Boosting

En el boosting los árboles se hacen crecer ***secuencialmente.*** Cada árbol utiliza la información **entregada, *y* *procesada,*** por el árbol anterior. A diferencia del *bagging* y de *random Forest* los árboles entrenados con *boosting* no utilizan muestras obtenidas por *bootstrap,* sino que cada árbol se entrena con una **versión modificada** del dataset original.

El algoritmo Boosting aprende ***lentamente****. **Cada árbol se entrena con los residuos del anterior**, o sea, con los errores del otro.* O sea que en vez de evaluar una función de error con los datos observados ***y*** utilizamos los ***residuos*** del árbol anterior $(y - \hat{y})$, enfocándonos especialmente en las observaciones en las que el árbol anterior **predijo mal**.
Este método se enfoca en las observaciones que son difíciles de predecir correctamente y va encadenando clasificadores simples para al final, ponderándolos, obtener un modelo ensamblado.
Cada uno de estos modelos simples se llama **clasificadores débiles** o "**weak learners**". Estos clasificadores débiles garantizan la obtención de un resultado de clasificación binaria mejor que el de arrojar una moneda al aire, que es la definición máxima de incertidumbre en este tipo de experimentos binomiales.

Algoritmo AdaBoost *(Adaptative Boosting)*

Freund and Schapire (1997)

Y ∈ {−1, 1} $\epsilon_t \doteq \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$

Generaremos sucesivos modelos h_1, h_2, …., h_t para luego "votar " en un formato de "comité" y obtener la predicción final:

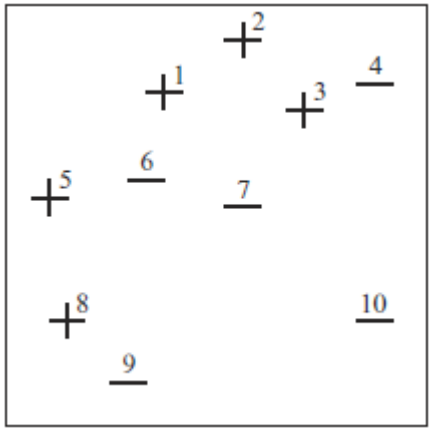
$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Los α_t serán calculados por el algoritmo y su valor dependerá de la capacidad discriminante del clasificador débil correspondiente.
Los pesos iniciales son $D_1(i) = 1/m$
Para cada iteración (o clasificación débil) $t = 1, \ldots, T$ se recalculan los pesos en función del resultado de la clasificación$_t$, incrementando su valor si ésta fue errónea, y reduciéndolo si fue correcta.

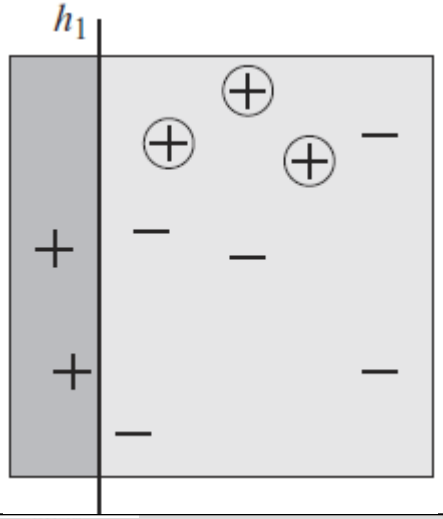
Analicemos el algoritmo de Boosting gráficamente

IMPORTANTE: En boosting las clases son **-1 y 1**

Dataset original D_1

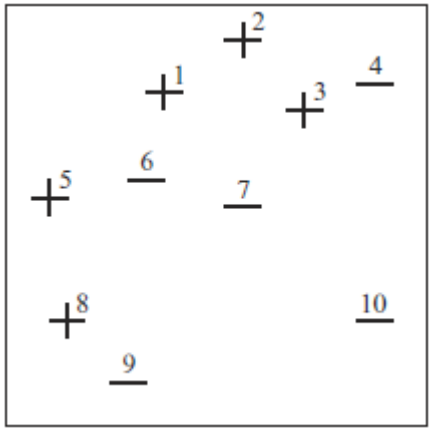


1º Clasificador

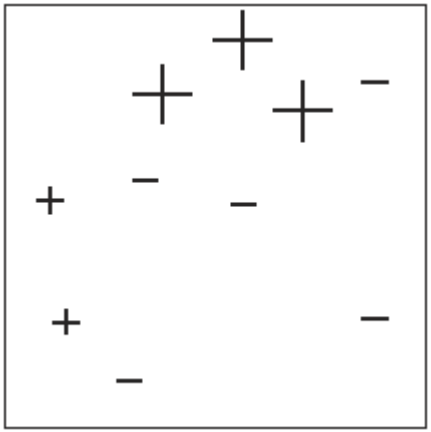


Ahora le asignaremos a cada observación un PESO en función a cómo la clasificó el "weak learner" (mayor a los que clasificó MAL, menor a los que clasificó BIEN)

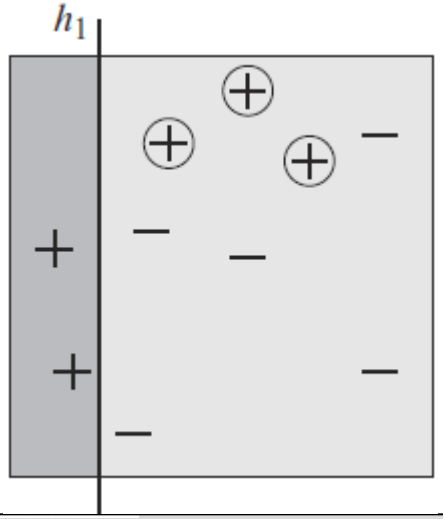
Dataset original **D_1**



Dataset Modificado (pesos) **D_2**

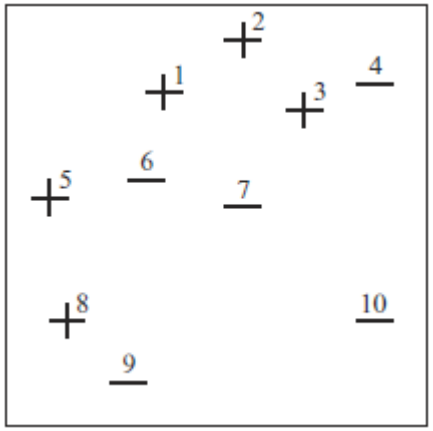


1º Clasificador

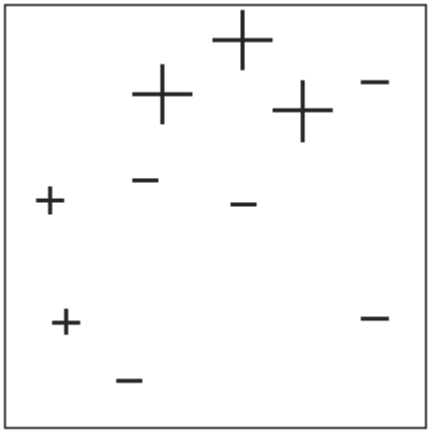


Obtengo es este dataset modificado de pesos D_2
Aumentando el peso de las instancias en las que predijo mal y reduciéndolo en las que predijo bien. Ahora entreno un nuevo clasificador débil *(weak learner)* con este nuevo dataset **D_2**

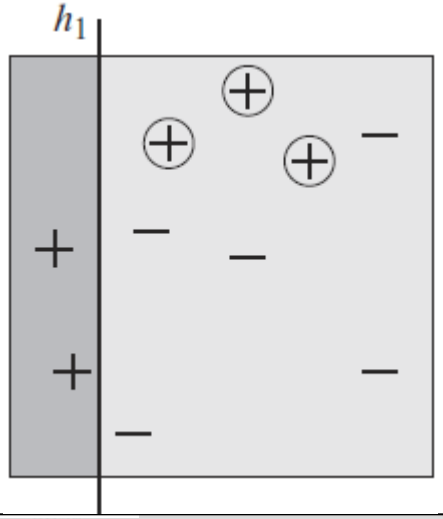
Dataset original $\mathbf{D_1}$



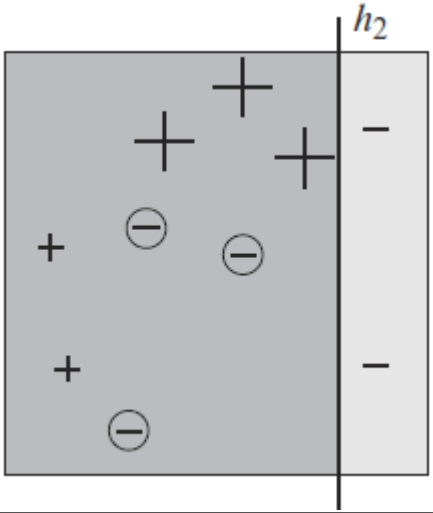
Dataset Modificado (pesos) $\mathbf{D_2}$



1º Clasificador

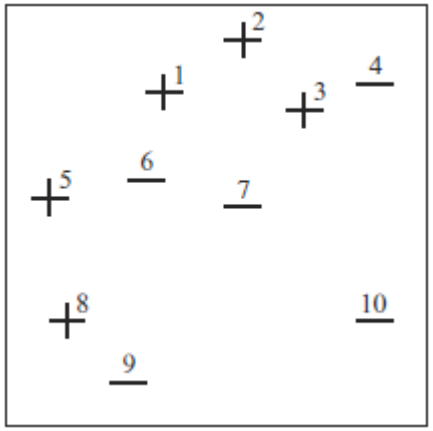


2º Clasificador

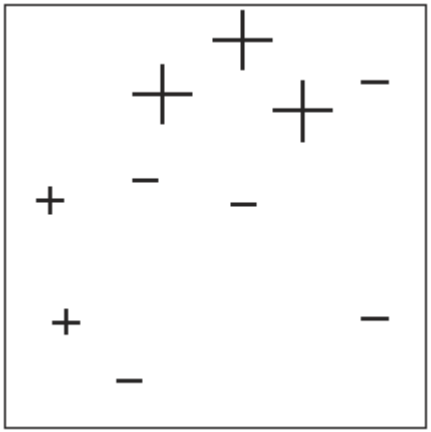


Identificamos los que clasificó mal y generamos un nuevo Dataset Modificado $\mathbf{D_3}$, en el que nuevamente incrementamos los pesos de las que predijo mal y reducimos las que predijo bien

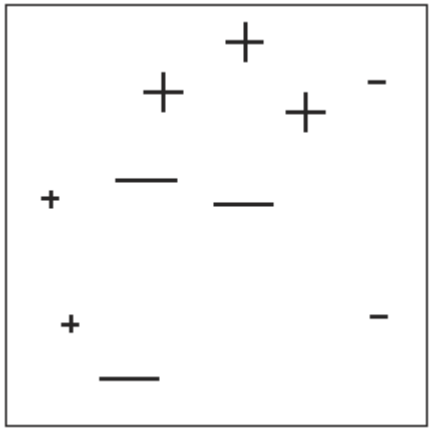
Dataset original $\mathbf{D_1}$



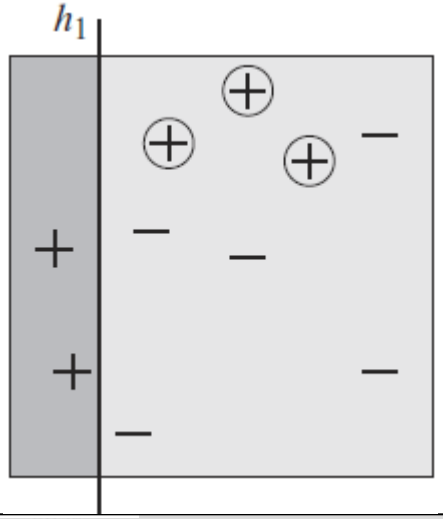
Dataset Modificado (pesos) $\mathbf{D_2}$



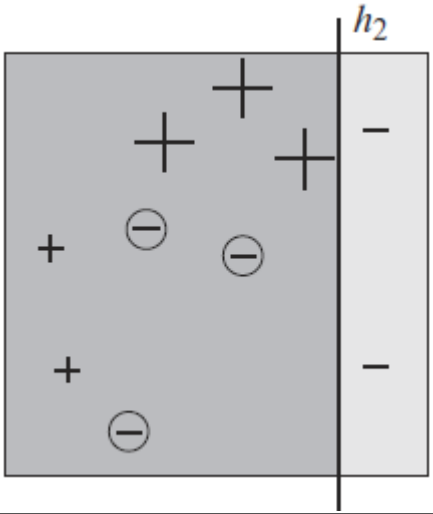
Dataset Modificado (pesos) $\mathbf{D_3}$



1º Clasificador

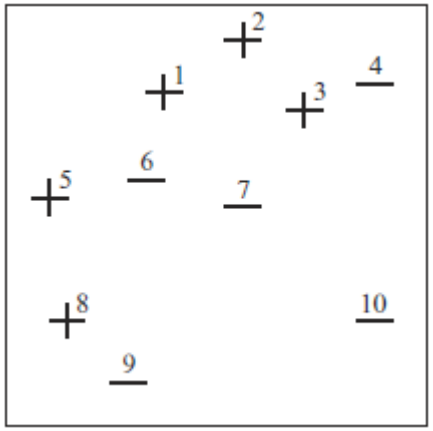


2º Clasificador

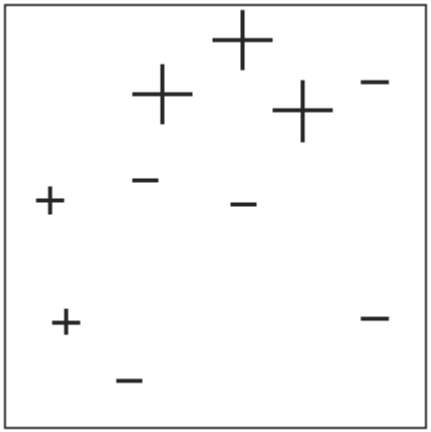


Y ahora entrenaremos un tercer clasificador débil sobre este dataset D3, que será:

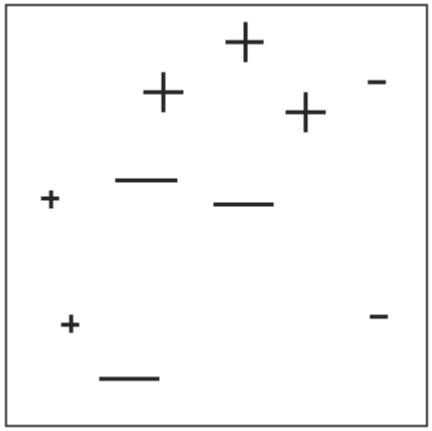
Dataset original $\mathbf{D_1}$



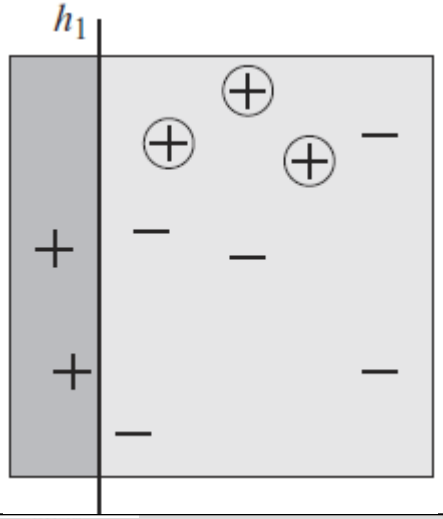
Dataset Modificado (pesos) $\mathbf{D_2}$



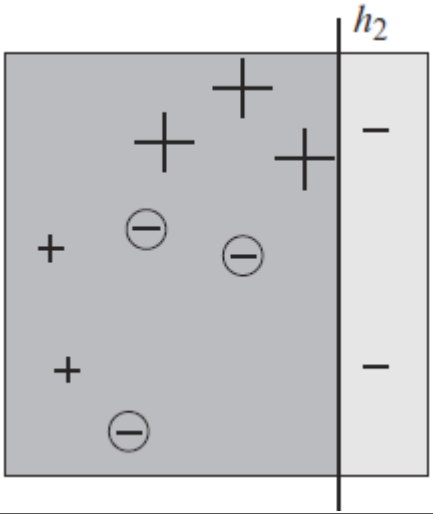
Dataset Modificado (pesos) $\mathbf{D_3}$



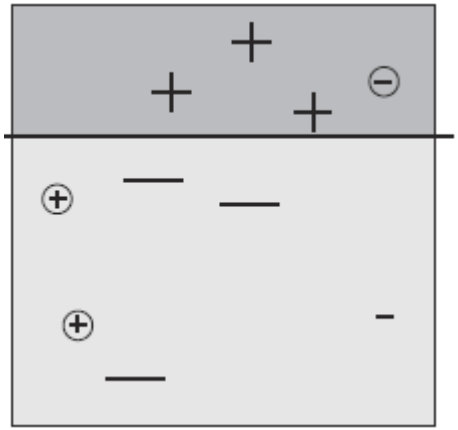
1º Clasificador



2º Clasificador



3º Clasificador



Los pasos y las fórmulas que utiliza el algoritmo AdaBoost son:
Dados $(x_1, y_1), \ldots, (x_m, y_m)$ donde $x_i \in \mathcal{X}, y_i \in \{-1, +1\}$

1) **Inicializar los pesos** $D_i = 1/m$ para i = 1, 2, . . . ,*m*

2) **De *t* = 1 a *T* repetir {**

   i. Entrenar un clas. débil sobre los pesos D_t y calcular hip. débil $h_t : \mathcal{X} \to \{-1, +1\}$

   ii. Objetivo: *Seleccionar* h_t *que minimice el error ponderado* $\epsilon_t \doteq \mathbf{Pr}_{i \sim D_t}[h_t(x_i) \neq y_i]$

   iii. Calcular $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$

   iv. Actualizar, para *i = 1, 2…. , m*

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases}$$

$$= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

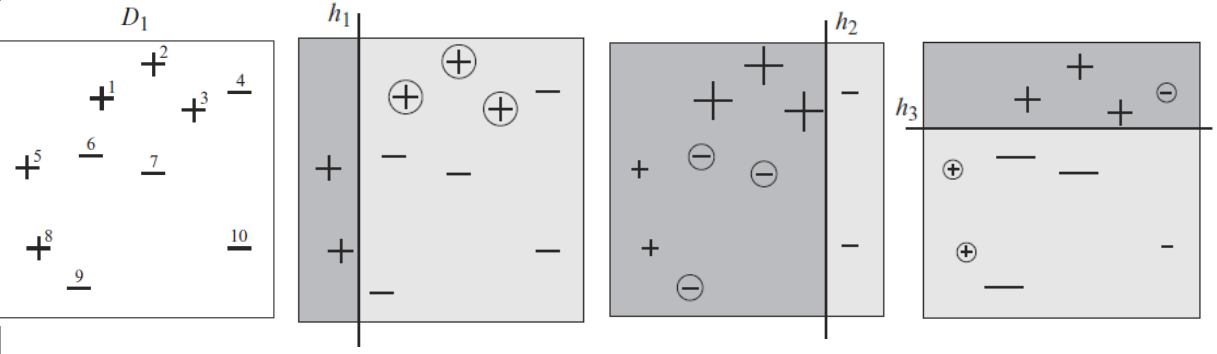
Donde Z_t es un factor de normalización
(elegido para que D_{t+1} sea una distribución)

}

3) **Clasificar a la observación *x* como:**

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Veamos un EJEMPLO



$$D_1(i) = 1/m$$

$$\epsilon_t \doteq \mathbf{Pr}_{i\sim D_t}[h_t(x_i) \neq y_i]$$

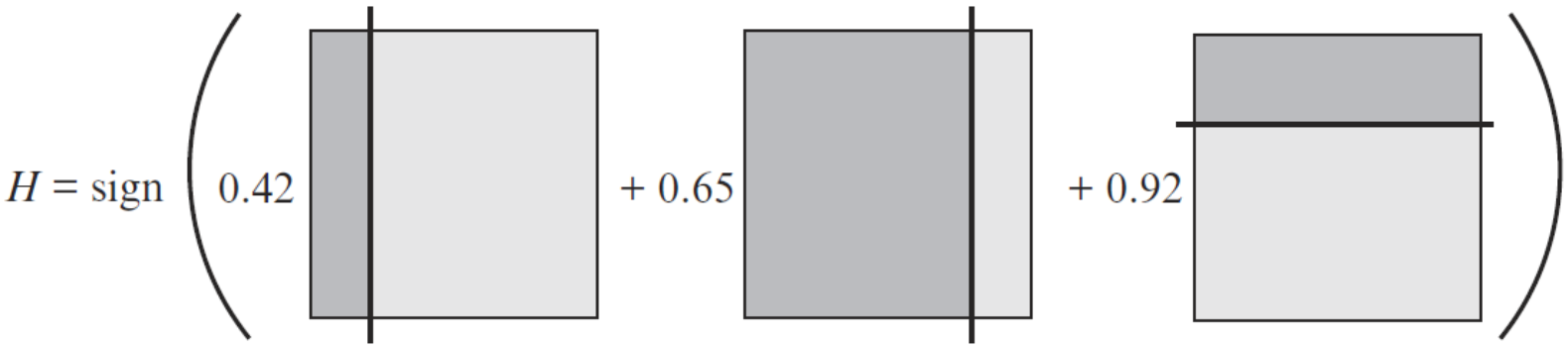
$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

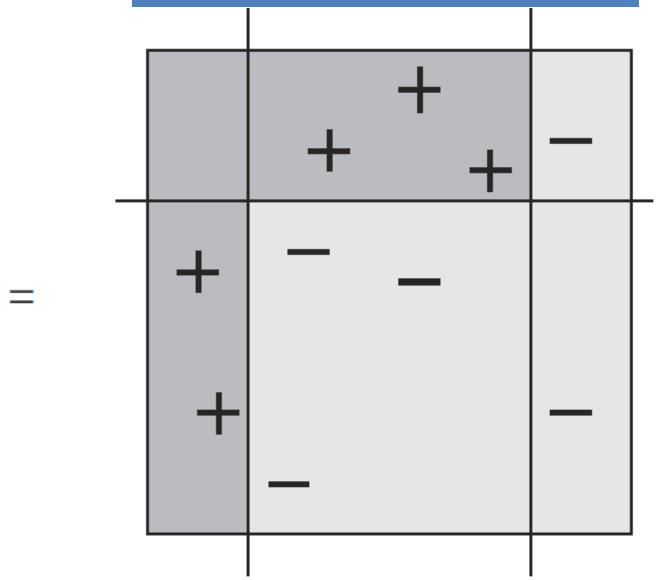
	1	2	3	4	5	6	7	8	9	10	
$D_1(i)$	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	$\epsilon_1 = 0.30$, $\alpha_1 \approx 0.42$
$e^{-\alpha_1 y_i h_1(x_i)}$	1.53	1.53	1.53	0.65	0.65	0.65	0.65	0.65	0.65	0.65	
$D_1(i)\ e^{-\alpha_1 y_i h_1(x_i)}$	0.15	0.15	0.15	0.07	0.07	0.07	0.07	0.07	0.07	0.07	$Z_1 \approx 0.92$
$D_2(i)$	0.17	0.17	0.17	0.07	0.07	0.07	0.07	0.07	0.07	0.07	$\epsilon_2 \approx 0.21$, $\alpha_2 \approx 0.65$
$e^{-\alpha_2 y_i h_2(x_i)}$	0.52	0.52	0.52	0.52	0.52	1.91	1.91	0.52	1.91	0.52	
$D_2(i)\ e^{-\alpha_2 y_i h_2(x_i)}$	0.09	0.09	0.09	0.04	0.04	0.14	0.14	0.04	0.14	0.04	$Z_2 \approx 0.82$
$D_3(i)$	0.11	0.11	0.11	0.05	0.05	0.17	0.17	0.05	0.17	0.05	$\epsilon_3 \approx 0.14$, $\alpha_3 \approx 0.92$
$e^{-\alpha_3 y_i h_3(x_i)}$	0.40	0.40	0.40	2.52	2.52	0.40	0.40	2.52	0.40	0.40	
$D_3(i)\ e^{-\alpha_3 y_i h_3(x_i)}$	0.04	0.04	0.04	0.11	0.11	0.07	0.07	0.11	0.07	0.02	$Z_3 \approx 0.69$

Los ejemplos en los el clasificador débil comete error es se indican subrayados

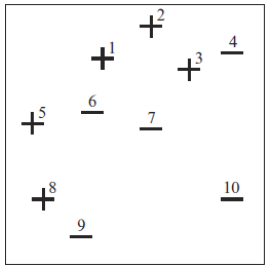
Ponderamos cada clasificador débil, los sumamos y en función del signo clasificamos:



Clasificador combinado



Ejemplo, obs #4



$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

,42 ,65 ,92

$$\text{sign}(-\alpha_1 - \alpha_2 + \alpha_3) = \text{sign}(-0.15) = -1$$

-1 -1 1

Boosting en lenguaje R – Librería GBM *(Generalized Boosting Models)*

gbm *Generalized Boosted Regression Modeling (GBM)*

Description

Fits generalized boosted regression models. For technical details, see the vignette: `utils::browseVig`

Usage

```
gbm(formula = formula(data), distribution = "bernoulli",
  data = list(), 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 = 0,
  keep.data = TRUE, verbose = FALSE, 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(n)+x). If `keep.data = FALSE` in the initial call to `gbm` then it is the user's responsibility to resupply the offset to `gbm.more`.

`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"` (squared error), `"laplace"` (absolute loss), `"tdist"` (t-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), `"coxph"` (right censored observations), `"quantile"`, or `"pairwise"` (ranking measure using the LambdaMart 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 `"tdist"` is specified, the default degrees of freedom is 4 and this can be controlled by specifying `distribution = list(name = "tdist", df = DF)` 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

list("conc") Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

: Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

list("mrr") Mean reciprocal rank of the highest-ranked positive instance

: Mean reciprocal rank of the highest-ranked positive instance

list("map") Mean average precision, a generalization of `mrr` to multiple positive instances

: Mean average precision, a generalization of `mrr` to multiple positive instances

list("ndcg:") Normalized discounted cumulative gain. The score is the weighted sum (DCG) of the user-supplied target values, weighted by log(rank+1), and normalized to the maximum achievable value. This is the default if the user did not specify a metric.

`ndcg` and `conc` allow arbitrary target values, while binary targets 0,1 are expected for `map` and `mrr`. For `ndcg` and `mrr`, 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. Burges (2010).

`data` 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` then `gbm` stores a copy with the object. If `keep.data=FALSE` then subsequent calls to `gbm.more` must resupply the same dataset. It becomes the user's responsibility to resupply the same data at this point.

`weights` an optional vector of weights to be used in the fitting process. Must be positive but do not need to be normalized. If `keep.data=FALSE` in the initial call to `gbm` then it is the user's responsibility to resupply the weights to `gbm.more`.

`var.monotone` an optional vector, the same length as the number of predictors, indicating which variables have a monotone increasing (+1), decreasing (-1), or arbitrary (0) relationship with the outcome.

`n.trees` Integer specifying the total number of trees to fit. This is equivalent to the number of iterations and the number of basis functions in the additive expansion. Default is 100.

`interaction.depth` Integer specifying the maximum depth of each tree (i.e., the highest level of variable interactions allowed). A value of 1 implies an additive model, a value of 2 implies a model with up to 2-way interactions, etc. Default is 1.

`n.minobsinnode` Integer specifying the minimum number of observations in the terminal nodes of the trees. Note that this is the actual number of observations, not the total weight.

`shrinkage` a shrinkage parameter applied to each tree in the expansion. Also known as the learning rate or step-size reduction; 0.001 to 0.1 usually work, but a smaller learning rate typically requires more trees. Default is 0.1.

`bag.fraction` the fraction of the training set observations randomly selected to propose the next tree in the expansion. This introduces randomnesses into the model fit. If `bag.fraction` < 1 then running the same model twice will result in similar but different fits. `gbm` uses the R random number generator so `set.seed` can ensure that the model can be reconstructed. Preferably, the user can save the returned `gbm.object` using `save`. Default is 0.5.

`train.fraction` The first `train.fraction * nrows(data)` observations are used to fit the `gbm` and the remainder are used for computing out-of-sample estimates of the loss function.

`cv.folds` Number of cross-validation folds to perform. If `cv.folds`>1 then `gbm`, in addition to the usual fit, will perform a cross-validation, calculate an estimate of generalization error returned in `cv.error`.

`keep.data` a logical variable indicating whether to keep the data and an index of the data stored with the object. Keeping the data and index makes subsequent calls to `gbm.more` faster at the cost of storing an extra copy of the dataset.

`verbose` Logical indicating whether or not to print out progress and performance indicators (`TRUE`). If this option is left unspecified for `gbm.more`, then it uses `verbose` from `object`. Default is `FALSE`.

`class.stratify.cv` Logical indicating whether or not the cross-validation should be stratified by class. Defaults to `TRUE` for `distribution = "multinomial"` and is only implemented for `"multinomial"` and `"bernoulli"`. The purpose of stratifying the cross-validation is to help avoiding situations in which training sets do not contain all classes.

`n.cores` The number of CPU cores to use. The cross-validation loop will attempt to send different CV folds off to different cores. If `n.cores` is not specified by the user, it is guessed using the `detectCores` function in the `parallel` package. Note that the documentation for `detectCores` makes clear that it is not failsafe and could return a spurious number of available cores.

predict.gbm *Predict method for GBM Model Fits*

Description

Predicted values based on a generalized boosted model object

Usage

```
## S3 method for class 'gbm'
predict(object, newdata, n.trees, type = "link",
  single.tree = FALSE, ...)
```

Arguments

object	Object of class inheriting from (gbm.object)
newdata	Data frame of observations for which to make predictions
n.trees	Number of trees used in the prediction. n.trees may be a vector in which case predictions are returned for each iteration specified
type	The scale on which gbm makes the predictions
single.tree	If single.tree=TRUE then predict.gbm returns only the predictions from tree(s) n.trees
...	further arguments passed to or from other methods

Details

predict.gbm produces predicted values for each observation in newdata using the the first n.trees iterations of the boosting sequence. If n.trees is a vector than the result is a matrix with each column representing the predictions from gbm models with n.trees[1] iterations, n.trees[2] iterations, and so on.

The predictions from gbm do not include the offset term. The user may add the value of the offset to the predicted value if desired.

If object was fit using gbm.fit there will be no Terms component. Therefore, the user has greater responsibility to make sure that newdata is of the same format (order and number of variables) as the one originally used to fit the model.

Value

Returns a vector of predictions. By default the predictions are on the scale of f(x). For example, for the Bernoulli loss the returned value is on the log odds scale, poisson loss on the log scale, and coxph is on the log hazard scale.

If type="response" then gbm converts back to the same scale as the outcome. Currently the only effect this will have is returning probabilities for bernoulli and expected counts for poisson. For the other distributions "response" and "link" return the same.