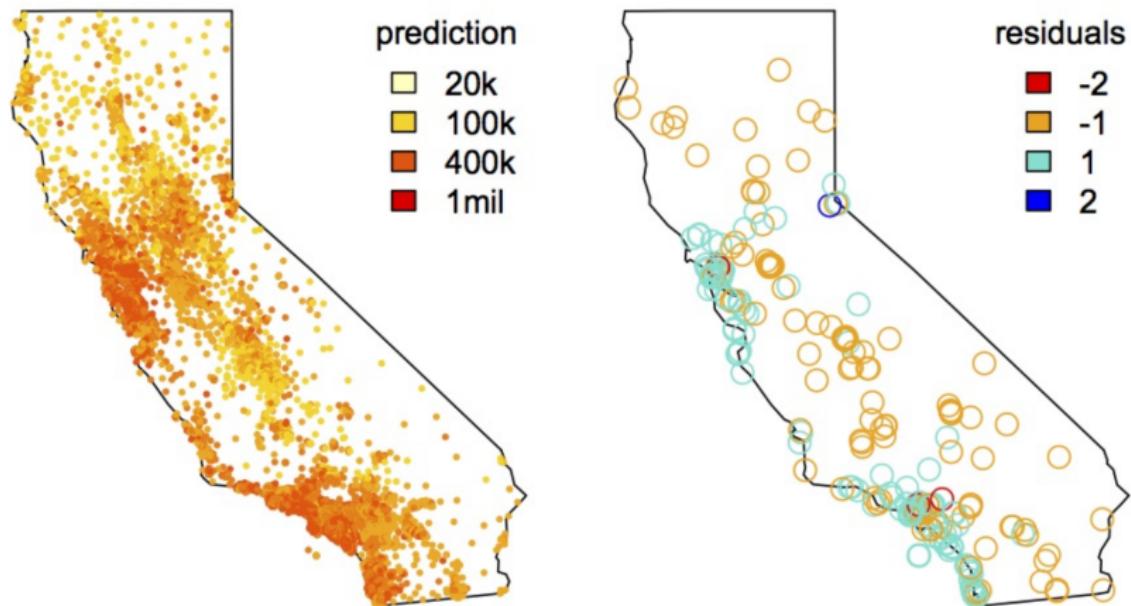
Queremos predecir el precio para cada código censal
1) LASOO 2) Prune Tree 3) Random Forest

```
carf<-ranger(logMedVal~., data= CAhousing,  
                write.forest = T, num.trees = 200,  
                min.node.size = 25,  
                importance = 'impurity')
```

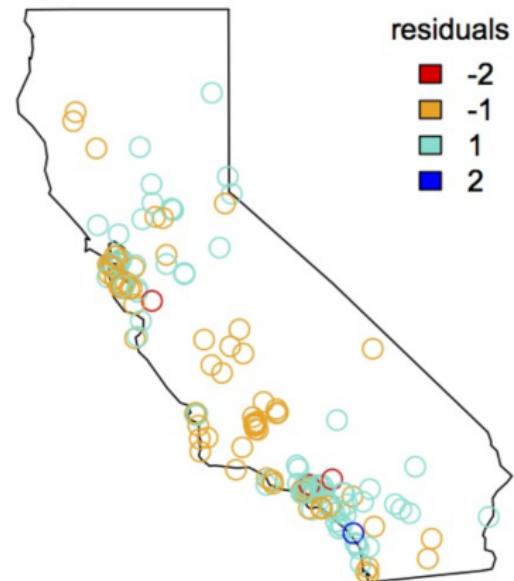
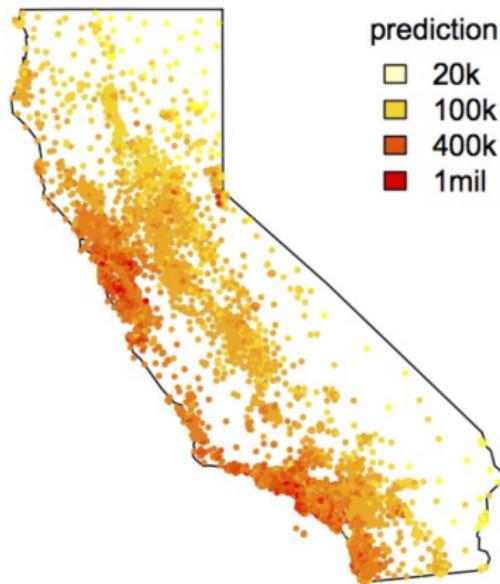
Ejemplo: California Housing data



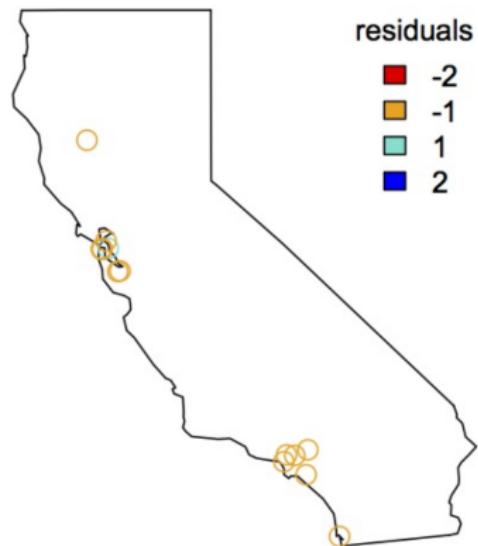
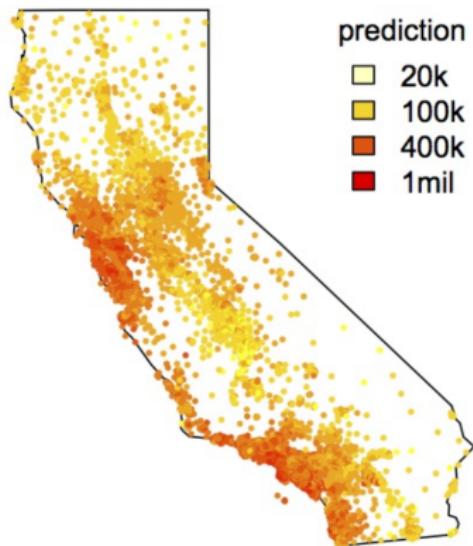
Fit Arbol



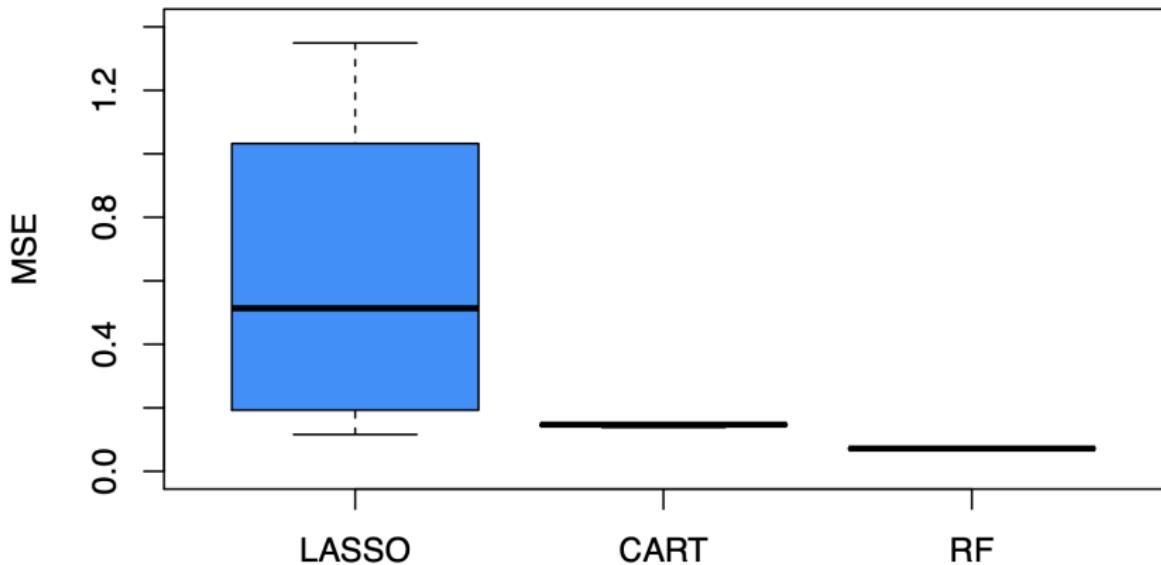
Fit LASSO



Fit Random Forest



Comparativa



Boosting

El Boosting es una de las ideas más poderosas para crear buenos modelos de predicción. Al igual que los random forests, los boosting machines combinan muchos “weak” learners que funcionan como un comité.

Sin embargo, hay diferencias gigantes en cómo funciona el Boosting.
Intuitivamente:

- ▶ En el boosting, el comité se agrupa de **super weak** learner (incluso stumps), mientras que RF agrupa árboles más grandes e independientes.
- ▶ En el boosting, los learners son construidos de manera secuencial; en el RF, de manera paralela.
- ▶ Al estimarse de manera secuencial, los Boosting algorithms **siempre** toman el último learner y lo usan para poner **enfoque** en lo que le falta aprender al algoritmo. Esto es como un proceso que va aprendiendo que es en lo que es malo automáticamente para corregirlo!

Boosting

- ▶ En cada nueva iteración, puede agregarse bagging para que el algoritmo observe nuevos puntos
- ▶ Finalmente, muchos boosting algoritmos incluyen un **peso** o **learning rate** cuando agregan los weak learners. Esto funciona como un regulizador!

En suma, un boosting algorithm estima mucho weak learners y los agrega. En cada learner, pone enfoque en los errores. Así, el algoritmo mejora donde más hace falta. Finalmente asigna un peso a cada learner en ese comité final.



ADABoost

Tomemos el ADABoost como ejemplo:

- ▶ Construye un weak learner $G_m(x)$, para $m \in \{1, \dots, M\}$. En él, le asigna a cada punto (y_i, X_i) un peso $w_i = \frac{1}{N}$
- ▶ Estima el error de ese weak learner

$$err = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}$$

- ▶ Le asigna un “peso” (votos) a $G_m(x)$ en el comité final α_m por G_m :

$$\alpha_m = \log((1 - err)/err)$$

ADABOOST

Noten como un weak learner muy bueno $err \rightarrow 0$, $\alpha_m \rightarrow \infty$. Por otro lado, si $err \rightarrow 1$, $\alpha_m \rightarrow 0$.

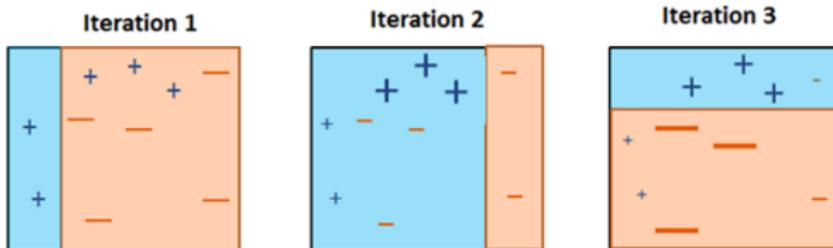
- ▶ Modifica los pesos de las observaciones de acuerdo al acierto: a las observaciones que el modelo atinó, $\downarrow w_i$; para observaciones donde el modelo falló $\uparrow w_i$.

$$w_i \leftarrow w_i * e^{\alpha_m * I(y_i \neq G_m(x_i))}$$

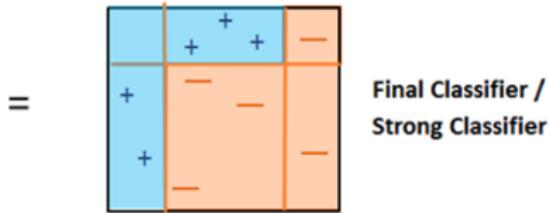
Noten como $\partial w_i / \partial [I(y_i \neq G_m(x_i))] > 0$. Es decir, el peso sube si no le atine en esa observación.

- ▶ Estima el siguiente $G_m(x)$ aplicando los nuevos pesos.
- ▶ Genera una estimación final con base en todos los learners:
$$G(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

ADABOOST



$$H = \text{sign} (0.38 x_1 + 0.58 x_2 + 0.87 x_3)$$



Gradient Boosting

Vayamos al Gradient Boosting. Este difiere un poco del ADABoost en cómo se enfoca en los errores. Sin embargo, la idea es muy parecida:

1 Empieza con un first learner

2 Para cada learner M :

- ▶ Estima el siguiente learner usando los **errores** como variable objetivo.
De aquí su nombre **gradient**
- ▶ Estima el nuevo learner γ con base en las regiones marcadas por el nuevo learner

3 Actualiza el modelo agregado con el aprendizaje

Gradient Boosting: Loss function

Empieza encontrando una Loss function por tipo de problema que sea diferenciable:

Setting	Loss function	$\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$0.5 * [y_i - f(x_i)]^2$	$y_i - f(x_i)$
Classification	Deviance	Kth component: $I(y_i = C_k) - p_{-k}(x_i)$

Gradient Boosting: Algoritmo

1. Empiezas con un algoritmo y estimación base

$$f_0(x_i) = \operatorname{argmin}_\gamma \sum_{i=1}^N L(y_i, \gamma)$$

- ▶ Típicamente γ será el promedio de las observaciones para regresión o la moda para clasificación. $f(\cdot)$ normalmente será un árbol; pero puede ser otro estimador $m \in \{1, m\}$
- 2. Para cada secuencia $m = 1, 2, \dots, M$
 - a) Para cada observación, estima el gradiente r_i (los residuales!) usando el algoritmo de la iteración anterior f_{m-1} :

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

Gradient Boosting: Algoritmo

- b) Entrena un árbol con los residuales r_{im} como variable objetivo. Esto arroja las regiones R_{jm}
- c) En cada región estima γ_{jm} : El promedio de las observaciones r_{im} (o moda en clasificación)
- d) Actualiza $f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^{Jm} \gamma_{jm} I(x \in R_{jm})$

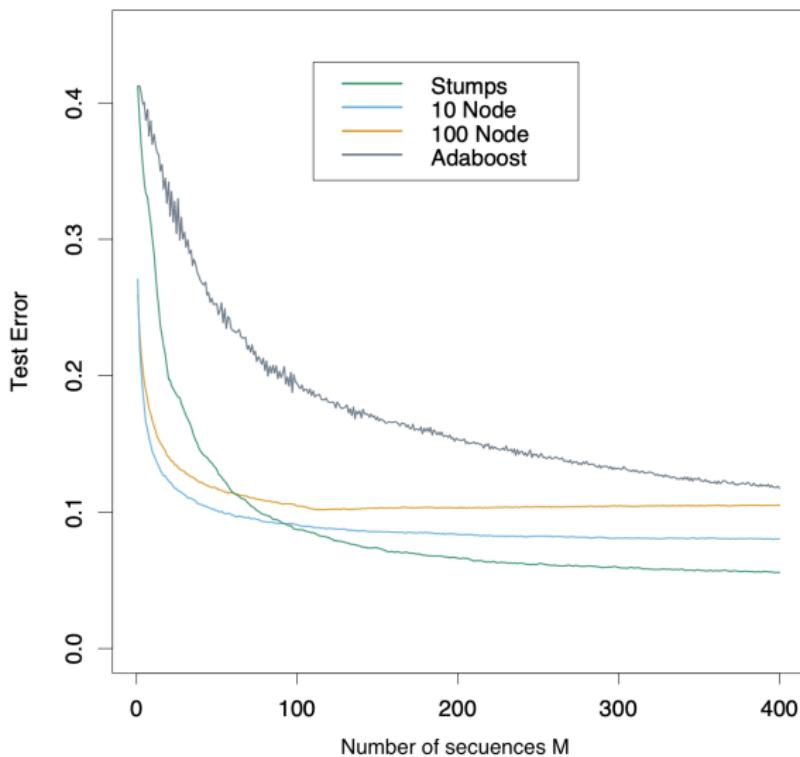
De vuelta: construyes un grupo de muchos modelos muy simples, haces que cada modelo tome los errores del anterior y se enfoque en eso. Finalmente, juntas todos los modelos, ponderándolos por un λ , que regulariza el aprendizaje.

Gradient Boosting Machines

Parámetros a tunear:

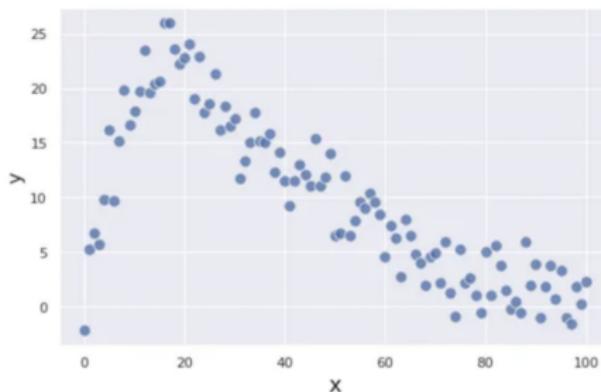
- ▶ Número de árboles M : A diferencia de bagging y RF, si M es muy grande, caeremos en overfitting. Usamos cross-validation para obtener B
- ▶ λ learning rate: Este controla (regulariza) al ritmo al que el algoritmo lee. Baja el peso de cada nuevo árbol. Se interpreta como “escepticismo”. Típicamente tiene valores de 0.01, 0.001.
- ▶ d Número de splits en cada árbol. Típicamente $d = 1$
- ▶ γ la función de aprendizaje

Qué tan grandes deben ser los árboles?



Ejemplo

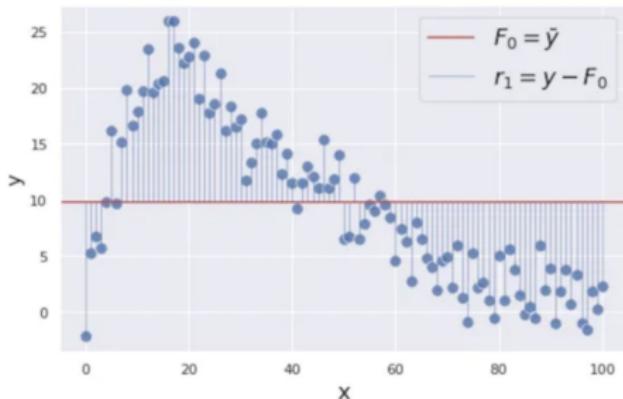
Tomemos un ejemplo de Towards Data Science para clarificar GBM.
Imaginen que tenemos una base de datos con sólo y y 1 x . Queremos construir un modelo predictivo de $\hat{y} = f(x)$.



Ahora, el paso 1 define un $f_0 = \bar{y}$. Empezamos con un predictor sencillo!
Noten como, formalmente, esto es $f_0(x_i) = \operatorname{argmin}_\gamma \sum_{i=1}^N L(y_i, \gamma)$

Ejemplo

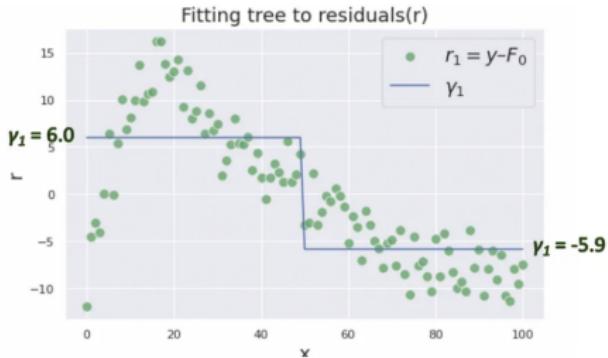
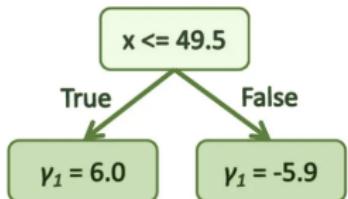
Para mejorar la predicción, nos enfocamos en los residuales $r_{i1} = y - f_0$:



Despues construimos un árbol de decisión sobre r_{im} . Donde ese árbol tiene muy pocos splits.

Ejemplo

Con el árbol, promediamos los residuales (γ_{jm}) en cada nodo terminal R_{jm} . Hacemos que el modelo se enfoque en los errores!



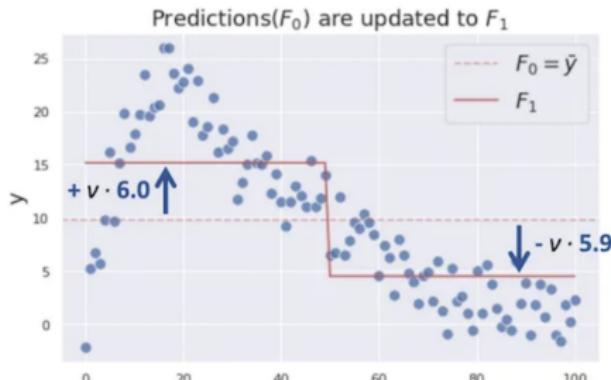
En este ejemplo tenemos $\gamma_{1m} = 6$ y $\gamma_{2m} = -5.9$!

Ejemplo

Actualiza el modelo $f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^{jm} \gamma_{jm} I(x \in R_{jm})$. En este ejemplo esto es:

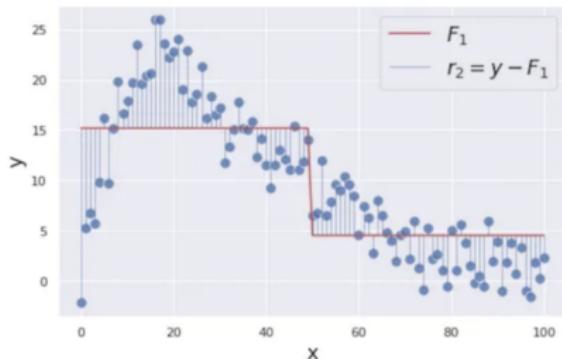
$$f_1(x) = f_0 + \lambda \gamma_{jm} I(x \in R_{jm})$$

$$f_1(x) = \begin{cases} f_0 + \lambda \cdot 6.0 & \text{if } x \leq 49.5 \\ f_0 - \lambda \cdot 5.9 & \text{if } x > 4.5 \end{cases}$$

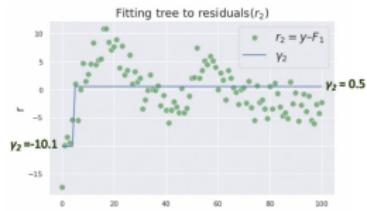
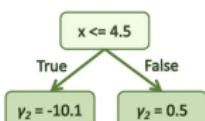


Ejemplo

Calculo los nuevos residuales r_{i2}



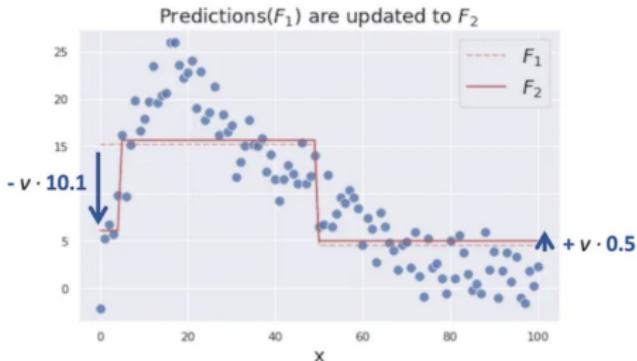
Estimo un árbol sobre r_{i2} :



Ejemplo

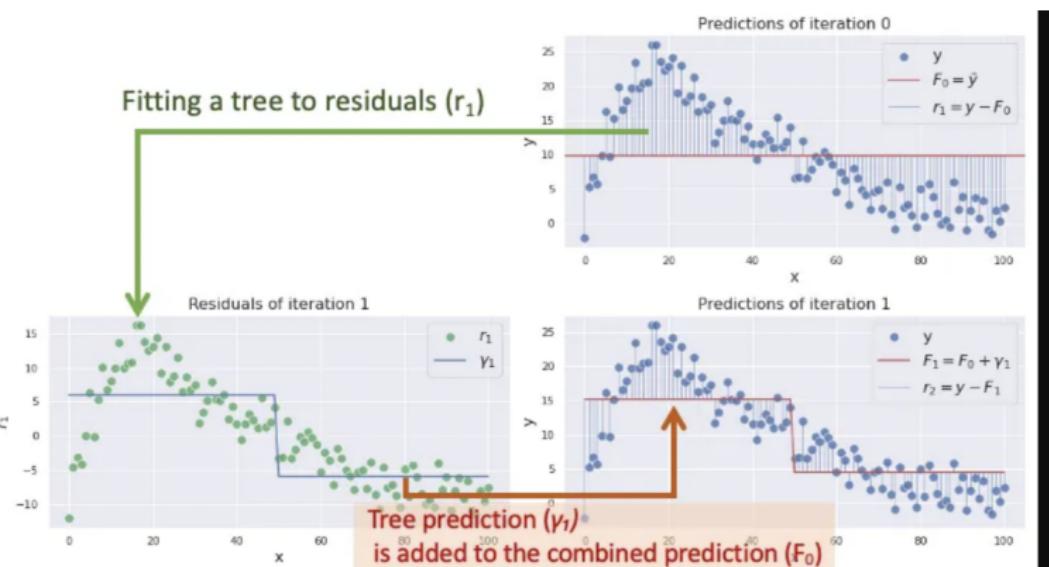
Actualizo el modelo, generando $f_2(x) = f_1(x) + \lambda \sum_{j=1}^{jm} \gamma_{j2} I(x \in R_{j2})$

$$f_2(x) = \begin{cases} f_1 - \lambda \cdot 10.1 & \text{if } x \leq 4.5 \\ f_1 + \lambda \cdot 0.5 & \text{if } x > 4.5 \end{cases}$$



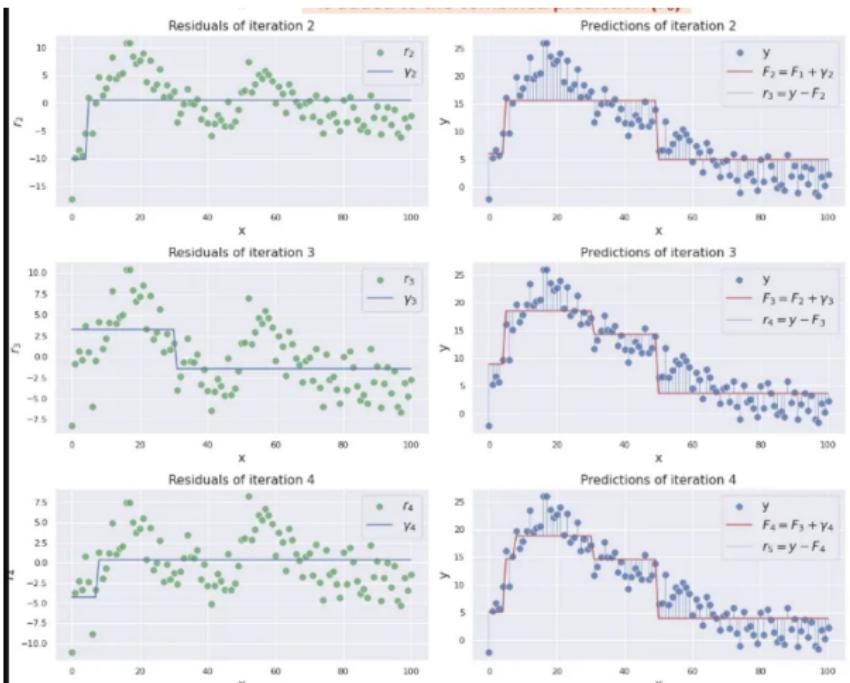
Ejemplo

Veamos como se ve esto en $m = 6$. $f_1(x)$



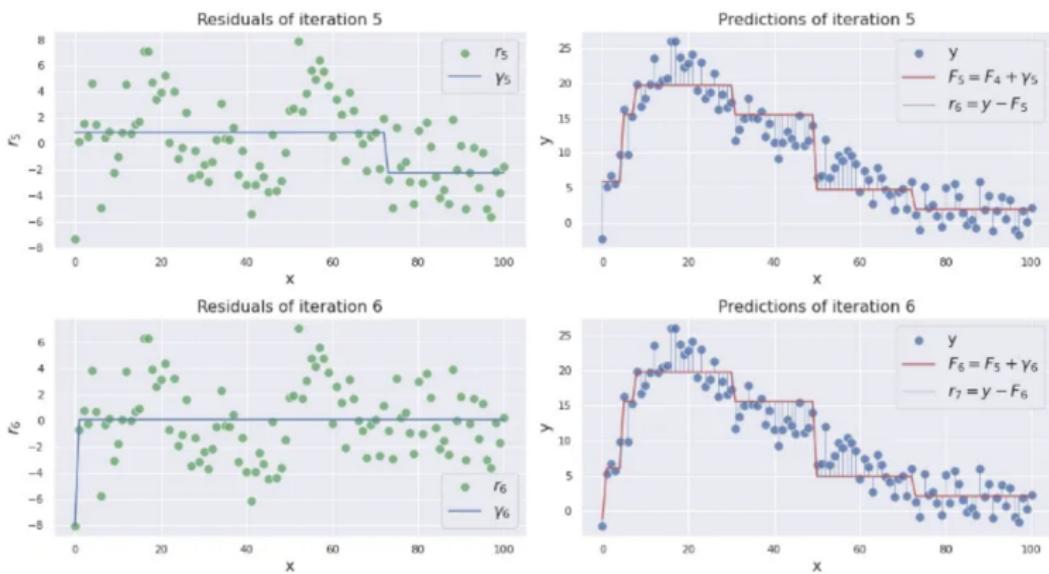
Ejemplo

Veamos como se ve esto en $f_2(x), f_3(x), f_4(x)$



Ejemplo

Veamos como se ve esto en $f_5(x), f_6(x)$



Gamma y Lambda

λ como dijimos, este valor añade algo de escepticismo $\lambda \in (0, 1)$. Si es muy grande, el algoritmo va a terminar muy rápido (bajo M) y puede seguir ruido. Típicamente tiene valores pequeños (0.01, 0.001)

Veamos como “derivar” las γ 's

$$f_0(x_i) = \operatorname{argmin}_\gamma \sum_{i=1}^N L(y_i, \gamma)$$

$$f_0(x_i) = \operatorname{argmin}_\gamma \sum_{i=1}^N 0.5(y_i - \gamma)^2$$

CPO: $\partial L / \partial \gamma$

$$\sum_{i=1}^N (y_i - \gamma^*) = 0$$

$$\sum_{i=1}^N y_i = N\gamma^*$$

$$\gamma^* = \frac{\sum_{i=1}^N y_i}{N}$$

Gamma

$$\gamma_{jm} = \operatorname{argmin}_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

$$\gamma_{jm} = \operatorname{argmin}_{\gamma} \sum_{x_i \in R_{jm}} 0.5(y_i - f_{m-1}(x_i) - \gamma)^2$$

CPO:

$$\sum_{i=1}^N (y_i - f_{m-1}(x_i) - \gamma^*) = 0$$

$$N\gamma^* = \sum_{x_i \in R_{jm}} (y_i - f_{m-1}(x_i))$$

$$\gamma_{jm}^* = \frac{\sum_{x_i \in R_{jm}} r_{im}}{n_j}$$

Porque se llama Gradient boosting y no Residual Boosting?

Si recuerdan, $r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f} \right]_{f=f_{m-1}}$. Dado que es un vector de una derivada parcial, es un gradiente.

Resulta que para un Loss-function particular $L(y_i, f(x_i)) = 0.5(y_i - f(x_i))^2$, el gradiente es exactamente lo mismo que los residuales. En realidad, se pueden implementar Gradient Boosting Machines para distintas Loss functions!!!

Ejemplo en R

La mejor librería para correr GBMs es gbm

gbm (gbm)

R Documentation

Generalized Boosted Regression Modeling (GBM)

Description

Fits generalized boosted regression models. For technical details, see the vignette: `utils::browsableVignettes("gbm")`.

Usage

```
gbm(formula = formula(data),
     distribution = "bernoulli",
     data = data,
     weights = weights,
     var.monotone = NULL,
     n.trees = 100,
     interaction.depth = 1,
     n.minobsinnode = 10,
     shrinkage = 0.1,
     bag.fraction = 0.5,
     train.fraction = 1,
     cv.folds = 5,
     keep.data = TRUE,
     verbose = FALSE,
     class.wt = NULL,
     class.stratify_cv = NULL,
     n.cores = NULL)
```

Arguments

`formula`: A symbolic description of the model to be fit. The formula may include an offset term (e.g. `y~offset(x)`). If `keep.data = FALSE` in the initial call to `gbm` then it is the user's responsibility to resupply the offset to `gbm$offset`. Otherwise, if the response is a factor, `multinomial` is assumed; otherwise, if the response has class `"Surv"`, `coxph` is assumed; otherwise, `gaussian` is assumed.

`distribution`: Either a character string specifying the name of the distribution to use or a list with a component `name` specifying the distribution and any additional parameters needed. If not specified, `gbm` will try to guess: if the response has only 2 unique values, `bernoulli` is assumed; otherwise, if the response is a factor, `multinomial` is assumed; otherwise, if the response has class `"Surv"`, `coxph` is assumed; otherwise, `gaussian` is assumed.

Currently available options are: `"gaussian"` (required entry), `"laplace"` (absolute loss), `"dlnorm"` (distribution loss), `"bernoulli"` (logistic regression for 0-1 outcomes), `"huberized"` (huberized hinge loss for 0-1 outcomes), `"classes"`, `"adaboost"` (the Adaboost exponential loss for 0-1 outcomes), `"poisson"` (count outcomes), `"cosh"` (right censored observations), `"quantile"`, or `"pairwise"` (ranking measure using the LambdaRank algorithm).

If quantile regression is specified, `distribution` must be a list of the form `list(name = "quantile", alpha = 0.25)` where `alpha` is the quantile to estimate. The current version's quantile regression method does not handle non-constant weights and will stop.

If `"dlnorm"` is specified, the default degrees of freedom is 4 and this can be controlled by specifying `distribution = list(name = "dlnorm", df = 4)` where `df` is your chosen degrees of freedom.

If `"pairwise"` regression is specified, `distribution` must be a list of the form `list(name = "pairwise", group = ..., metric = ..., max.rank = ..., ...)`. `metric` and `max.rank` are optional, see below. `group` is a character vector with the column names of `data` that jointly indicate the group an instance belongs to (typically a query in Information Retrieval applications). For training, only pairs of instances from the same group and with different target labels can be considered. `metric` is the IR measure to use, one of:

- `softDCC`: Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve
- `hardDCC`: Mean reciprocal rank of the highest-ranked positive instance
- `meanRPR`: Mean reciprocal rank of the highest-ranked positive instance
- `softMAP`: Mean average precision, a generalization of `map` to multiple positive instances
- `hardMAP`: Mean average precision, a generalization of `map` to multiple positive instances
- `map`: Normalized discounted cumulative gain. The score is the weighted sum (DCCG) of the user-supplied target values, weighted by `log(1+exp)`, and normalized to the maximum achievable value. This is the default if the user did not specify a metric.

`data` and `cvData` allow arbitrary target values, while binary targets 0,1 are expected for `map` and `arr`. For `map` and `arr`, a cut-off can be chosen using a positive integer parameter `max.rank`. If left unspecified, all ranks are taken into account.

Note that splitting of instances into training and validation sets follows group boundaries and therefore only approximates the specified `train.fraction` ratio (the same applies to cross-validation folds). Internally, queries are randomly shuffled before training, to avoid bias.

Weights can be used in conjunction with pairwise metrics, however it is assumed that they are constant for instances from the same group.

For details and background on the algorithm, see e.g. Büger (2010).

`formula`: an optional data frame containing the variables in the model. By default the variables are taken from `environment(formula)`, typically the environment from which `gbm` is called. If `keep.data=TRUE` in the initial call to `gbm` stores a copy with the object. If `keep.data=FALSE` then `environment(formula)` will remain unaltered in memory. It increases the user's memory requirements if the same frame at this point.

Ejemplo en R

La mejor librería para correr GBMs es gbm



```
gbm
date
weights
var.monotone
n.trees
interaction.depth
n.minobsinnode
shrinkage
bag.fraction
train.fraction
cv.folds
keep.data
verbose
class.stratify
n.cores
Details
gbm_fits provides the link between R and the C++ gbm engine. gbm is a front-end to gbm_fits that uses the familiar R modeling formulas. However, model.frame is very slow if there are many predictor variables. For power-users with many variables use gbm.fit. For general practice gbm is preferable.
This package implements the generalized boosted modeling framework. Boosting is the process of iteratively adding basis functions in a greedy fashion so that each additional basis function further reduces the selected loss function. This implementation closely follows Friedman's Gradient Boosting Machine (Friedman, 2001).
In addition to many of the features documented in the Gradient Boosting Machine, gbm offers additional features including the out-of-bag estimator for the optimal number of iterations, the ability to store and manipulate the resulting gbm object, and a variety of other loss functions that had not previously had associated boosting algorithms, including the Cox partial likelihood for censored data, the poisson likelihood for count outcomes, and a gradient boosting implementation to minimize the AdaBoost exponential loss function.
Value
A gbm.object object.
Author(s)
Greg Ridgeway <greg@stat.unc.edu>
Quadratic regression code developed by Brian Krieger <bk@stat.uct.ac.za>
t-distribution, and multivariate code developed by Harry Southworth and Daniel Edwards
Parwise code developed by Stefan Schröd schrod@w2i.com
References
Y. Freund and R.E. Schapire (1997) "A decision-theoretic generalization of on-line learning and an application to boosting," Journal of Computer and System Sciences, 55(1):119-139.
G. Ridgeway (1999). "The state of boosting," Computing Science and Statistics 31:172-181.
J.H. Friedman (2001). "Additive Logistic Regression: A Statistical View of Boosting," Annals of Statistics 29(2):337-374.
J.H. Friedman (2002). "Greedy Function Approximation: A Gradient Boosting Machine," Annals of Statistics 29(3):1189-1232.
J.H. Friedman (2005). "Stochastic Gradient Boosting," Computational Statistics and Data Analysis 36(4):397-378.
B. Krieger (2007). "Cost-Sensitive Stochastic Gradient Boosting Within a Quantitative Regression Framework," Ph.D. Dissertation, University of California at Los Angeles, Los Angeles, CA, USA. Advisor(s) Richard A. Berk. http://hdl.handle.net/1254603.
C. Burges (2010). "From RankNet to LambdaRank to LambdaMART: An Overview," Microsoft Research Technical Report MSR-TR-2010-82.
See Also
```

Ejemplo en R

```
set.seed(102) # for reproducibility

gbm1 <- gbm(Y ~ ., data = data, var.monotone = c(0, 0, 0, 0, 0, 0,
                                                    distribution = "gaussian", n.trees = 100,
                                                    shrinkage = 0.1, interaction.depth = 1,
                                                    bag.fraction = 0.5, train.fraction = 0.5,
                                                    n.minobsinnode = 10, keep.data = TRUE,
                                                    verbose = FALSE, n.cores = 1)
```

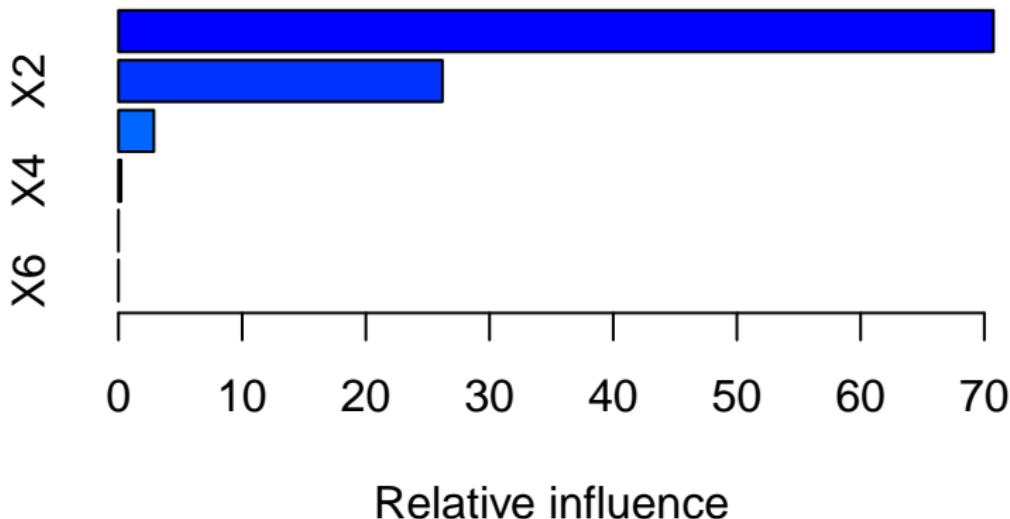
predictions will be on the link scale

```
best.ite <- gbm.perf(gbm1, method = "OOB")
```



Ejemplo R

```
summary(gbm1, n.trees = best.iter) # Variable importance
```



```
##      var      rel.inf
## X3    X3 70.7401714
## X2    X2 26.1887224
## X1    X1  2.8637156
## X4    X4  0.2073207
```

Extreme Gradient Boosting Machines (XGBoosting)

Los Extreme Gradient Boosting Machines son muy parecidos a los GBM; le agregan algunas cosas:

- ▶ El concepto de `mtry` de los RF para ortogonalizar los árboles
- ▶ El bagging (ya lo tenía GBM)
- ▶ Regularizadores de los nodos de los árboles! `alpha` `lambda`
- ▶ Es una implementación super rápida y paralelizable de GBM !

Extreme Gradient Boosting Machines (XGBoosting)

Una generalización de los Gradient Boosting Machines; se agregan más hiperparámetros:

- ▶ M nrounds: Número de árboles
- ▶ early_stopping_rounds: Resuelve la miopía y la gobierna.
Cuántas rondas pueden pasar sin que disminuya el RSS para parar.
Igual los hace más ligeros que un Random Forest.
- ▶ d max_depth: El tamaño máximo permitido por árbol.
- ▶ min_child_weight: Muy similar a mínimas observaciones en los nodos terminales del árbol.
- ▶ gamma: La ganancia mínima requerida dentro de cada árbol. Controla la profundidad y cantidad de árboles.
- ▶ λ eta: El tasa de aprendizaje [0, 1].
- ▶ subsample: La muestra sobre N que toma para estimar el árbol
- ▶ colsample_: La muestra de columnas por cada árbol (Como en RF!)

Extreme Gradient Boosting Machines (XGBoosting)

- ▶ Como ven, los XGB combinan muchos conceptos que vimos en trees, LASSO e incluso Random Forest.
- ▶ Esto los convierte en los algoritmos favoritos (por poderosos) de muchos científicos de datos.
- ▶ Tienden a ser más ligeros que los Random Forests, ya que cada árbol es muy pequeño
- ▶ la librería para estimarlos en R es `library(xgboost)`
- ▶ Típicamente se tunea estos parámetros haciendo un grid de todos ellos (que cubra mucho del espacio) y despues se observa cuál combinación fue la ganadora. Esta es la parte más complicada

Ejemplo en R

```
library(tidyverse)
library(tidymodels) # caret igual sirve
library(xgboost)

# Loading the data lists for training
load('Bases output/universo_training.Rdata')

# Creating the grid that covers most of the hyper space
xgb_grid<-grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  learn_rate(),
  size = 15)
```

Ejemplo en R

```
Xs<-sparse.model.matrix(~. + 0, data = data)
y<-data$y

# XGBOOST matrix
training<-xgb.DMatrix(data = Xs, label = y)

# Watchlist
watchlist<-list(training = training, validation = validation)
names(watchlist$validation)<- names(watchlist$training)

# transforming xgb_grid to have proper names
xgb_grid<-xgb_grid %>%
  rename(max_depth = tree_depth,
         min_child_weight = min_n,
         gamma = loss_reduction,
         subsample = sample_size,
         eta = learn_rate) %>%
  mutate(eval metric = 'auc')
```

Ejemplo en R

```
a<-Sys.time()
xgb_eth<-map(xgb_grid_list,
              function(x)
                xgb.train(params = as.list(x),
                           data = training$buy_eth,
                           nrounds = 300,
                           objective = "binary:logistic",
                           early_stopping_rounds = 5,
                           verbose = 1,
                           watchlist = c(training = watchlist$traini
                                         test = watchlist$validati
Sys.time() -a
save(xgb_eth, xgb_grid_list, training, watchlist, file = "Modelo")
```

Ejemplo en R

```
#####
# Choosing the best model for each coin
#####
write.xlsx(xgb_grid, file = 'Tablas/xgb_grid.xlsx')

# Exporting the parameters and AUC

# ETH
params_eth<-map_dfr(xgb_eth, ~.$params)
params_eth<-bind_cols(params_eth, auc = map_dbl(xgb_eth, function(x) {
  params_eth$model_number<-seq(1:15)
```

Ejemplo en R

max_depth	min_child_weight	gamma	subsample	eta	eval_metric	objective	validate_parameters	auc
10	7	5.717E-05	0.1388513759	0.015048133	auc	binary:logistic	TRUE	0.883717
2	17	0.151017646	0.163572226	1.4994E-09	auc	binary:logistic	TRUE	0.5
12	17	2.54111E-08	0.532761323	0.082454355	auc	binary:logistic	TRUE	0.884458
6	2	1.45029E-10	0.876723072	1.86427E-07	auc	binary:logistic	TRUE	0.853164
7	37	0.000699505	0.94807782	9.6865E-08	auc	binary:logistic	TRUE	0.858693
9	23	0.004955619	0.922445802	1.69647E-08	auc	binary:logistic	TRUE	0.865919
15	6	1.479444991	0.368763574	0.003911569	auc	binary:logistic	TRUE	0.882513
13	28	3.55132E-06	0.634782503	0.000456521	auc	binary:logistic	TRUE	0.877462
4	32	2.37289E-07	0.271732136	1.1008E-05	auc	binary:logistic	TRUE	0.835202
2	39	1.83201E-09	0.647606425	0.000341731	auc	binary:logistic	TRUE	0.761712
11	34	0.462088589	0.51822083	1.99108E-09	auc	binary:logistic	TRUE	0.5
5	14	4.78266E-09	0.773989784	2.41154E-06	auc	binary:logistic	TRUE	0.84666
11	10	1.6519E-05	0.316428185	5.87259E-07	auc	binary:logistic	TRUE	0.876701
8	21	5.456894471	0.724965436	2.30408E-10	auc	binary:logistic	TRUE	0.5
4	25	0.000804574	0.425232621	8.8488E-05	auc	binary:logistic	TRUE	0.845176