

TDVI: Inteligencia Artificial

Ensamble de Modelos II: Boosting / XGBoost

UTDT - LTD

Estructura de la clase

- Repaso
- Boosting básico
- XGBoost

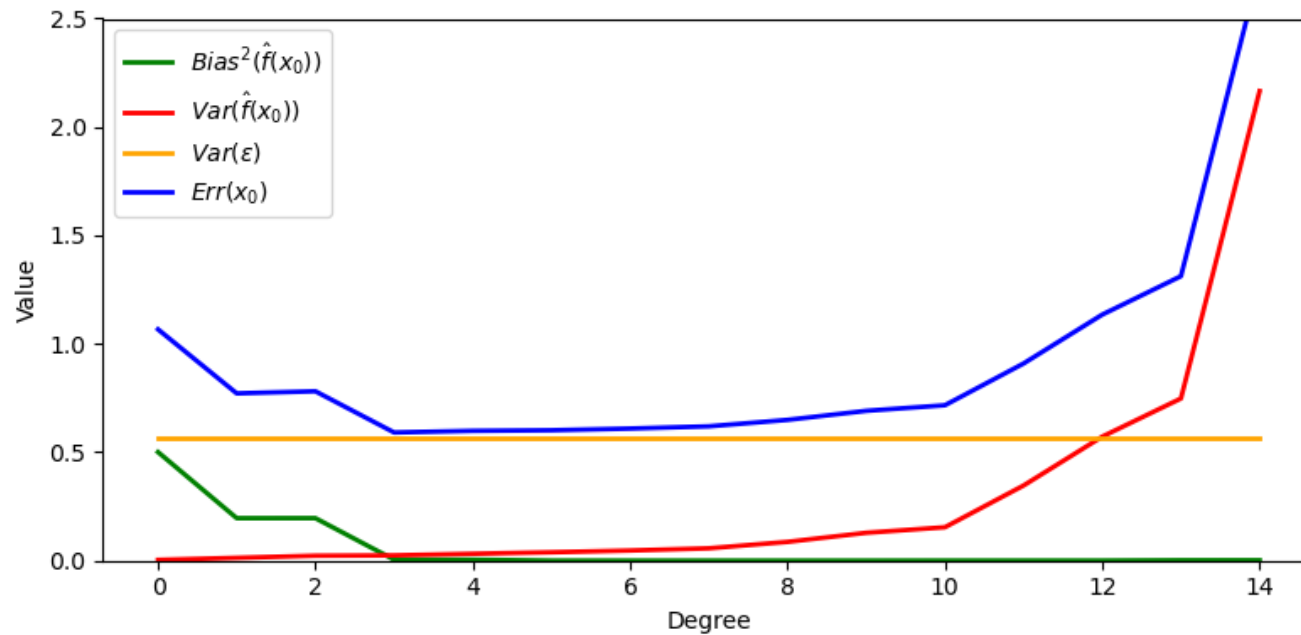
Estructura de la clase

- Repaso
- Boosting básico
- XGBoost

Repaso

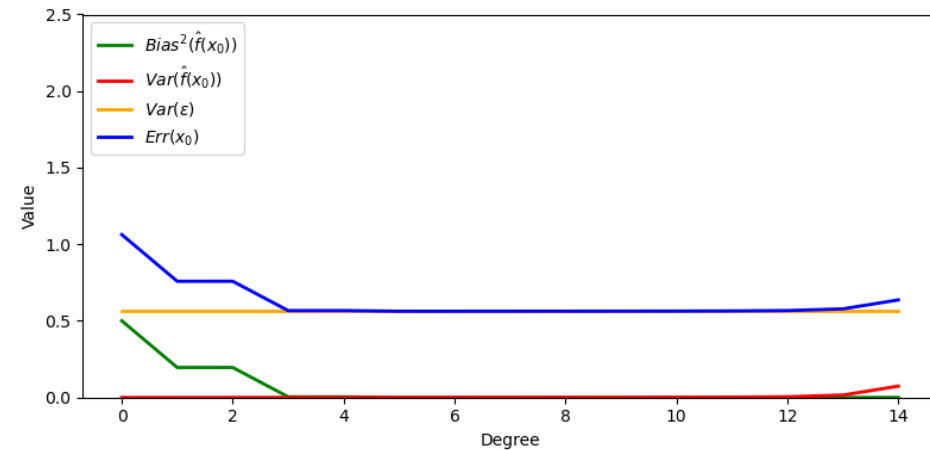
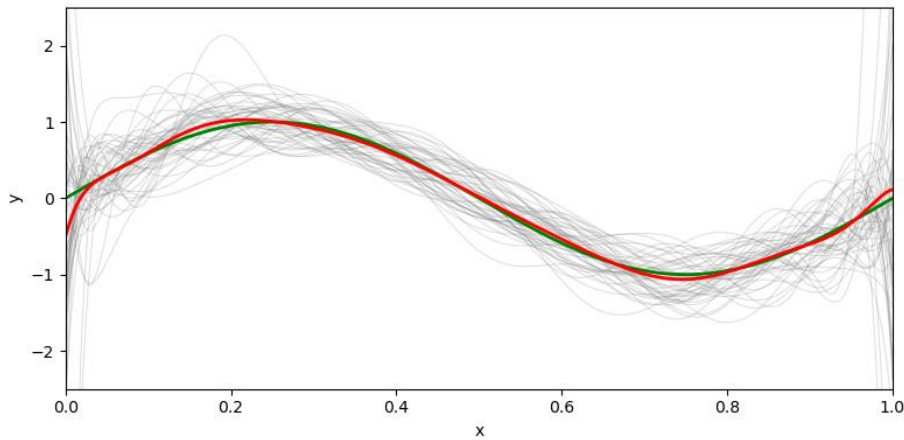
El error esperado, se puede expresar como:

$$Err(x_0) = \sigma_\epsilon^2 + Bias^2(\hat{f}(x_0)) + Var(\hat{f}(x_0))$$



Repaso

Si pudiéramos **promediar las predicciones** de modelos entrenados con distintas instanciaciones del train set y considerar al promedio como el modelo final, **reduciríamos la varianza**

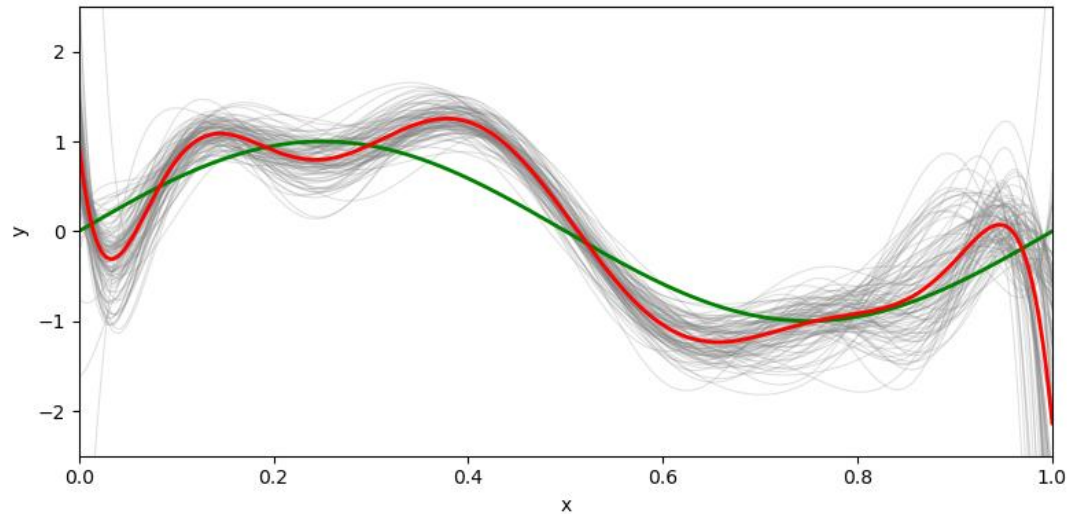


Realidad: **no lo podemos hacer**

Repaso

Intentamos hacer algo similar mediante *bagging*, suele mejorar la performance

Pero los **modelos base** suelen estar muy correlacionados



Idea: utilizar árboles y decorrelacionarlos *perturbando cómo se construyen*

Random forest: idéntico a bagging, pero en cada split de cada árbol, no se consideran todas las variables para hacer el split, **se considera sólo una muestra** (que varía de split en split)

Estructura de la clase

- Repaso
- Boosting básico
- XGBoost

Boosting básico

Idea:

1. Entreno un árbol de decisión
2. Veo en qué observaciones de entrenamiento predijo mal
3. Construyo un segundo árbol de decisión que se enfoque en aquellas observaciones que el primero predijo mal
4. Tomo como mi predicción final alguna combinación de las predicciones de cada árbol

¿Tiene sentido?

¿Podría seguir con un tercer árbol?

Boosting básico

Boosting hace uso de esta idea. Cada árbol es construido de **manera secuencial**, usando información de los árboles previos. Es una estrategia para **ensamblar modelos**

Boosting (como bagging) puede aplicarse a diferentes modelos de clasificación y regresión

Ahora vamos a enfocarnos en el caso de **árboles de decisión aplicados a regresión**

Boosting básico

Algorithm 8.2

Boosting for Regression Trees

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \dots, B$, repeat:

(a) Fit a tree \hat{f}^b with d splits ($d + 1$ terminal nodes) to the training data (X, r) .

(b) Update \hat{f} by adding in a shrunk version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$$

(8.10)

- (c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$

(8.11)

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^B \lambda \hat{f}^b(x).$$

(8.12)

Ejemplo:

Inicialización

y	\hat{f}	r
10	0	10
3	0	3
2	0	2
...
1	0	1
5	0	0

Iteración 1

\hat{f}^1	$\lambda * \hat{f}^1$	\hat{f}	r
8	0.8	0.8	9.2
1	0.1	0.1	2.9
4	0.4	0.4	1.6
...
2	0.2	0.2	0.8
6	0.6	0.6	4.4

Iteración 2

\hat{f}^2	$\lambda * \hat{f}^2$	\hat{f}	r
8.7	0.87	1.67	8.33
2.5	0.25	0.35	2.65
1.2	0.12	0.52	1.48
...
1.2	0.12	0.32	0.68
5.1	0.51	1.11	3.89

Boosting básico

La clave es que boosting **aprende lento**. Cada modelo predice la parte no predicha hasta ese momento (pero de a poco)

Detalles:

- Cada árbol suele ser pequeño y con pocos cortes
- El hiperparámetro λ (*learning rate*) retarda el aprendizaje
- Difícil de paralelizar: cada árbol nuevo adicional que se entrena depende de lo que los anteriores predijeron

Estructura de la clase

- Repaso
- Boosting básico
- XGBoost

XGBoost

Extreme Gradient Boosting (XGBoost) es un modelo (y librería) usado para resolver problemas de aprendizaje supervisado; i.e.: predecir el valor de y_i dado de x_i (donde $x_i \in R^d$)

La función que optimiza tiene dos componentes: uno de ajuste en entrenamiento y otro de regularización:

$$Obj(\Theta; X, y) = L(\Theta; X, y) + \Omega(\Theta)$$

El costo por mal ajuste en entrenamiento mide qué tan mal un modelo, con parámetros Θ , ajusta los datos de entrenamiento. Suele expresarse como una función del error de predicción de cada observación, i.e.: $L(\Theta; X, y) = \sum_{i=1}^n l(y_i, \hat{y}_i(\Theta; X, y))$

- l del error cuadrático: $(y_i - \hat{y}_i)^2$
- l de *cross-entropy*: $-y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$

XGBoost

El componente de regularización ($\Omega(\theta)$) penaliza la **complejidad del modelo**

El valor de trade-off λ es un hiperparámetro que asigna un **peso relativo** a los dos componentes de pérdida

Distintas combinaciones de $L(\theta; X, y)$ y $\Omega(\theta)$ dan lugar a **distintos modelos** de aprendizaje

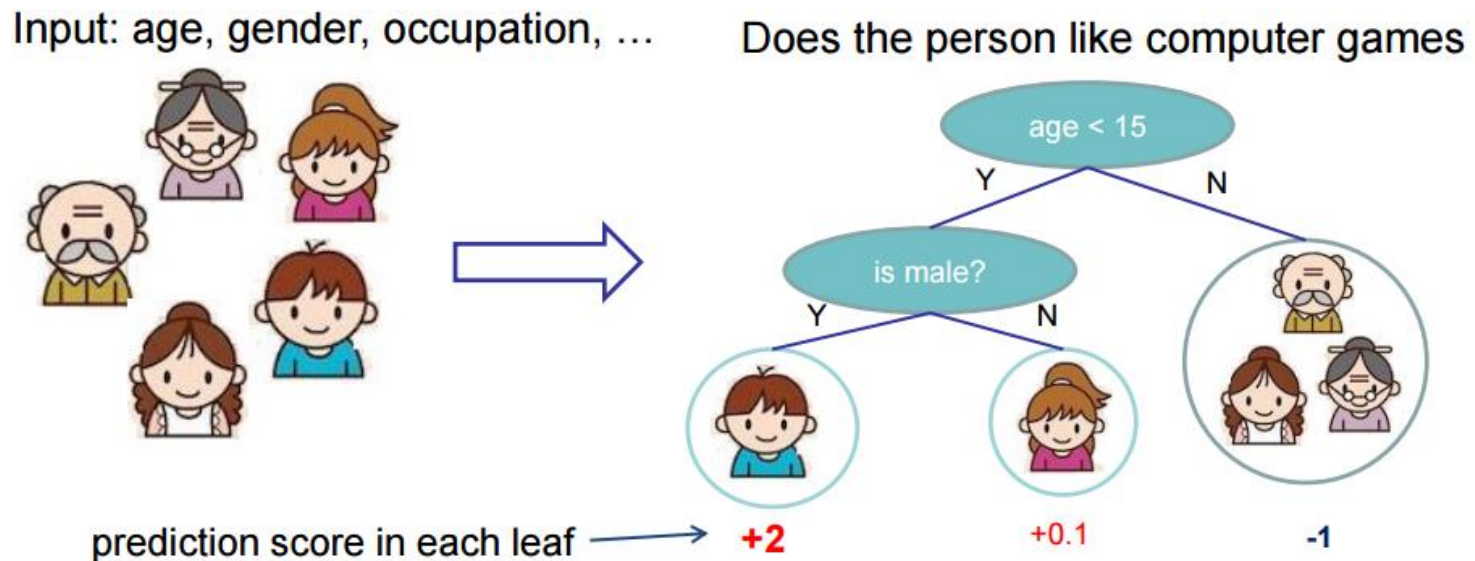
Por ejemplo, si el modelo es lineal (con coeficientes w), se tiene:

- Regresión lineal: $Obj(\theta; X, y; \lambda) = \sum_{i=1}^n (y_i - \hat{y}_i)^2$
- Ridge regression: $Obj(\theta; X, y; \lambda) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \|w\|^2$
- LASSO: $Obj(\theta; X, y; \lambda) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \|w\|_1$

XGBoost

XGBoost aprende **ensambles de árboles de decisión**

En árboles, un **número real** (puntaje) se encuentra asociado a cada hoja

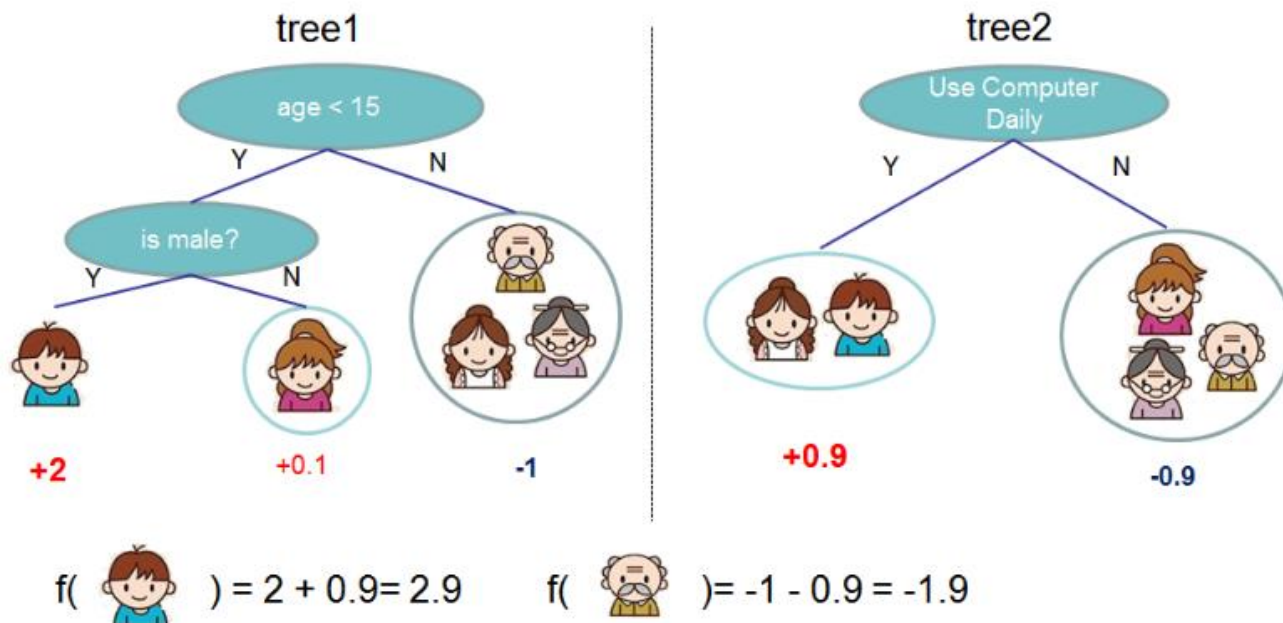


Antes asignábamos la media de y en la hoja, o la proporción de cada clase, **podría no ser así**

XGBoost

Generalmente, un único árbol **no logra captar todas las particularidades** de los datos

Se suele usar un ensamble de árboles. Una opción es tomar como predicción final para una observación la **suma de la predicciones de los modelos base**



XGBoost

La **predicción para cada observación** vendrá dada por la siguiente expresión:

$$\hat{y}_i^{(K)} = \sum_{k=1}^K f_k(x_i), \quad f_k \in \mathcal{F}$$

En donde:

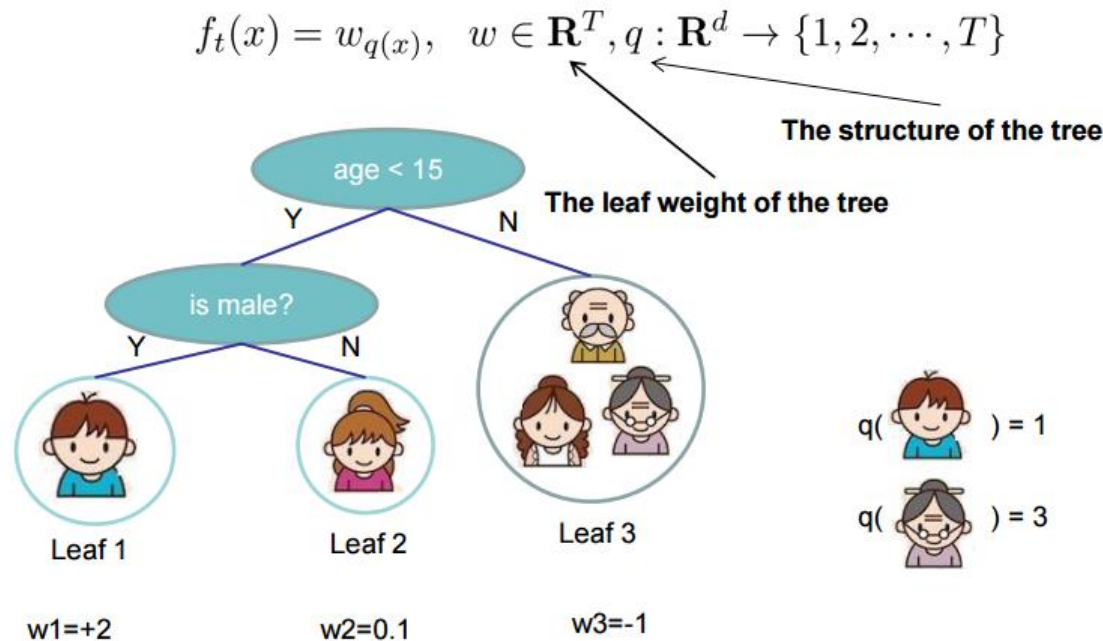
- K es el número de árboles del ensamble
- \mathcal{F} es el conjunto de todos los árboles posibles
- f es una función en el espacio funcional \mathcal{F}

XGBoost

Cada árbol f_t predice sobre la base de la siguiente expresión:

$$f_t(x) = w_{q(x)}, w \in \mathbb{R}^T, q: \mathbb{R}^d \rightarrow \{1, 2, \dots, T\}$$

En donde: 1) w es un vector de puntajes (valores predichos), 2) q es una función que asigna cada observación a una hoja (la estructura del árbol) y 3) T es la cantidad de hojas



XGBoost

Combinando lo anterior se tiene que la función objetivo vendría dada por:

$$Obj^{(K)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(K)}) + \sum_{k=1}^K \Omega(f_k)$$

$$Obj^{(K)} = \sum_{i=1}^n l\left(y_i, \sum_{k=1}^K f_k(x_i)\right) + \sum_{k=1}^K \Omega(f_k)$$

Para cada árbol se debe definir una estructura y para cada hoja de cada árbol un puntaje (**esto es intratable**)

Entrenar todos los árboles simultáneamente es complejo, entonces se opta por **entrenar sucesivamente** uno a la vez

XGBoost

XGBoost hace **boosting**. Sigue una estrategia **secuencial y aditiva** que se basa en la siguiente relación:

$$\hat{y}_i^{(0)} = 0$$

$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + f_1(x_i)$$

$$\hat{y}_i^{(2)} = \hat{y}_i^{(1)} + f_2(x_i)$$

:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) = \sum_{k=1}^t f_k(x_i)$$

XGBoost

¿Supongamos que ya conocemos la estructura del árbol en la iteración t , cómo obtenemos w ? Serán aquellos que **optimicen** $Obj^{(t)}$

$$Obj^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t \Omega(f_k)$$

$$Obj^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + cte$$

Realizando una expansión de Taylor de 2^{do} orden ($f(x + \Delta x) \approx f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$) de l se tiene:

$$Obj^{(t)} \approx \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2] + \Omega(f_t) + cte$$

En donde:

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \quad \text{y} \quad h_i = \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})$$

XGBoost

Después de remover las constantes se tiene que el objetivo en t pasa a aproximarse por:

$$Obj^{(t)} \approx \sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$

Una ventaja de esta expresión es que sólo depende de g_i y de h_i . De modo que pudiendo evaluar la derivada primera y segunda de cualquier función de costo, podemos usar dicha función como objetivo XGBoost (muy dúctil):

E.g.: error cuadrático (regresión):

$$g_i = 2(\hat{y}_i^{(t-1)} - y_i) \quad h_i = 2$$

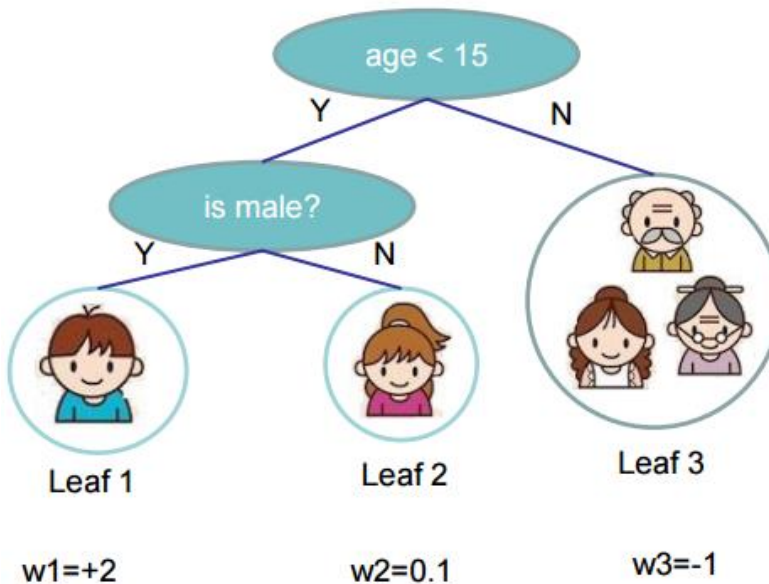
E.g.: *cross-entropy* (clasificación):

$$g_i = -\frac{y_i}{\hat{y}_i} + \frac{(1-y_i)}{(1-\hat{y}_i)} \quad h_i = \frac{y_i}{\hat{y}_i^2} - \frac{(1-y_i)}{(1-\hat{y}_i)^2}$$

XGBoost

Queda ver **qué hacer con el componente de regularización** $\Omega(f_t)$. En XGBoost se asocia la complejidad de f_t con 1) su cantidad de hojas (T) y con 2) el valor de los puntajes que se asigna a las observaciones (w)

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$



$$\Omega = \gamma 3 + \frac{1}{2} \lambda (4 + 0.01 + 1)$$

XGBoost

Uniendo todo en una **misma expresión**:

$$Obj^{(t)} \approx \sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

Reordenando para que las sumatorias queden **expresadas respecto a las hojas**, se tiene:

$$Obj^{(t)} \approx \sum_{j=1}^T \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Llamando G_j a $\sum_{i \in I_j} g_i$ y H_j a $\sum_{i \in I_j} h_i$ se tiene que la expresión anterior se puede escribir como:

$$Obj^{(t)} \approx \sum_{j=1}^T \left[(G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2) \right] + \gamma T$$

XGBoost

Como cada w_j es independiente del resto, para cada hoja, la expresión anterior es una función cuadrática. Tomando la estructura q como fija, se puede encontrar la expresión que el costo

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$






Reemplazando eso en la función objetivo se tiene que:

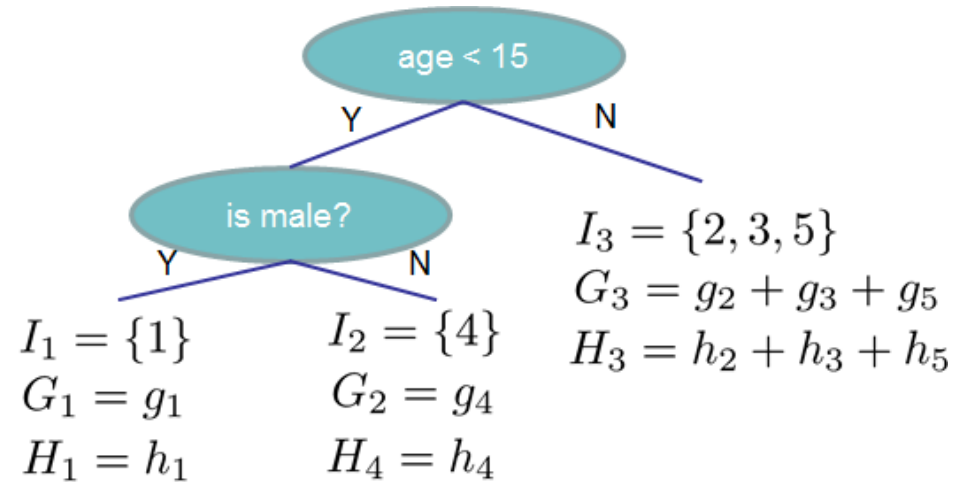
$$Obj^{(t)*} \approx -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

Esta última expresión mide aproximadamente qué tan bueno es el ajuste del modelo a los datos de entrenamiento

XGBoost

Instance index gradient statistics

1		g_1, h_1
2		g_2, h_2
3		g_3, h_3
4		g_4, h_4
5		g_5, h_5



$$Obj = - \sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

Debe remarcarse que **cada w_j puede calcularse fácilmente**

XGBoost

Hasta ahora supusimos la estructura del árbol como fija

¿Cómo la definimos?

Igual que antes (de manera **greedy**):

- Se empieza con un árbol de profundidad 0.
- Para cada split candidato el cambio de la función objetivo vendrá dado por:

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

- Se evalúa la ganancia para cada variable y para cada punto de corte (previamente ordenando de manera creciente la variable candidata), se elige el mejor split y se lo agrega a la estructura. Se repite este paso con la estructura actualizada

XGBoost

¿Cómo trabaja XGBoost los missings?

Sparsity aware split-finding

Por cada variable x_j para la que se evalúan splits, se mide:

1. Qué ganancia máxima se obtendría si los missings fueran el valor más bajo de x_j (o sea, **si siempre fueran a la izquierda**)
2. Qué ganancia máxima se obtendría si los missings fueran el valor más alto de x_j (o sea, **si siempre fueran a la derecha**)

Se dirigen los missings hacia la dirección que genera **mayor ganancia**

XGBoost

Resumen del algoritmo detrás de XGBoost

- Se añade un árbol f_t en cada iteración.
- El comienzo de cada iteración se calcula para observación i los valores de $g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$ y $h_i = \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})$.
- En base a estos valores se construye de manera greedy el árbol f_t guiándose $Obj^{(t)*} \approx -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$.
- Se agrega el nuevo árbol al ensamble ($\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$). Nota: comúnmente no se agregan por completo las predicciones del nuevo árbol sino una proporción de las mismas, de modo que:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \epsilon f_t(x_i)$$

- Manejo de missings mediante *sparsity aware split-finding*

XGBoost

La implementación de XGBoost se encuentra sumamente optimizada, además incorpora otros hiperparámetros de aprendizaje ([link](#)). Típicamente se suele modificar:

- **nrounds**: número de árboles
- **max_depth**: profundidad máxima de los árboles
- **eta**: learning rate
- **gamma**: mínima reducción del error para generar un corte
- **colsample_bytree**: variables a muestrear y considerar en cada árbol
- **min_child_weight**: mínima cantidad de observaciones en los hijos para considerar un corte
- **subsample**: muestreo de observaciones a considerar en cada árbol

XGBoost puede hacer overfitting! En general los parámetros en rojo/azul, a medida que son mayores/menores, más tienden a hacer overfitting

XGBoost

Veamos cómo funciona XGBoost en Python

Bibliografía

- Capítulo 8 (salvo lo referido a BART)
- Tutorial de XGBoost y árboles ([link](#))

Avanzada

- Chen & Guestrin, "*XGBoost: A Scalable Tree Boosting System*" ([link](#))