DigitalHouse > Coding School

DATA SCIENCE

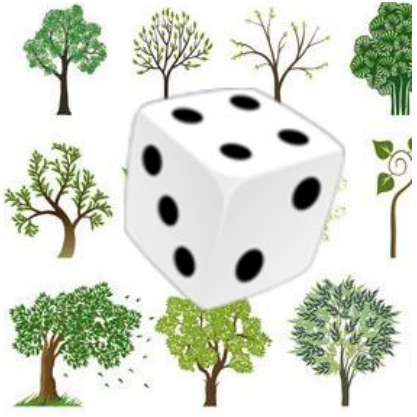
Clase 48

Random Forests
y Boosting

2017

Random Forests y Boosting

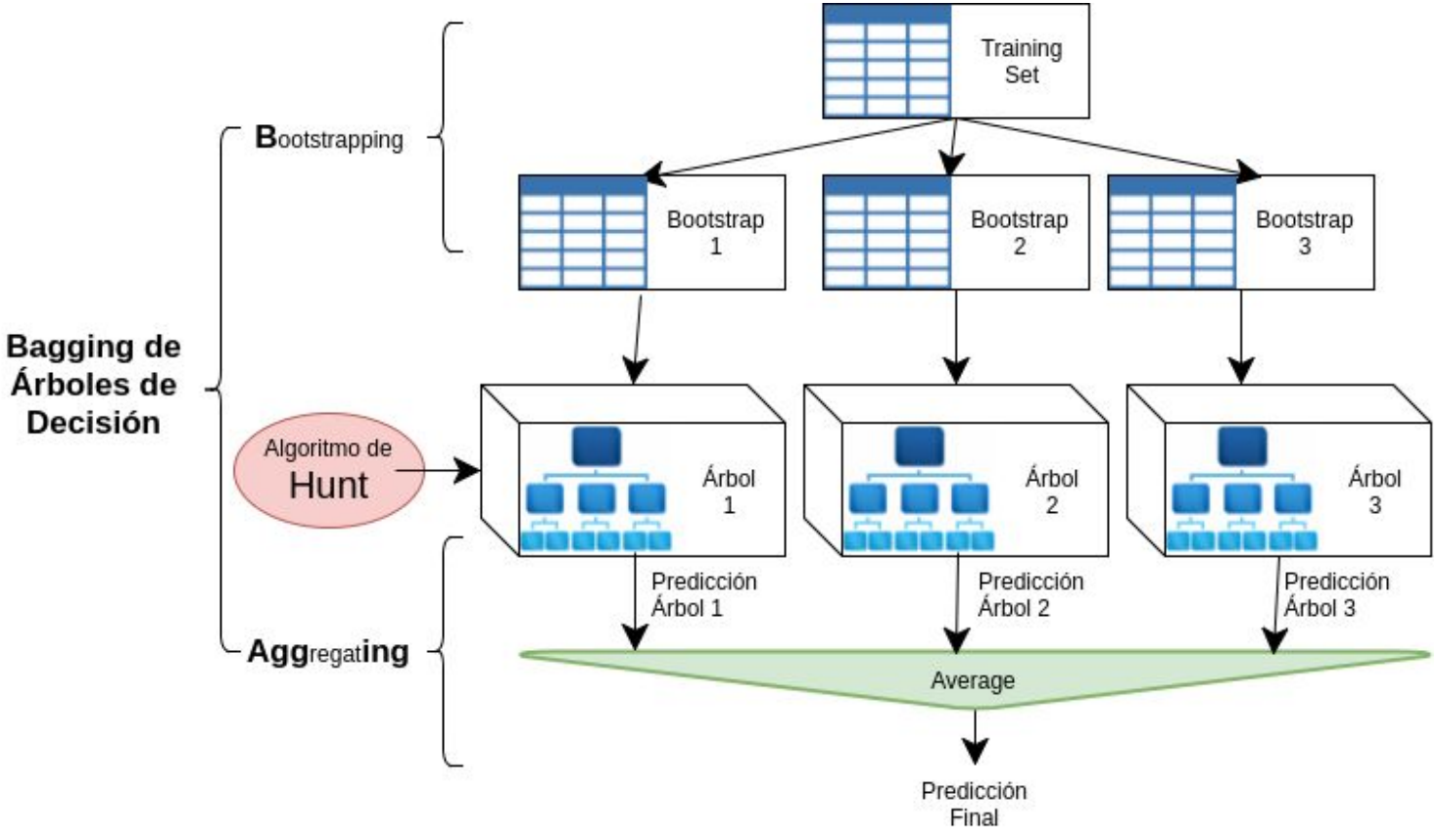
1. Explicar que es un Random Forest y la diferencia con el Bagging de árboles de decisión
2. Explicar qué son los modelos Extra Trees
3. Describir el Boosting y cómo difiere de Bagging
4. Aplicar Adaboost y Gradient Boosting a problemas de clasificación



Random Forest

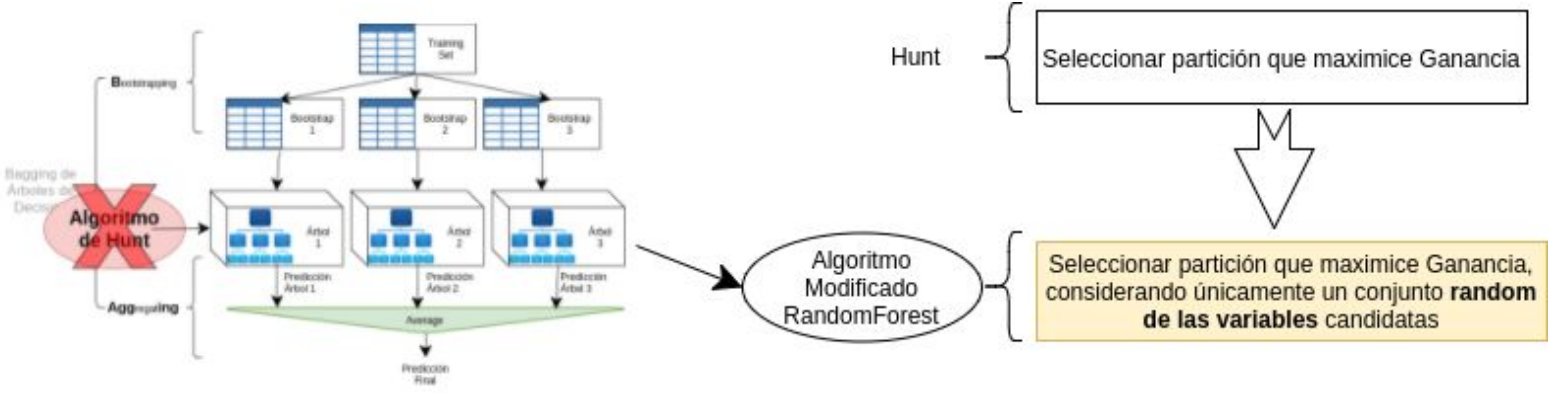
Introducción

- Como ya vimos, los árboles de decisión son modelos muy poderosos de machine learning. Son muy fáciles de usar porque requieren el seteo de muy pocos parámetros y se desempeñan bastante bien.
- Pero los Árboles de Decisión tienen algunas limitaciones, en particular, los árboles que crecen muy profundamente tienden a aprender patrones altamente irregulares, se sobre-ajustan.
- El bagging ayuda a mitigar este problema al exponer diferentes árboles a diferentes sub-sets del set de entrenamiento.


Training Set
Bootstrapping
Bootstrap 1
Bootstrap 2
Bootstrap 3
Bagging de Árboles de Decisión
Algoritmo de Hunt
Árbol 1
Árbol 2
Árbol 3
Predicción Árbol 1
Predicción Árbol 2
Predicción Árbol 3
Aggregating
Average
Predicción Final

- Los Random forests son una forma adicional de promediar múltiples árboles de decisión profundos, entrenados en diferentes partes del mismo set de entrenamiento, con el objetivo de reducir la varianza. Esto se produce a expensas de un pequeño aumento en el sesgo y cierta pérdida de interpretabilidad, pero en general **aumenta considerablemente el rendimiento** del modelo final.

- Los *Random forests* se diferencian del bagging de árboles de decisión en una sola cosa: usan un algoritmo de aprendizaje de árbol modificado que selecciona, en cada división candidata, un **subconjunto aleatorio de variables**. Este proceso se denomina a veces bagging de variables (feature bagging).

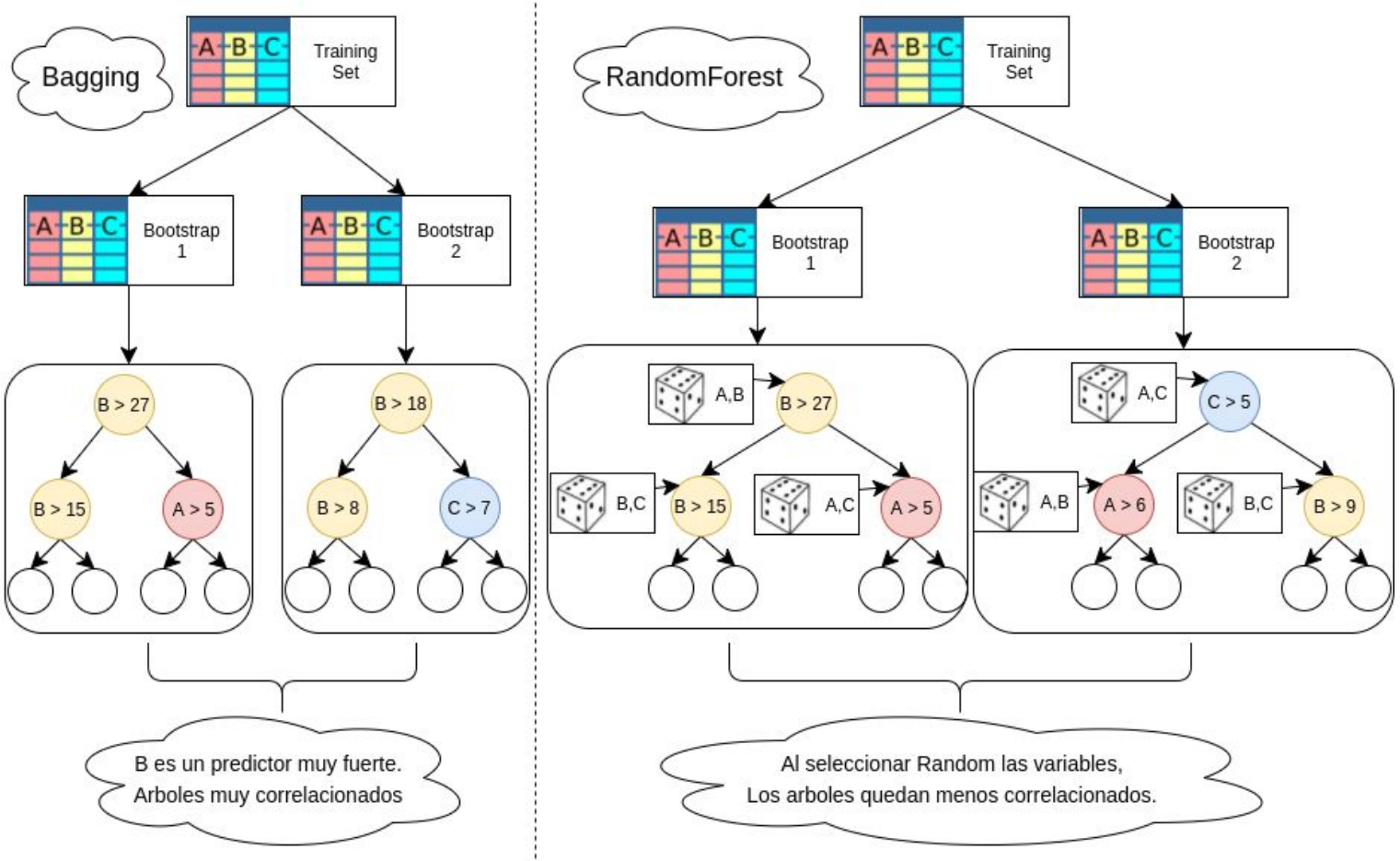


- La razón para hacer esto es la correlación de los árboles en una muestra de bootstrap normal: si una o algunas variables son **predictores muy fuertes** para la variable target, estas variables **serán seleccionadas en muchos de los árboles** base del bagging, haciendo que queden correlacionados. Seleccionando un subconjunto aleatorio de las variables en cada división, contrarrestamos esta correlación entre los árboles base, fortaleciendo el modelo final.

- Para un problema de clasificación con *p* variables, se suelen utilizar $\sqrt{p}$ de las variables en cada división.

- Para problemas de regresión, recomiendan utilizar *p/3*.

- Pero también podría considerarse como un hiperparámetro para tunear.


Bagging
A B C
Training Set
A B C
Bootstrap 1
A B C
Bootstrap 2
B > 27
B > 15
A > 5
B > 18
B > 8
C > 7
B es un predictor muy fuerte.
Arboles muy correlacionados
RandomForest
A B C
Training Set
A B C
Bootstrap 1
A B C
Bootstrap 2
A,B
B > 27
B,C
B > 15
A,C
A > 5
A,C
C > 5
A,B
A > 6
B,C
B > 9
Al seleccionar Random las variables,
Los arboles quedan menos correlacionados.

Algoritmo Random Forest

1. For $b = 1$ to B:

   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.

   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.

      i. Select m variables at random from the p variables.

      ii. Pick the best variable/split-point among the m.

      iii. Split the node into two daughter nodes.

2. Output the ensemble of trees $\{T_b\}_1^B$.

Extremely randomized trees

- La adición de un paso más de aleatorización produce árboles muy aleatorizados o **ExtraTrees**.
- Éstos se entrenan usando bagging y el método de selección aleatoria de variables, como en un *Random Forest* ordinario, pero con una capa random adicional.
- En lugar de calcular la combinación variable/división óptima local (ej ganancia de información), para cada variable en consideración se generan una división aleatoria (dentro del rango de la variable). Y luego se selecciona la variable/división que maximice la ganancia.
- La diferencia principal es que la división para cada variable no será la óptima, sino una seleccionada random.

Split_a_node(S)

Input: the local learning subset S corresponding to the node we want to split

Output: a split $[a < a_c]$ or nothing

– If **Stop_split**(S) is TRUE then return nothing.

– Otherwise select K attributes $\{a_1, \ldots, a_K\}$ among all non constant (in S) candidate attributes;

– Draw K splits $\{s_1, \ldots, s_K\}$, where $s_i = \textbf{Pick\_a\_random\_split}(S, a_i), \forall i = 1, \ldots, K$;

– Return a split s_* such that $\text{Score}(s_*, S) = \max_{i=1,\ldots,K} \text{Score}(s_i, S)$.

Selecciona K variables igual que Random Forest

Pick_a_random_split(S,a)

Inputs: a subset S and an attribute a

Output: a split

– Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of a in S;

– Draw a random cut-point a_c uniformly in $[a^S_{\min}, a^S_{\max}]$;

– Return the split $[a < a_c]$.

Stop_split(S)

Input: a subset S

Output: a boolean

– If $|S| < n_{\min}$, then return TRUE;

– If all attributes are constant in S, then return TRUE;

– If the output is constant in S, then return TRUE;

– Otherwise, return FALSE.

Split_a_node(S)

Input: the local learning subset S corresponding to the n

Output: a split $[a < a_c]$ or nothing

– If **Stop_split**(S) is TRUE then return nothing.

– Otherwise select K attributes $\{a_1, \ldots, a_K\}$ among all non constant (in S) candidate attributes;

– Draw K splits $\{s_1, \ldots, s_K\}$, where $s_i = \textbf{Pick\_a\_random\_split}(S, a_i), \forall i = 1, \ldots, K$;

– Return a split s_* such that $\text{Score}(s_*, S) = \max_{i=1,\ldots,K} \text{Score}(s_i, S)$.

Genera una división candidata para cada una de las K variables

Pick_a_random_split(S,a)

Inputs: a subset S and an attribute a

Output: a split

– Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of a in S;

– Draw a random cut-point a_c uniformly in $[a^S_{\min}, a^S_{\max}]$;

– Return the split $[a < a_c]$.

La división se genera con un random en el rango de la variable

Stop_split(S)

Input: a subset S

Output: a boolean

– If $|S| < n_{\min}$, then return TRUE;

– If all attributes are constant in S, then return TRUE;

– If the output is constant in S, then return TRUE;

– Otherwise, return FALSE.

Split_a_node(S)

Input: the local learning subset S corresponding to the node we want to split

Output: a split $[a < a_c]$ or nothing

– If **Stop_split**(S) is TRUE then return nothing.

– Otherwise select K attributes $\{a_1, \ldots, a_K\}$ among all non constant (in S) candidate attributes;

– Draw K splits $\{s_1, \ldots, s_K\}$, where s_i = **Pick_a_random_split**(S, a_i), $\forall i = 1, \ldots, K$;

– Return a split s_* such that Score(s_*, S) = $\max_{i=1,\ldots,K}$ Score(s_i, S).

Retorna la división que obtenga el máximo Score.

Pick_a_random_split(S,a)

Inputs: a subset S and an attribute a

Output: a split

– Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of a in S;

– Draw a random cut-point a_c uniformly in $[a^S_{\min}, a^S_{\max}]$;

– Return the split $[a < a_c]$.

Stop_split(S)

Input: a subset S

Output: a boolean

– If $|S| < n_{\min}$, then return TRUE;

– If all attributes are constant in S, then return TRUE;

– If the output is constant in S, then return TRUE;

– Otherwise, return FALSE.

Práctica guiada:

Random Forest y Extra Trees

Práctica: Random Forest y ExtraTrees en Scikit Learn

En esta práctica vamos a comparar el rendimiento de los siguientes algoritmos:

- Árboles de decisión
- Bagging + Árboles de decisión
- Random Forest
- Extra Trees

Para ello vamos a comenzar con la lectura del dataset de aceptabilidad de autos.

```
import pandas as pd
 df = pd.read_csv('./datasets/car.csv')
 df.head()
```

	buying	maint	doors	persons	lug_boot	safety	acceptability
0	vhigh	vhigh	2	2	small	low	unacc
1	vhigh	vhigh	2	2	small	med	unacc
2	vhigh	vhigh	2	2	small	high	unacc
3	vhigh	vhigh	2	2	med	low	unacc
4	vhigh	vhigh	2	2	med	med	unacc

Esta vez vamos a codificar las características usando un esquema de codificación One Hot, es decir, las consideraremos como variables categóricas. También necesitamos codificar la etiqueta usando el LabelEncoder.

```
from sklearn.preprocessing import LabelEncoder
 y = LabelEncoder().fit_transform(df['acceptability'])
 X = pd.get_dummies(df.drop('acceptability', axis=1))
 X.ix[:,0:8].head()
```

	buying_high	buying_low	buying_med	buying_vhigh	maint_high	maint_low	maint_med	maint_vhigh
0	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1
2	0	0	0	1	0	0	0	1
3	0	0	0	1	0	0	0	1
4	0	0	0	1	0	0	0	1

Para que los resultados sean consistentes hay que exponer los modelos exactamente al mismo esquema de validación cruzada.

```
from sklearn.model_selection import cross_val_score,StratifiedKFold
 cv = StratifiedKFold(n_splits=3, random_state=41, shuffle=True)
```

Ahora vamos a inicializar el clasificador de árbol de decisión y evaluar su rendimiento:

```
from sklearn.tree import DecisionTreeClassifier
 from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier, BaggingClassifier


def evaluar_rendimiento(modelo, nombre):
     s = cross_val_score(modelo, X, y, cv=cv, n_jobs=-1)
     print "Rendimiento de {}:\t{:0.3} ± {:0.3}".format( \
         nombre, s.mean().round(3), s.std().round(3))


dt = DecisionTreeClassifier(class_weight='balanced')


evaluar_rendimiento(dt,"Árbol de decisión")


Rendimiento de Árbol de decisión:  0.965 ± 0.006
```

Ahora intenten ustedes con los modelos restantes y evalúen el rendimiento.
Sería recomendable que vean la documentación para ver qué parámetros aceptan.
http://scikit-learn.org/stable/modules/ensemble.html#forest

```
dt = DecisionTreeClassifier(class_weight='balanced')
 bdt = BaggingClassifier(DecisionTreeClassifier())
 rf = RandomForestClassifier(class_weight='balanced')
 et = ExtraTreesClassifier(class_weight='balanced')

 evaluar_rendimiento(dt,  "Árbol de decisión")
 evaluar_rendimiento(bdt, "Bagging AD")
 evaluar_rendimiento(rf,  "Random Forest")
 evaluar_rendimiento(et,  "Extra Trees")
```

```
Rendimiento de Árbol de decisión:   0.965 ± 0.006
 Rendimiento de Bagging AD:          0.976 ± 0.006
 Rendimiento de Random Forest:       0.943 ± 0.002
 Rendimiento de Extra Trees:         0.956 ± 0.002
```

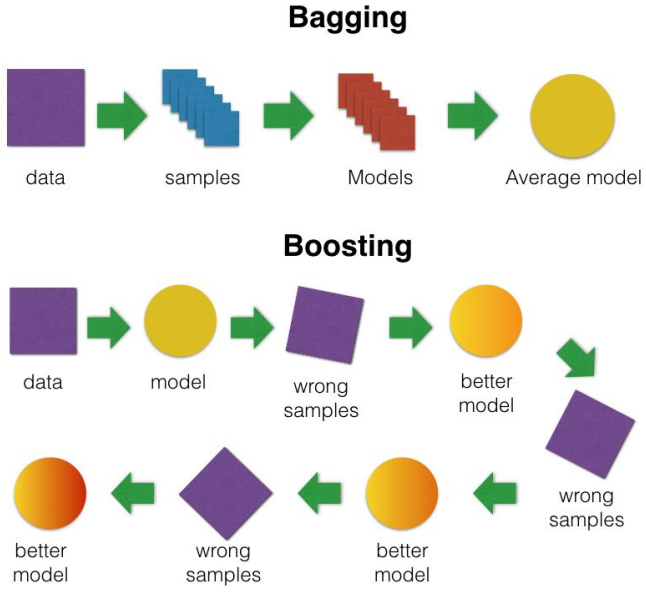
En este caso, el bagging de árboles de decisión anda mejor que el resto.
Con otros set de datos, los modelos Random Forest y Extra Trees podrían tener mejores resultados y merecen ser probados.

Boosting

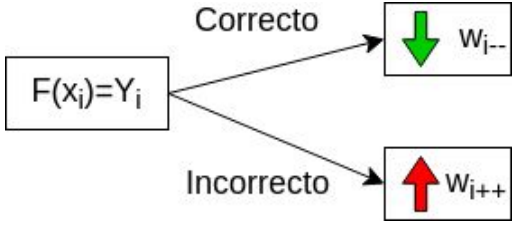
Introducción

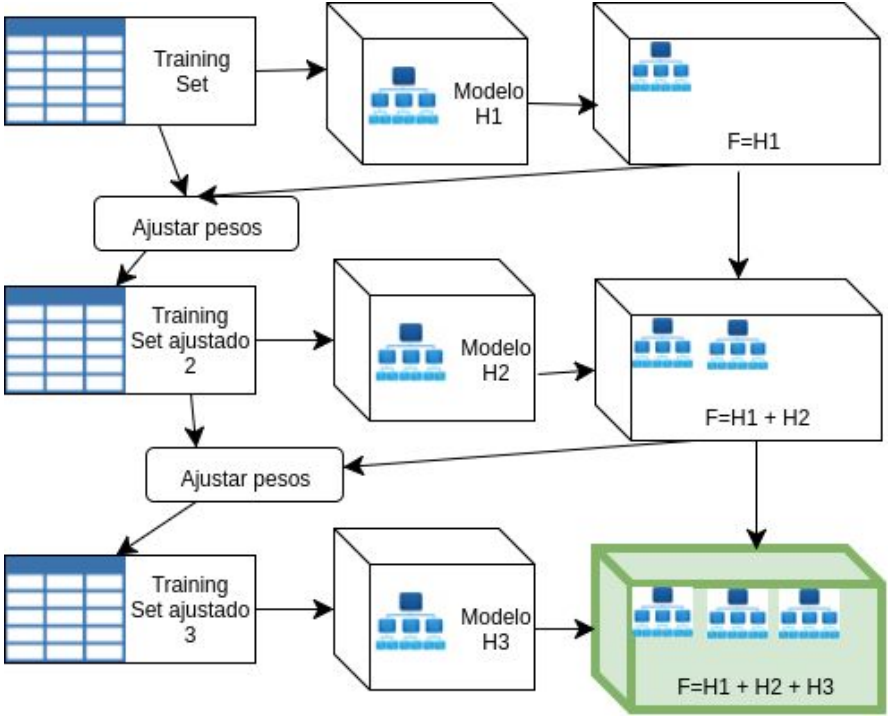
- Con *Bagging* y *Random Forests* entrenamos modelos en subsets separados y luego combinamos su predicción. Es como si estuviéramos paralelizando el entrenamiento y luego combinando los resultados.

- El *Boosting* es otra técnica de ensamble la cual es **secuencial**.

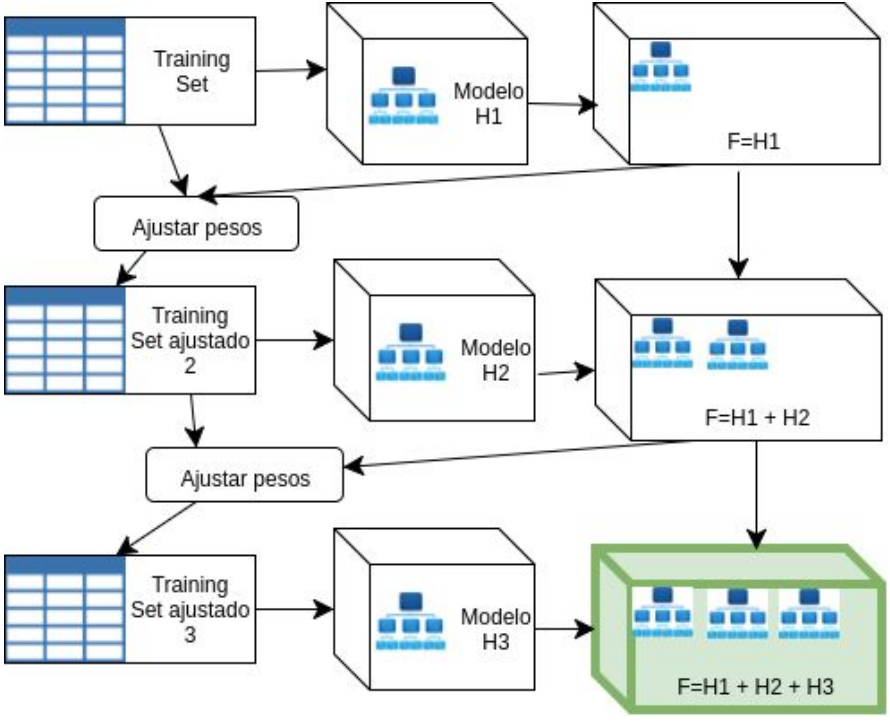


- El ***Boosting*** es un **procedimiento iterativo** que va construyendo un modelo final en pasos.
- En cada nuevo paso intentará aprender de los errores cometidos en los pasos previos.
- Trabaja sobre los errores del modelo anterior o bien usándolos para cambiar la ponderación en el siguiente modelo o bien entrenando un modelo que prediga los mismos.





- La primera iteración utiliza **pesos uniformes** para todos los registros. En las iteraciones posteriores, los pesos se ajustan para **enfatizar los errores** en la iteración anterior.
- La predicción final se construye mediante un **voto ponderado** de los distintos modelos base. Donde los pesos para cada modelo base dependen de su error de entrenamiento.
- Adaboost toma un modelo base débil e intenta hacerlo fuerte al re-entrenarlo en las muestras mal clasificadas.



Algoritmo AdaBoost.M1.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   (c) Compute $\alpha_m = \log((1 - \mathrm{err}_m)/\mathrm{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \mathrm{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Se inicializan todos los pesos iguales.
Habrá un peso Wi asociado a cada uno de los ejemplos Xi del set de entrenamiento. Siendo N la cantidad de ejemplos en el set de entrenamiento

El algoritmo entrenará M clasificadores.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Se entrena el clasificador Gm, considerando el set de entrenamiento y el peso wi asignado a cada uno de los ejemplos.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Se calcula el error de clasificación ponderado de Gm.
ERRm será la suma del peso de los ejemplos mal clasificados / suma todos los pesos
Mínimo de 0 cuando no haya errores.
Máximo de 1 cuando sean todos errores.
Se puede ver que los ejemplos de alto peso mal clasificados influyen más que los de pesos bajos.

1

2. For $m = 1$ to M:

   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.

   (b) Compute

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

   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.

   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.

3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Se calcula el coeficiente de aporte de este Clasificador en el ensamble. El valor será mayor cuanto más preciso sea el clasificador Gm, dándole mayor importancia a su voto en el comité.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1,$
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Observar que este coeficiente es el que determina el peso del voto de este clasificador en el comité resultante

	Err	1-Err/ Err	log(1-Err/ Err)
Buen clasificador	0.01	99	1.99
Azar	0.5	1	0
Mal clasificador	0.99	0.01	-2

1.

2.

(a) Fit a classifier $G_m(x)$ to the training data using weights w_i.

(b) Compute

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

(c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.

(d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.

3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Se recalculan los pesos de los ejemplos del set de entrenamiento.
Aumentando los pesos de aquellos ejemplos mal clasificados.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Existen variaciones de este algoritmo donde además se disminuye el peso de los ejemplos bien clasificados. Y se agrega un paso posterior de normalización de los pesos.

Se obtiene como resultado el ensamble G(x) donde cada Gm(x) hace su aporte con su voto ponderado por su coeficiente Am.

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

El **Gradient Boosting** es una generalización de boosting para funciones de pérdida diferenciables. Es un procedimiento preciso y efectivo que se puede usar para problemas de regresión y clasificación.
Modelos de Gradient Boosting de árboles se utilizan en una variedad de áreas, incluyendo ranking de búsqueda web, ecología, etc.

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Los pseudo-residuos equivalen a la negativa de la derivada de la función de pérdida con respecto a la función F.
Típicamente L es ½ (Y-F(X))^2, entonces r = Y - F

Input: training set $\{(x$

Algorithm:

1. Initialize mode

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:

   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

Para entrenar tomamos como target los pseudo-residuos

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Input: training set $\{(x_i, y_i)$

Algorithm:

Para agregar el nuevo modelo al ensamble resolvemos un problema de optimización, consistente en buscar el gamma que nuestra función de pérdida.

1. Initialize model with

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

Sumamos el modelo al ensamble, premultiplicándolo por el peso que encontramos en el paso previo.

1. Initialize model with

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Las ventajas de GBT son:

- Manejo natural de datos de tipo mixto (atributos heterogéneos)
- Potencia predictiva
- Robustez a outliers (a través de funciones de pérdida robustas)

Práctica Independiente: AdaBoost y Gradient Boosting

Práctica Independiente: Ada Boost y Clasificador Gradient Boosting

- Probar la performance de los modelos AdaBoost y GradientBoostingClassifier sobre el dataset de autos.
- Usar el código desarrollado antes como punto de partida.

Solución:

```
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
 ab = AdaBoostClassifier()
 gb = GradientBoostingClassifier()
 evaluar_rendimiento(ab, "AdaBoost")
 evaluar_rendimiento(gb, "GradientBoostingClassifier")
```

```
Rendimiento de AdaBoost:                        0.811 ± 0.002
 Rendimiento de GradientBoostingClassifier:   0.982 ± 0.006
```

CONCLUSIÓN

- Aquí vimos Random Forest, ExtraTrees y Boosting. Estas son formas diferentes de mejorar el rendimiento de un aprendiz débil.

- Algunos de estos métodos funcionan mejor en unos casos, algunos mejor en otros.
  Por ejemplo, los Árboles de Decisión son más ágiles y fáciles de comunicar, pero tienen tendencia a sobre-ajustarse.
  Por otro lado, los métodos de Ensamble se desenvuelven mejor en escenarios más complejos, pero pueden llegar a ser muy complicados y difíciles de explicar.