

DigitalHouse >
Coding School

DATA SCIENCE

MÓDULO 6

Boosting, Gradient
Boosting y XGBoost

Boosting, Gradient Boosting y XGBoost



1

Describir el Boosting y cómo difiere de Bagging

2

Aplicar Adaboost y Gradient Boosting a problemas de clasificación

3

Presentar las ventajas de emplear XGBoost

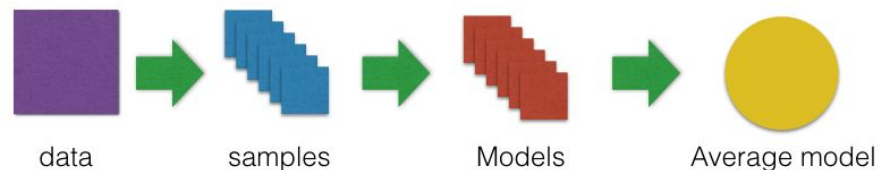
Boosting



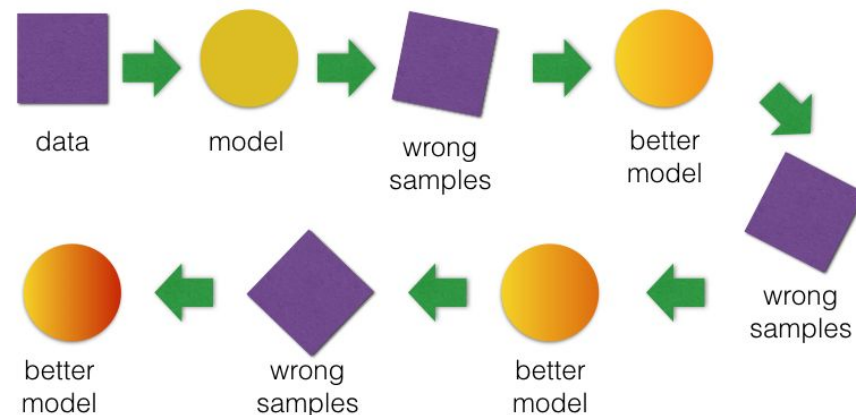
Introducción

- 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**.

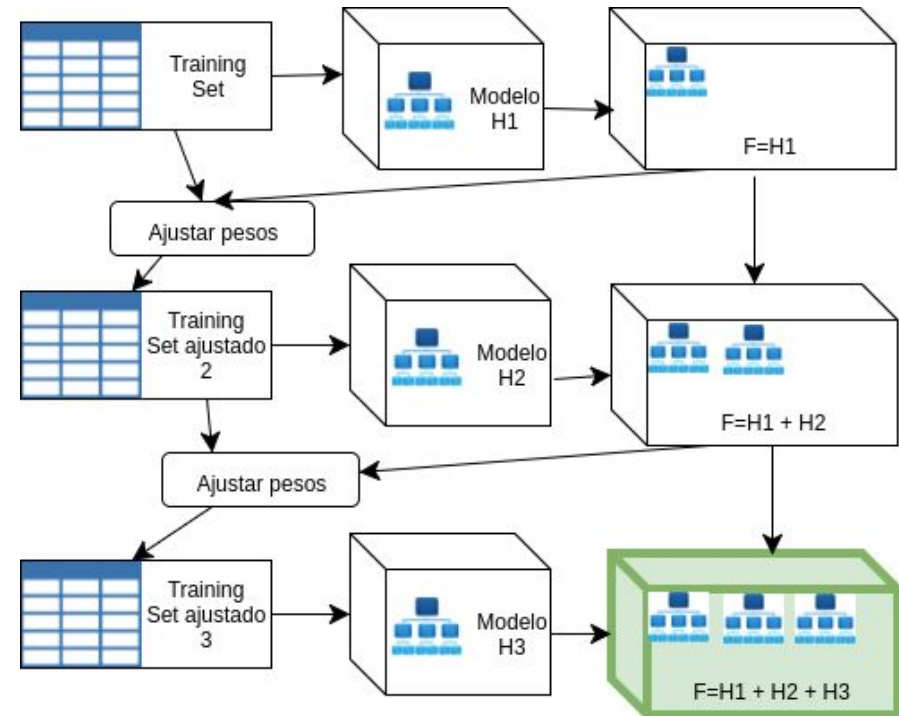
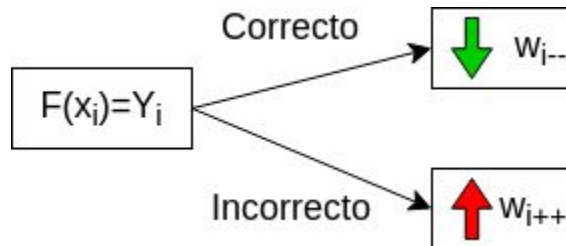
Bagging



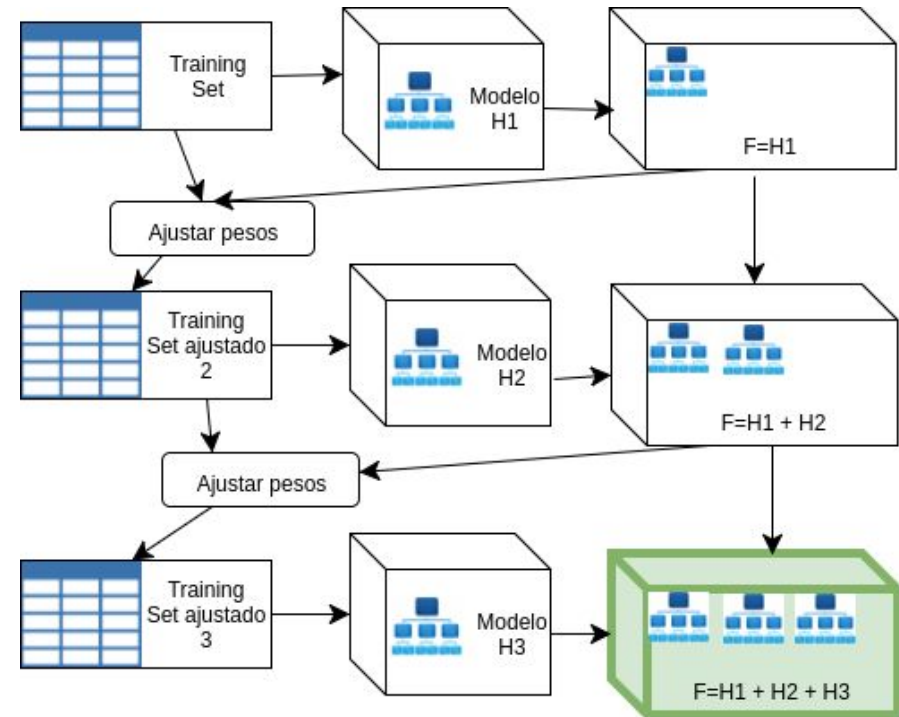
Boosting



- 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 **enfatar 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 $w_i = 1/N$, $i = 1, 2, \dots, N$.
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 $\alpha_m = \log((1 - \text{err}_m) / \text{err}_m)$.
 - (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \dots, N$.
3. Output $G(x) = \text{sign} \left[\sum_{m=1}^M \alpha_m G_m(x) \right]$.

Se inicializan todos los pesos iguales.
Habrá un peso W_i asociado a cada uno de los ejemplos X_i 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 $w_i = 1/N$, $i = 1, 2, \dots, N$.

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

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

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

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
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 $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \dots, N$.

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 G_m .

ERR $_m$ será la suma del peso de los ejemplos mal clasificados / suma todos los pesos

Mínimo de 0 cuando no haya errores.

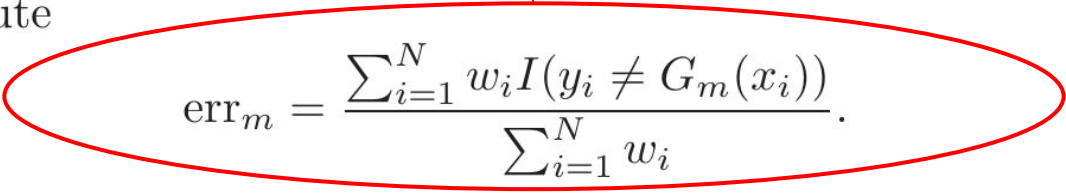
Máximo de 1 cuando sean todos errores.

1 Se puede ver que los ejemplos de alto peso mal clasificados influyen más que los de pesos bajos.

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

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 G_m , dándole mayor importancia a su voto en el comité.

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
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 $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, \dots, N$.
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

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

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \dots, N$.

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

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 $G_m(x)$ hace su aporte con su voto ponderado por su coeficiente α_m .

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
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 $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \dots, N$.
3. Output $G(x) = \text{sign} \left[\sum_{m=1}^M \alpha_m G_m(x) \right]$.

Gradient Boosting



- Los algoritmos de **gradient boosting** son algoritmos para **aprendizaje supervisado** que interpretan al boosting como un problema de optimización → **generan una función de pérdida y buscan optimizarla**. Esta idea la desarrolló por primera vez Leo Breiman.
- Entrenamos **modelos débiles de forma secuencial** para ir minimizando la función de pérdida.
- La pérdida está representada por los residuos.

- El objetivo de cualquier problema de aprendizaje supervisado es el de definir una función de pérdida y minimizarla. En el caso de regresión, por ejemplo tenemos el ECM, definido como:

$$Loss = MSE = \sum (y_i - y_i^p)^2$$

where, y_i = ith target value, y_i^p = ith prediction, $L(y_i, y_i^p)$ is Loss function

Queremos que nuestras predicciones minimicen a nuestra función de pérdida.

- Usando el descenso del gradiente podemos actualizar nuestras predicciones con una tasa de aprendizaje *alfa* de modo tal que la suma de los residuos tienda a cero:

$$y_i^p = y_i^p + \alpha * \delta \sum (y_i - y_i^p)^2 / \delta y_i^p$$

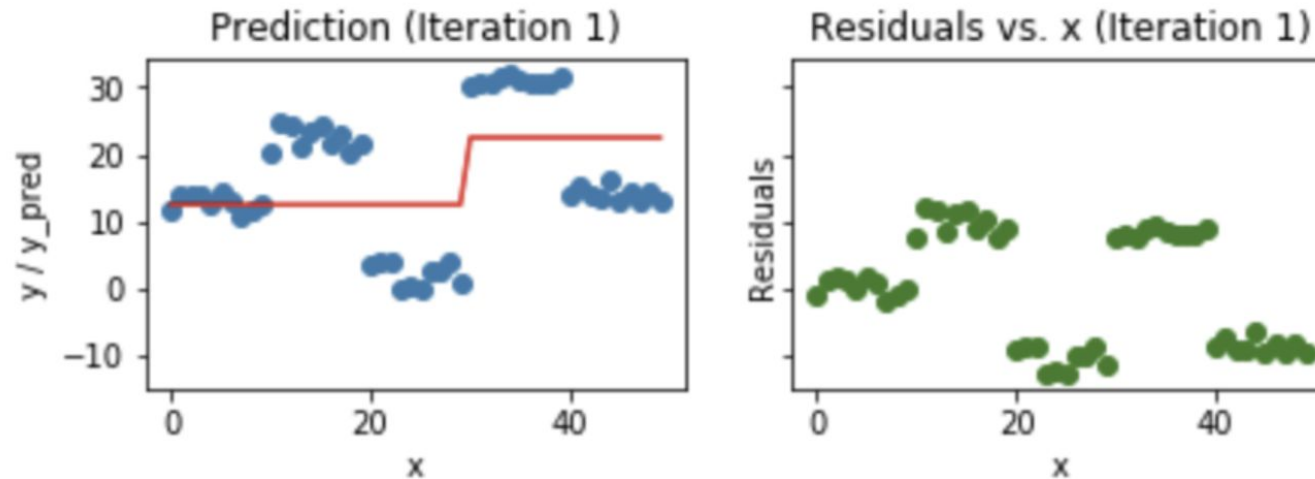
which becomes, $y_i^p = y_i^p - \alpha * 2 * \sum (y_i - y_i^p)$

where, α is learning rate and $\sum (y_i - y_i^p)$ is sum of residuals

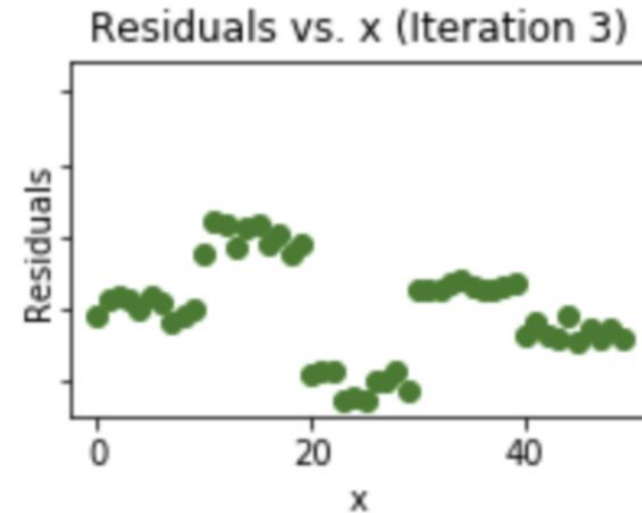
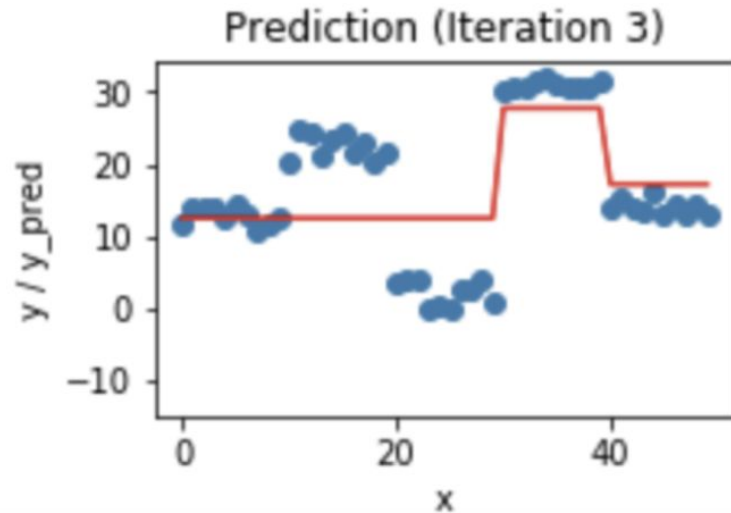
- Algoritmo de gradient boosting:
 1. Ajustar un modelo utilizando a **X** como features y a **y** como target. Obtener las predicciones del modelo **y_predicted**
 2. Calcular los residuos: **e1 = y - y_predicted**
 3. Ajustar un modelo utilizando a **X** como features y a **e** como target. Obtener las predicciones **e1_predicted**.
 4. Sumar las predicciones: **y_predicted2 = y_predicted + alfa * e1_predicted**, donde alfa es el learning rate.
 5. Calcular los residuos **e2 = y - y_predicted2**

Repetir los pasos 2 a 5 hasta que el modelo comienza a sobre ajustar.

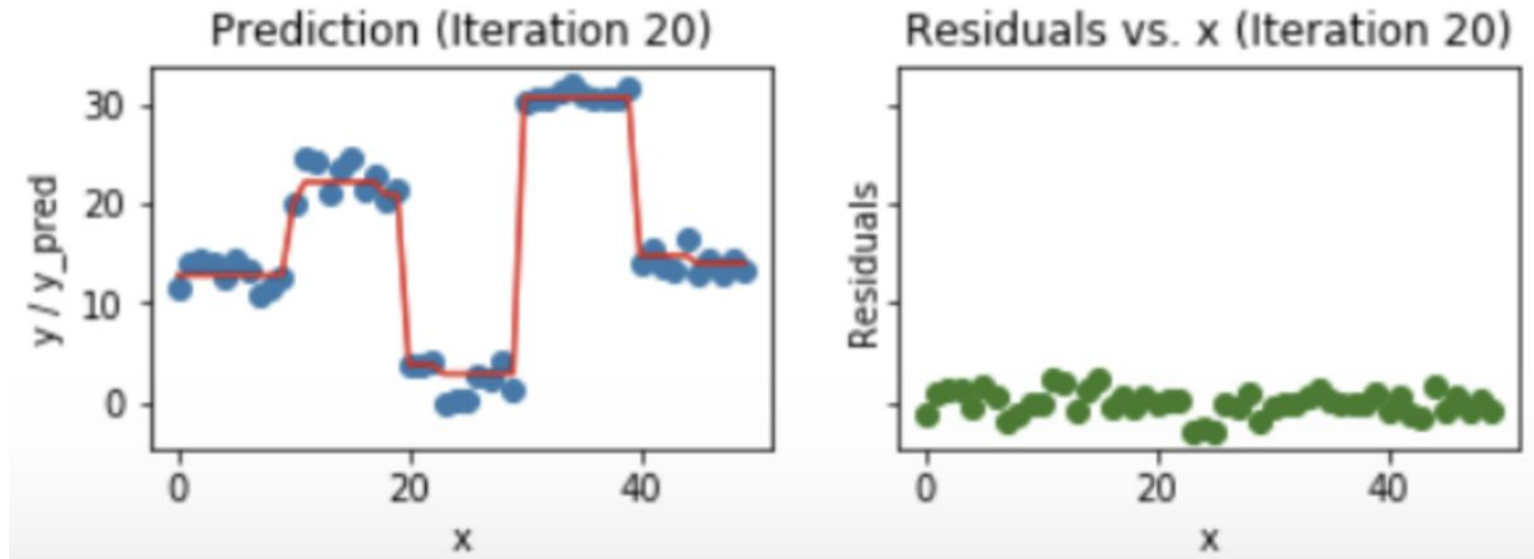
- Analicémoslo paso por paso. En la primera iteración, se usa un modelo simple para ajustar a los datos.
- Los residuos se pueden observar en el gráfico de la derecha. La función de pérdida busca minimizar estos residuos.



- Luego, se agregan modelos débiles para que se concentren en las áreas donde los modelos ya existentes performan mal.
- Vemos como después de 3 iteraciones el ensamble de modelos débiles ya comienza a performar mejor.



- Luego de 20 iteraciones el modelo ajusta muy bien a los datos y los residuos cayeron prácticamente a cero.



XGBoost



- El primer release de **XGBoost** nace en marzo de 2014 y se empieza a emplear en competencias de Kaggle en 2015, rápidamente convirtiéndose en el algoritmo más popular entre los ganadores en problemas con datasets estructurados.
- Este algoritmo es una **reimplementación de Gradient Boosting**, con importantes mejoras que veremos en esta presentación.
- Fue diseñado por Tianqi Chen, de la Universidad de Washington, y rápidamente recibió numerosos colaboradores a su [proyecto](#).

¿Por qué usar XGBoost en vez de simplemente Gradient Boosting?

1- Velocidad

XGBoost se impuso por lograr ajustes de modelos ampliamente más rápidos que los implementados en las librerías previamente existentes (Scikit Learn de Python, gbm de R, H2O y Spark MLlib)

2- Escalabilidad

La implementación de XGBoost permite entrenar usando muchísima más información ya que es muy eficiente en el uso de RAM.

¿Por qué usar XGBoost en vez de simplemente GBM?

3- Rendimiento

La capacidad de realizar una búsqueda de hiper parámetros más amplia en la misma cantidad de tiempo y de entrenar con más información, sumado a una mejora clave en la función objetivo, permite que con este algoritmo se obtengan mejores resultados.

XGBoost soporta las 3 principales implementaciones de Gradient Boosting:

1- Gradient Boosting

2- Stochastic Gradient Boosting, con sub-muestras en filas, columnas y columnas por split

3- Gradient Boosting con regularización, con normas L1 y L2

Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$$

Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \boxed{\gamma T} + \frac{1}{2} \lambda ||w||^2$$

T es la cantidad de
hojas



Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \boxed{\gamma T} + \frac{1}{2} \lambda ||w||^2$$


Entonces gamma es cuánto debe reducir la función de pérdida un split para que se produzca

Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \boxed{\gamma T} + \frac{1}{2} \lambda ||w||^2$$

Cuanto mayor es gamma más difícil es realizar un split, entonces obtenemos árboles más simples



Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$$

Esta es la norma 2 o L2, que vimos en ridge.

Gradient Boosting con regularización

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$$

Los w son los scores en las hojas. En este caso, penaliza los scores muy grandes, favoreciendo que ninguna hoja sea especialmente importante

Gradient Boosting con regularización

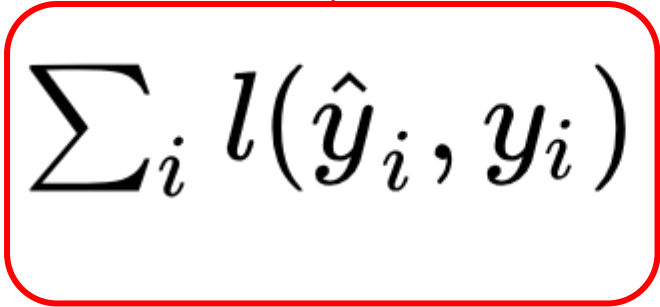
$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$$

La versión actual también incluye la norma 1



Se pueden incluir funciones de pérdida personalizadas


$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$$

Una de las mejoras de XGBoost es el algoritmo para encontrar los splits. XGBoost implementa tres tipos de algoritmos:

- **Exact Greedy:** este algoritmo es el que empleaban previamente Scikit y GBM (R). Esencialmente es un método performante de probar todos los posibles puntos de split y quedarse con el mejor de todos. Este método es mucho más lento que los próximos dos.
- **Aproximado:** este algoritmo no prueba todos los posibles puntos de corte sino que sólo toma los percentiles de cada variable y prueba ellos. Se recalculan los bins en cada iteración.
- **Histograma:** los valores de las variables continuas son almacenados en bins de un histograma, y reutilizados a lo largo de los cálculos. Luego, el split se hace entre los bins del histograma. Permite entre 2 y 256 bins.

Además de las mejoras antes mencionadas, se incluyó en el desarrollo de XGBoost lo siguiente:

- **Shrinkage (eta)**: básicamente este parámetro lo que hace es quitar peso a cada nuevo árbol que se agrega al ensamble, multiplicando su influencia por una constante.
- **Sampling de columnas**: de la misma forma que Random Forest, esta librería implementa el sampling de columnas para reducir el overfitting e incrementar la variabilidad de los árboles. `colsample_bytree` y `colsample_bylevel`.
- **Sampling de filas (subsample)**: qué proporción de las filas tomar para entrenar cada árbol.
- **Ausencia de datos**: a los valores nulos XGBoost le asignará una dirección por default en cada nodo. Es decir, los agrupará con los valores de la izquierda o de la derecha del split, según cuál sea la dirección que da mejores resultados.

En febrero de 2016 surge la versión beta de **LightGBM**, un proyecto desarrollado por Microsoft. Este algoritmo se caracteriza por ocupar mucha menos memoria RAM y ser, generalmente, mucho más rápido que el XGBoost original, obteniendo prácticamente los mismos resultados, si no mejores.

LightGBM emplea un algoritmo para buscar los splits diferentes llamado GOSS (Gradient-based one-side sampling). Este algoritmo hace lo siguiente:

- Para cada observación calcula el gradiente.
- Se dejan sólo las instancias con un gradiente elevado y se samplea un porcentaje de las instancias con un gradiente bajo. La idea es que las observaciones con un gradiente chico ya pueden ser bien predichas.
- Se entrena el árbol con este set de datos, y para evaluar los splits a las observaciones muestreadas se las multiplica por una constante.

- En 2017 Yandex abre el [proyecto](#) de **CatBoost**. Este algoritmo también es muy rápido y presenta resultados excelentes sin necesidades de costosas búsquedas de hiperparámetros.
 - Permite transformar a one-hot encoding las variables que tengan menos valores distintos que una determinada cantidad, y tiene un manejo especial para las variables categóricas. En este segundo caso, el algoritmo implementa distintas métricas que se aplican en cada split para analizar la relación entre las categorías y el label.
 - Este algoritmo además implementa una alerta para detección temprana de overfitting que permite detener rápidamente el entrenamiento ante esta situación.
- Otro algoritmo que está obteniendo buenos resultados es FastBDT, pero no parece ser un proyecto ni tan general ni tan estable como los otros.

XGBoost	LightGBM
<p>n_estimators: Number of boosted trees to fit.</p> <p>max_depth: Maximum tree depth for base learners.</p> <p>learning_rate: Boosting learning rate (eta)</p> <p>colsample_bytree: Subsample ratio of columns when constructing each tree.</p> <p>subsample: Subsample ratio of the training instance.</p> <p>min_child_weight: Minimum sum of instance weight(hessian) needed in a child.</p> <p>reg_alpha: L1 regularization term on weights</p> <p>reg_lambda: L2 regularization term on weights</p>	
<p>gamma: Minimum loss reduction required to make a further partition on a leaf node of the tree.</p> <p>max_delta_step: Maximum delta step we allow each tree's weight estimation to be.</p> <p>colsample_bylevel: Subsample ratio of columns for each split, in each level.</p> <p>booster: Specify which booster to use: gbtree, gblinear or dart</p> <p>www.digitalhouse.com</p>	<p>num_leaves: max number of leaves in one tree</p> <p>subsample_for_bin: number of data that sampled to construct histogram bins. Setting this to larger value will give better training result, but will increase data loading time. Set this to larger value if data is very sparse</p> <p>min_split_gain: the minimal gain to perform split</p> <p>min_child_samples: minimal number of data in one leaf.</p> <p>subsample_freq: frequency for bagging. 0 means disable bagging; k means perform bagging at every k iteration. to enable bagging, subsample should be set to value smaller than 1.0 as well.</p> <p>boosting_type: puede ser 'gbdt', 'rf', 'dart' y 'goss'.</p>

Catboost

Algunos parámetros similares a los algoritmos previos: **n_estimators**, **learning_rate**, **max_depth**, **reg_lambda**, **colsample_bylevel**, **subsample**.

border_count (max_bin): The number of splits for numerical features. Allowed values are integers from 1 to 255 inclusively.

od_pval, **od_wait** y **od_type** son parámetros para la detección de overfitting.

nan_mode: Forbidden', 'Min', 'Max'.

Además existen parámetros para controlar el manejo de variables categóricas, como **store_all_simple_ctr**, **max_ctr_complexity**, **counter_calc_method**, entre otros.

one_hot_max_size: Use one-hot encoding for all features with a number of different values less than or equal to the given parameter value. Ctrs are not calculated for such features.