

FIAP

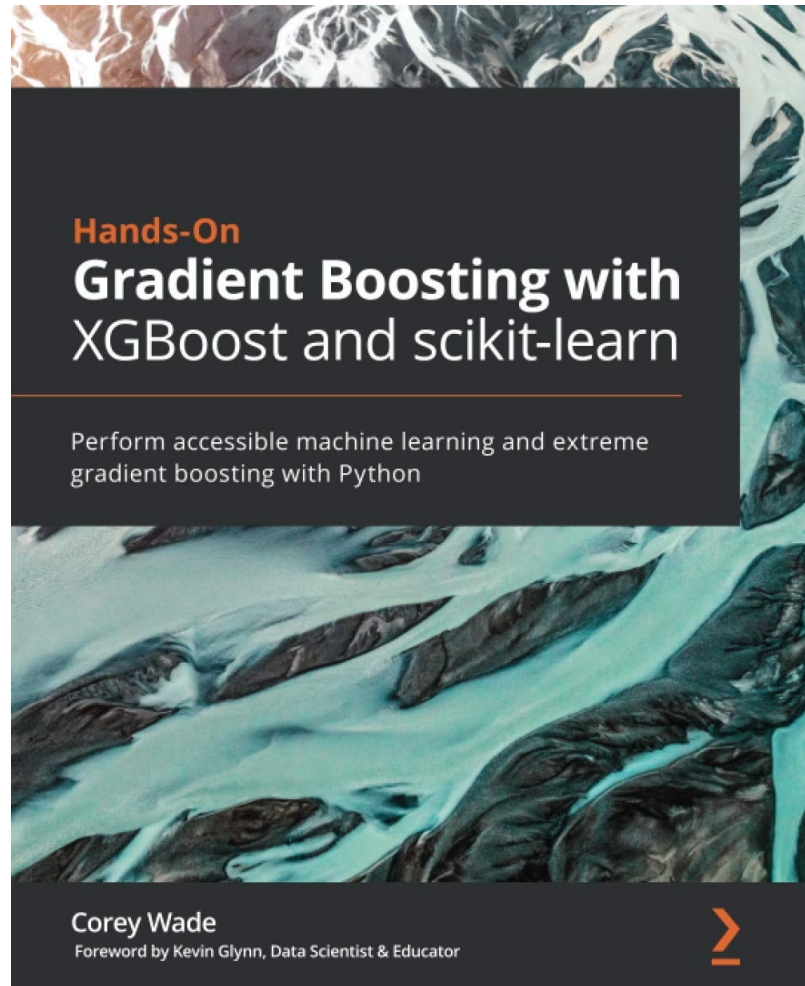
NABA

Agenda

1. XGBoost

1. Definição
2. Premissa
3. Ensemble Methods
4. Intuição
5. Matemática explicada
6. Otimizações

Bibliografía Básica





Definição



Definição

XGBoost é a abreviação para *e**X**treme **G**radient **B**oosting*. É uma biblioteca de software de código aberto comprovada pela indústria que fornece uma estrutura de *Gradient Boosting* para escalar bilhões de pontos de dados rápida e eficientemente.

Nas palavras de Tianqi Chen:

“XGBoost se refere, na verdade, ao objetivo de engenharia de forçar os recursos computacionais para algoritmos de *boosted trees*.”

Nosso objetivo aqui é entender o processo de construção e ajuste do XGBoost tanto para classificação quanto para regressão, seja usando scikit-learn ou a API original para Python.

Vamos usar os hiperparâmetros do XGBoost para melhorar as métricas, corrigir valores faltantes e ajustar *base learners**.

Antes de discutirmos o algoritmo, é importante instalar a biblioteca corretamente. Siga os passos descritos no jupyter notebook.

Premissa

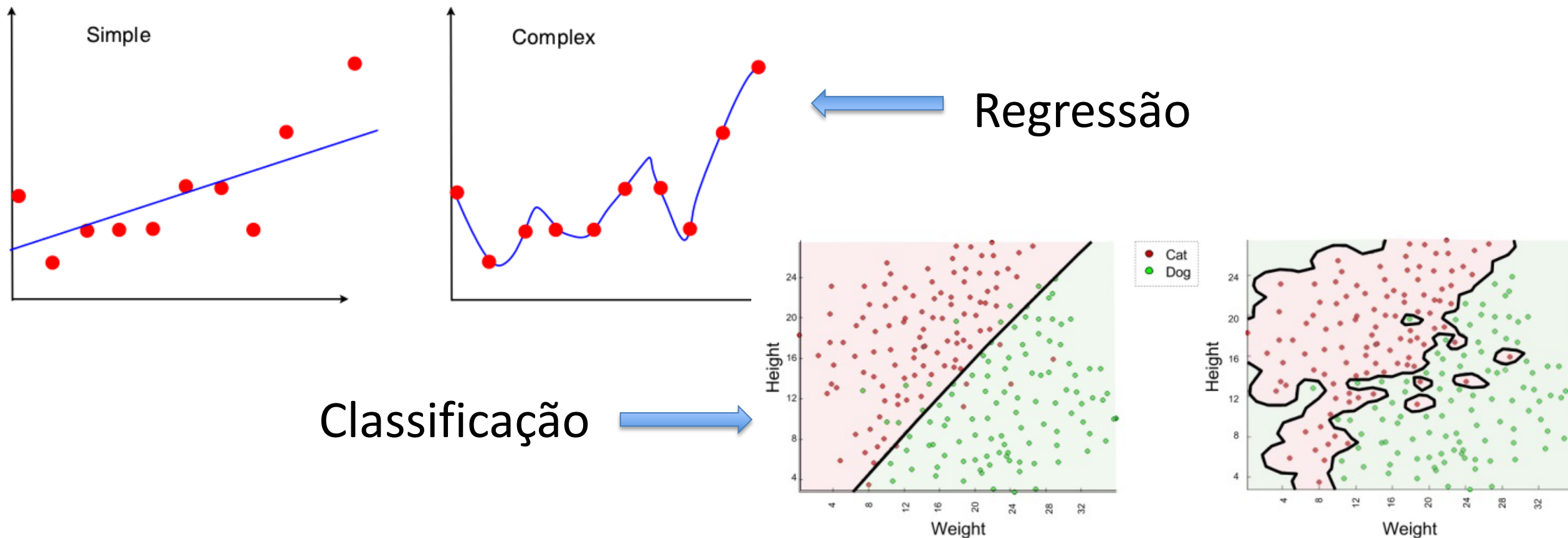


Algumas premissas são importantes para termos um completo entendimento de como o XGBoost funciona.

Anteriormente, vimos o *tradeoff* entre *bias* e *variance*. Além dele, precisamos discutir outro ponto importante que fundamenta a tese do XGBoost: **Regularização**.

Regularização é uma importante técnica usada para prevenir *overfitting*.

Intuitivamente, quando você tem um problema de classificação ou regressão, você pode usar uma função simples ou complexa para ajustar os dados de treinamento:



A ideia consiste, portanto, em **escolher um modelo mais complexo apenas se o aumento sutil da complexidade melhorar significativamente a performance no conjunto de treino.**

Por exemplo: se você está usando funções polinomiais para classificação, o grau do polinômio é a medida de complexidade do modelo. Assim, se um polinômio de grau 2 fornece uma acurácia de 85% e um polinômio de grau 3 fornece uma acurácia de 90%, você vai escolher o polinômio de grau 3, visto que a redução do erro na etapa de treino é substancial para um aumento modesto da complexidade.

Entretanto, se um polinômio de grau 2 fornece uma acurácia de 85% e apenas um polinômio de grau 10 vai fornecer uma acurácia de 90%, o mais adequado é permanecer com o polinômio de grau 2.

Temos, basicamente, dois tipos de regularização, que podem ser usadas tanto para classificação quanto para regressão.

A **regularização L1, ou norma L1, ou Lasso** (para regressão) combate *overfitting* reduzindo os parâmetros a zero, o que torna algumas características obsoletas. Desse modo, L1 pode ser entendida como uma forma de *feature selection*, visto que quando atribuímos 0 ao peso de uma feature, estamos erradicando a significância dessa feature.

Matematicamente, temos a seguinte função:

$$LossFunction = \frac{1}{N} \sum_{i=1}^N (\hat{y} - y)^2 + \lambda \sum_{i=1}^N |\theta_i|$$

Já a **regularização L2, ou norma L2, ou Ridge** (para regressão) combate *overfitting* forçando os pesos a serem pequenos, mas não iguais a zero.

Matematicamente, temos a seguinte função:

$$LossFunction = \frac{1}{N} \sum_{i=1}^N (\hat{y} - y)^2 + \lambda \sum_{i=1}^N |\theta_i|^2$$

É importante salientar que **a L2 não é robusta a outliers**, isto porque o termo quadrático irá expandir a diferença no erro para os outliers. A regularização tentará corrigir isso penalizando os pesos.

As principais diferenças entre L1 e L2 são:

1. L1 penaliza a soma dos valores absolutos dos pesos, enquanto L2 penaliza a soma dos quadrados dos pesos
2. L1 é esparsa, enquanto L2 não.
3. L2 não realiza *feature selection*, visto que os pesos não chegam a 0. Já L1 é usada para *feature selection*.
4. L1 é robusta a outliers, enquanto a L2 não.
5. A decisão de qual usar depende do objetivo do problema.
6. *Elastic net* é uma opção que combina L1 e L2

Ensemble Methods



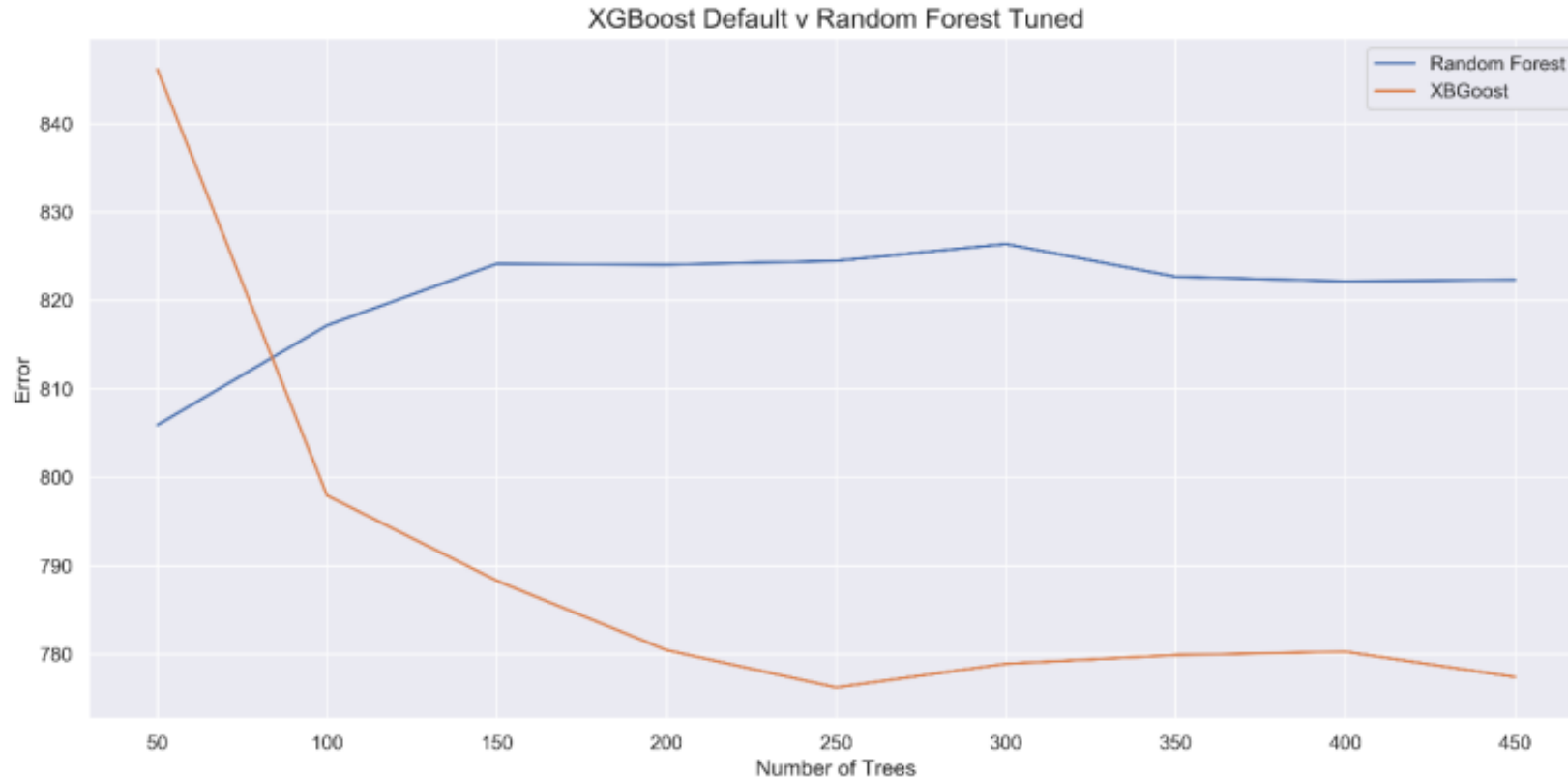
Ensemble Methods

Quando estudamos Random Forest, vimos que existem algumas maneiras de combinar a predição de vários modelos a fim de obter uma predição melhor. Dentre essas maneiras, discutimos os sistemas votantes e, dentro da categoria de ensemble methods, analisamos o bagging, que é a base do RF.

XGBoost usa uma técnica de ensemble também, denominada boosting, que é diferente daquela usada pelo RF.

Ensemble Methods

Ao final do dia, Random Forest é limitada por suas árvores individuais. Se todas elas produzirem o mesmo erro, Random Forest também irá. Mesmo ajustando os hiperparâmetros da melhor maneira possível, Random Forest ainda fica atrás do XGBoost em sua versão padrão.



Ensemble Methods

Assim, precisamos de um ensemble method capaz de **melhorar a partir das deficiências iniciais; um ensemble method que vai aprender a partir dos erros das árvores em futuras iterações**. Boosting foi projetado para aprender a partir dos erros das árvores nas primeiras iterações. Boosting, especialmente o Gradient Boosting, lida com esses tópicos.

A fim de entender as vantagens do XGBoost sobre o tradicional Gradient Boosting, vamos aprender como o Gradient Boosting funciona. A estrutura e hiperparâmetros gerais do Gradient Boosting foram incorporadas no XGBoost.

Ensemble Methods

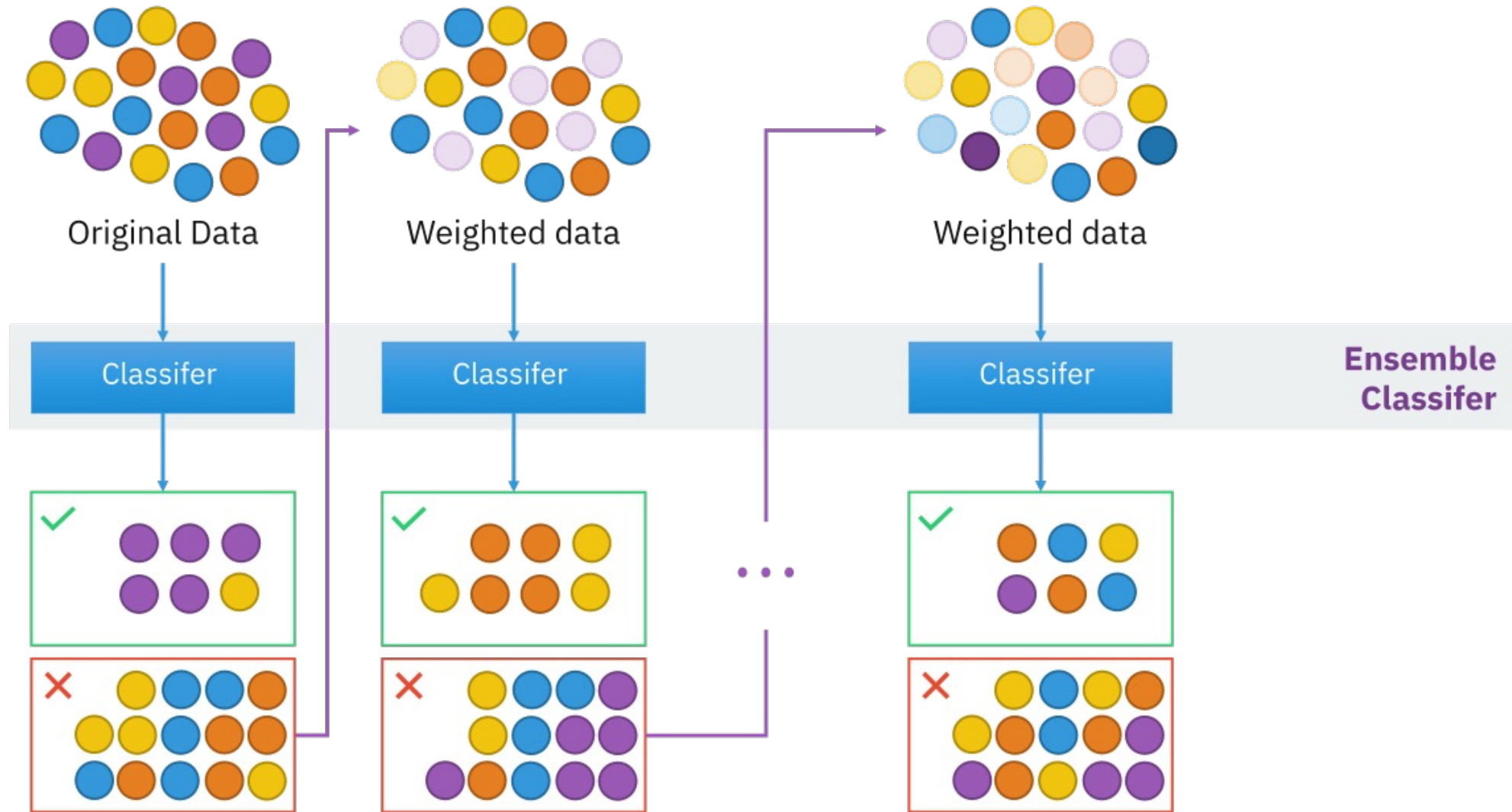
Boosting, em contraste com Bagging, aprende a partir dos erros das árvores individuais. A ideia geral é ajustar novas árvores baseado nos erros das árvores anteriores.

Em Boosting, corrigir erros para uma nova árvores é uma abordagem distinta de Bagging.

Num modelo que usa Bagging, novas árvores não prestam atenção nas anteriores. Além disso, novas árvores são construídas do zero usando bootstrapping, e o modelo final agrega todas as arvores individuais.

Em Boosting, entretanto, cada nova árvore é construída a partir da anterior. As árvores não operam isoladamente; ao invés disso, elas são construídas umas sobre as outras.

Ensemble Methods



Ensemble Methods

A ideia geral por trás dos algoritmos de Boosting é transformar *weak learners* em *strong learners*. Um *weak learner* é um algoritmo de machine learning que dificilmente performa melhor que um “chute”. Já um strong learner aprendeu consideravelmente sobre os dados e performa de maneira suficientemente boa.

Começar com um *weak learner* tem um propósito: construindo modelos nessa ideia, Boosting trabalha focando em correção de erro iterativa e não estabelecendo um poderoso modelo como baseline.

Essa é a maneira que o Gradient Boosting trabalha:

Ensemble Methods

O Gradient Boosting treina cada nova árvore inteiramente baseado nos erros de predição da árvore anterior, isto é, para cada nova árvore, o Gradient Boosting olha para os erros e então constrói uma nova árvore completa ao redor desses erros, ou seja, ele não se preocupa com as predições corretas.

Assim, podemos estabelecer a ideia geral por trás por Gradient Boosting: **calcular os residuals da predição de cada árvore e somar todos os residuals para “scorar” o modelo.**

No código, vamos criar baselines de comparação para ver o real poder do XGBoost. Vamos treinar modelos de regressão e classificação usando Decision Trees e Random Forest para poder compará-los com XGBoost.

Vamos, também, treinar um modelo de Gradient Boosting pra entender o funcionamento do boosting.



Intuição



Intuição

Considere o seguinte simples dataset. Nosso objetivo é prever salário (mil) usando o XGBoost:

Idade	Mestrado?	Salário
23	Não	50
24	sim	70
26	Sim	80
26	Não	65
27	Sim	85

Intuição

Passo 1: Faça uma predição inicial e calcule os resíduos

Essa predição pode ser qualquer valor. Vamos adotar o valor médio das variáveis que queremos prever:

$$\frac{50 + 70 + 80 + 65 + 85}{5} = 70$$

Podemos calcular os resíduos da seguinte maneira:

$$residuals = observed\ values - predicted\ values$$

Intuição

Inserindo o valor dos resíduos

Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Intuição

Passo 2: Construa uma árvore

Cada árvore começa com uma única folha e todos os resíduos vão pra lá

-20, 0, 10, -5, 15

Agora, calculamos o Similarity Score dessa folha (irei detalhar isso à frente):

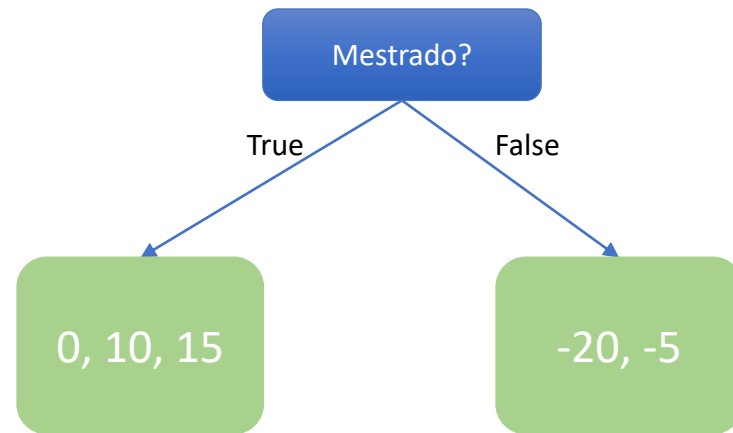
$$\textit{Similarity Score} = \frac{(\textit{sum of residuals})^2}{\#residuals + \lambda} = \frac{(-20+0+10-5+15)^2}{5+1} = 0$$

λ é um parâmetro de regularização que reduz a sensibilidade da predição a pontos individuais e previne overfitting. Será explicado em detalhes. O valor padrão é 1.

Intuição

Passo 2: Construa uma árvore

Agora precisamos verificar se melhoramos o resultado agrupando os resíduos se dividirmos eles em dois grupos usando thresholds baseados em nossos preditores. Vamos começar com Mestrado?



Agora, calculamos o Similarity Score dessas folhas:

$$SS = \frac{(0+10+15)^2}{3+1} = 156.25$$

$$SS = \frac{(-20-5)^2}{2+1} = 208.33$$

Intuição

Passo 2: Construa uma árvore

Agora precisamos quantificar o quão melhor as folhas agrupam os resíduos em relação ao nó raiz. Calculamos o Gain desse split para isso. Caso ele seja positivo, o split deve ser feito.

$$Gain = SS_l + SS_r - SS_{root} = 156.25 + 208.33 - 0 = 364.58$$

Agora é preciso comparar esse Gain com o Gain dos splits usando o preditor Idade.

Intuição

Passo 2: Construa uma árvore

Como Idade é uma variável contínua, precisamos ordenar as linhas de maneira descendente e, depois, calcular a média dos valores adjacentes:

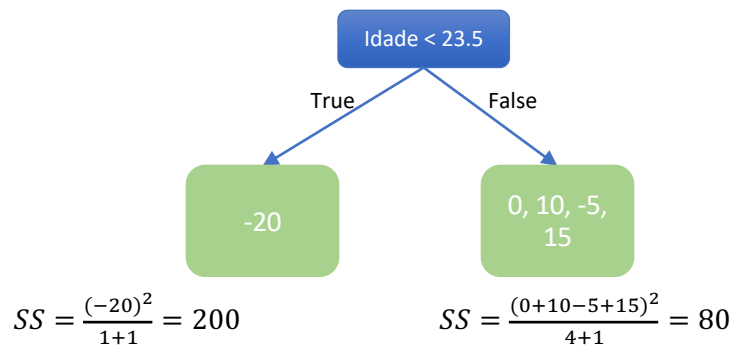
Idade	
23	23.5
24	
26	25
26	26
27	26.5

Agora dividimos os resíduos usando as 4 médias como threshold e calculamos o Gain para cada split.

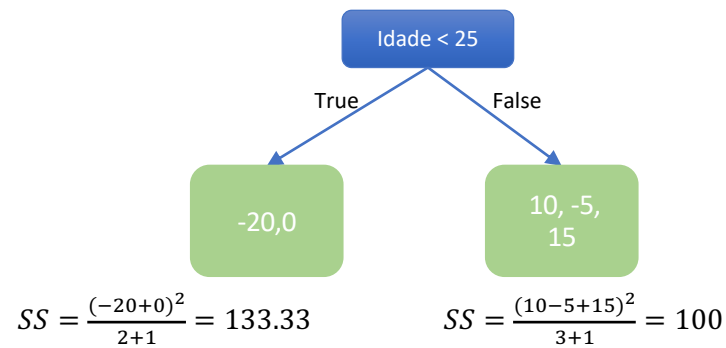
Intuição

Passo 2: Construa uma árvore

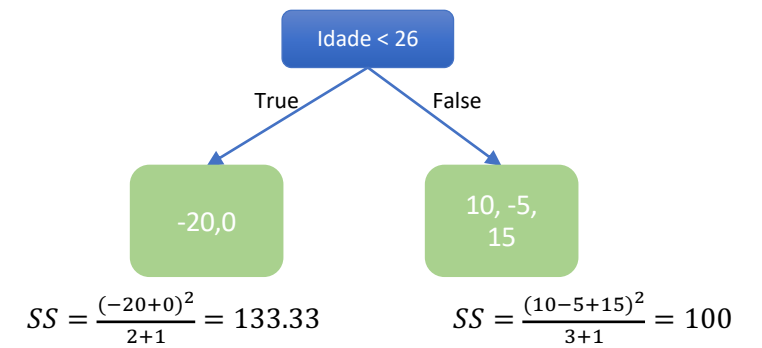
Agora dividimos os resíduos usando as 4 médias como threshold e calculamos o Gain para cada split.



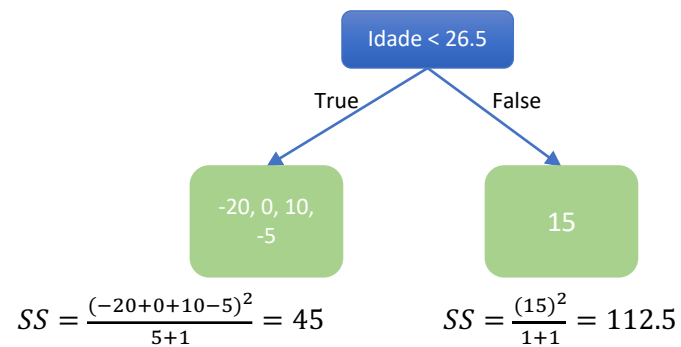
$$\text{Gain} = 200 + 80 - 0 = 280$$



$$\text{Gain} = 133.33 + 100 - 0 = 233.33$$



$$\text{Gain} = 133.33 + 100 - 0 = 233.33$$

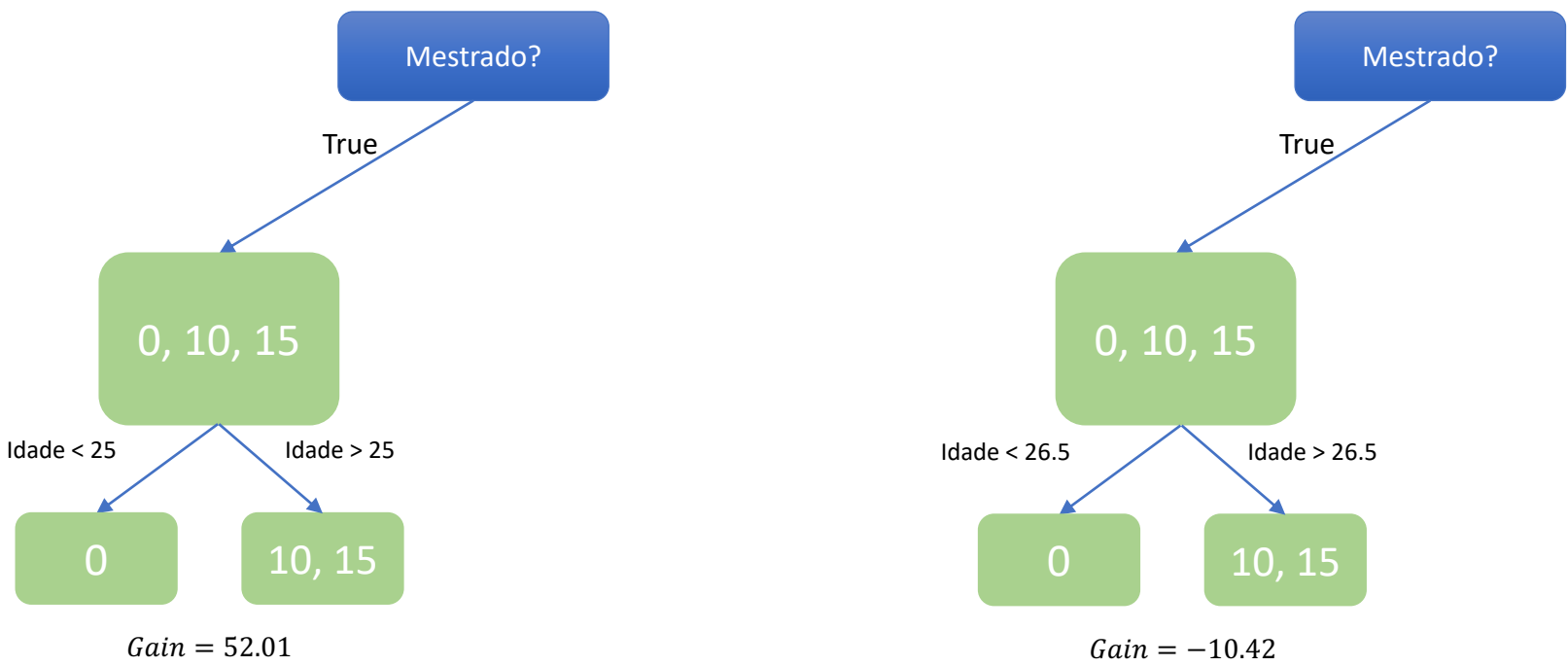


$$\text{Gain} = 45 + 112.5 - 0 = 157.5$$

Intuição

Passo 2: Construa uma árvore

De todos os splits para cada preditor, Mestrado? Possui o maior valor de Gain, então ele será usado como split inicial. Continuamos crescendo nossa árvore olhando, agora, para quando Mestrado? Tem “sim” como resposta e avaliamos os dois possíveis thresholds de Idade:



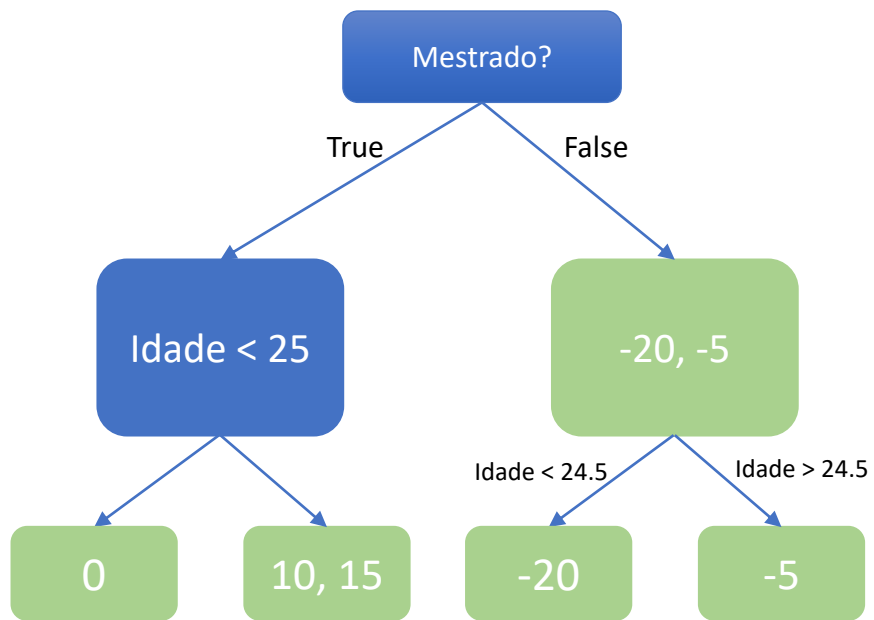
Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Visto que Idade < 25 nos forneceu um valor positivo de Gain, dividimos o nó da esquerda usando esse threshold.

Intuição

Passo 2: Construa uma árvore

Agora olhamos para o nó da direita observando apenas amostras cujo valor de Mestrado? é “não”. Como temos apenas duas amostras nesse nó, o único split possível é quando $\text{Idade} < 24.5$.



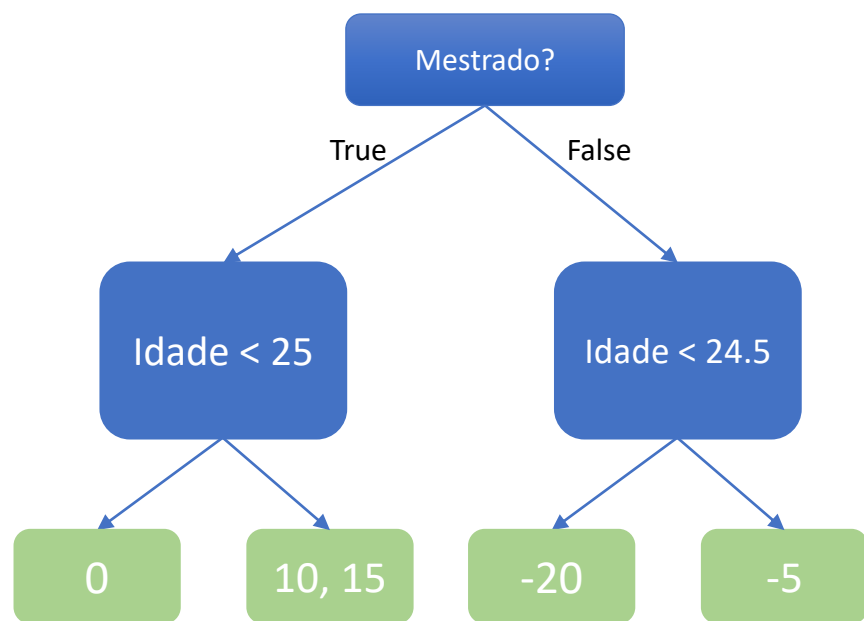
$\text{Gain} = 4.17$

Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

O Gain desse split é positivo, que nos leva à seguinte árvore final:

Intuição

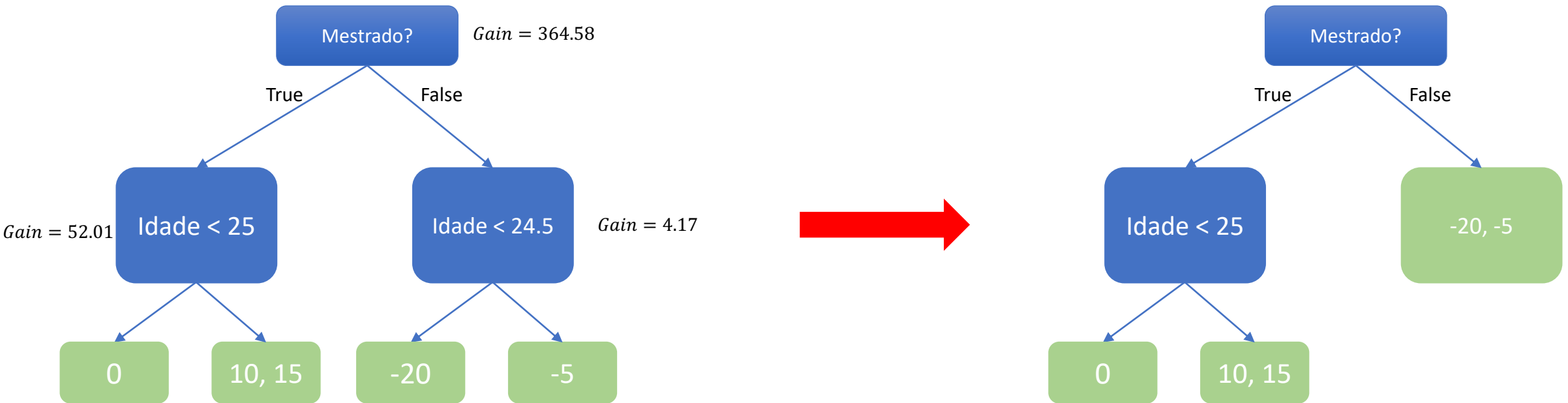
Passo 2: Construa uma árvore



Intuição

Passo 3: Pruning

O objetivo é evitar overfitting. Para isso, começamos das folhas e vamos até a raiz verificando se os splits são válidos ou não. Para estabelecer a validade, usamos γ . Se $Gain - \gamma$ é positivo, mantemos o split. Caso contrário, removemos. O valor padrão é zero, mas a título de didática, vamos usar 50. Nossa árvore possui os seguintes valores de Gain:



Visto que $Gain - \gamma$ é positivo para todos os splits menos $Idade < 24.5$, removemos esse nó.

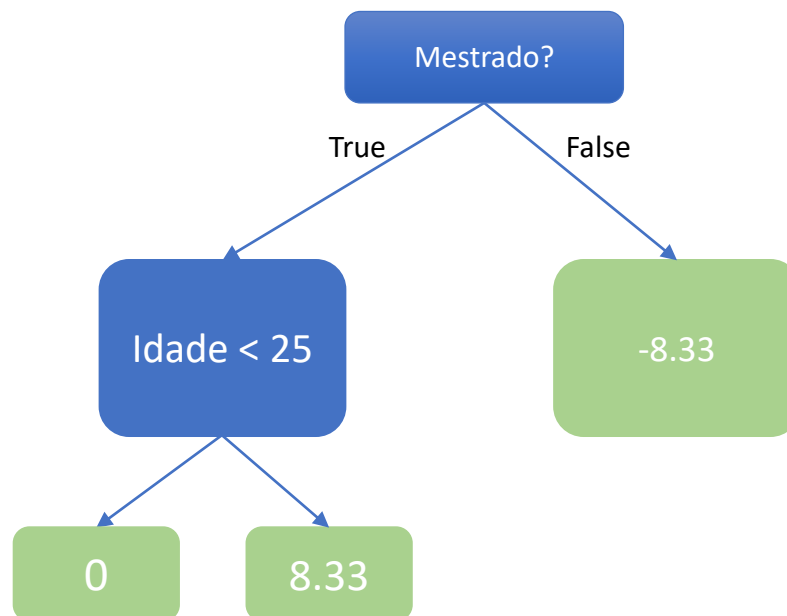
Intuição

Passo 4: Calcular o Output Values das folhas

Estabelecemos um único valor nos nós folhas para fornecer a predição final. Para isso, usamos a seguinte fórmula:

$$\text{Output Value} = \frac{\text{Sum of Residuals}}{\text{Number of Residuals} + \lambda}$$

Usando $\lambda = 1$, obtemos a seguinte árvore final:

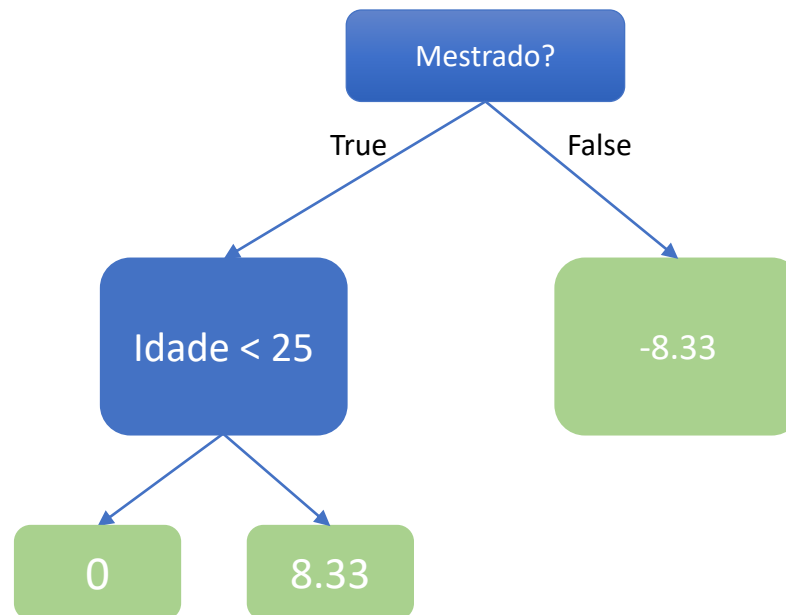


Intuição

Passo 5: Fazer novas previsões

Agora verificamos o quanto nosso modelo melhorou fazendo novas previsões. Para isso, usamos a seguinte fórmula:

$$70 + \varepsilon \times$$



Por padrão, o learning rate = 0.3

Intuição

Passo 5: Fazer novas predições

Calculando os novos valores: $70 + 0.3 \times -8.33 = 67.5$

Idade	Mestrado?	Salário	Valores Preditos
23	Não	50	67.5
24	sim	70	70
26	Sim	80	72.5
26	Não	65	67.5
27	Sim	85	72.5

Intuição

Passo 6: Calcular os residuals usando as novas predições

Idade	Mestrado?	Salário	Residuals
23	Não	50	-17.5
24	sim	70	0
26	Sim	80	7.5
26	Não	65	-2.5
27	Sim	85	12.5

Observamos que os residuals novos são menores que os anteriores, indicando que estamos indo na direção certa. Agora, o processo se resume em repetir os passos 2-6 até que o valor dos residuals seja bem pequeno (próximo a zero) ou o número máximo de iterações seja atingido.

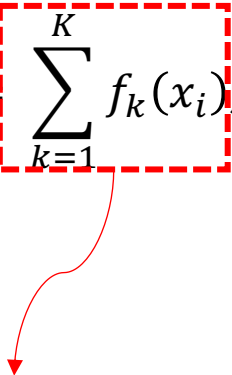
Matemática explicada



Matemática explicada

Considere: $\mathcal{D} = \{(x_i, y_i)\}$ ($|\mathcal{D}| = n, x_i \in \mathcal{R}^m, y_i \in \mathcal{R}$) um dataset com n exemplos e m features. Um modelo de árvore ensemble utiliza k funções aditivas para prever um resultado:

$$\hat{y} = \phi(x) = \sum_{k=1}^K f_k(x_i), f_k \in F$$



Cada árvore é
construída em cima
do erro da anterior
(boosting)

Matemática explicada

Considere: $\mathcal{D} = \{(x_i, y_i)\}$ ($|\mathcal{D}| = n, x_i \in \mathcal{R}^m, y_i \in \mathcal{R}$) um dataset com n exemplos e m features. Um modelo de árvore ensemble utiliza k funções aditivas para prever um resultado:

$$\hat{y} = \phi(x) = \sum_{k=1}^K f_k(x_i), f_k \in F$$

Família de
funções CART



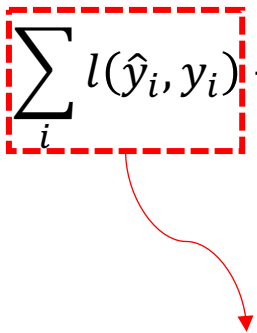
$$F = \{f(x) = w_q(x)\} (q : \mathcal{R}^m \rightarrow T, w \in \mathcal{R}^T)$$

- w = score de cada folha
- q = estrutura de cada árvore que mapeia uma amostra a um nó folha
- T = # de folhas na árvore

Matemática explicada

Para aprender o conjunto de funções usado nesse modelo, minimizamos o seguinte objetivo regularizado:

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



Funcao convexa
diferenciável = *loss*
function

Matemática explicada

Para aprender o conjunto de funções usado nesse modelo, minimizamos o seguinte objetivo regularizado:

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

Termo de regularização:
mede a complexidade
das árvores = evita
overfitting



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

Matemática explicada

O XGBoost é treinado de maneira aditiva. Formalmente, seja $\hat{y}_i^{(t)}$ a predição da i — *ésima* instância na t — *ésima* iteração. Precisamos adicionar f_t para minimizar a seguinte função objetivo:

$$\mathcal{L}^t = \sum_{i=1}^n l\left(\underbrace{y_i}_{\text{Predição anterior}} + \underbrace{f_t(x_i)}_{\text{Predição atual}}\right) + \underbrace{\Omega(f_t)}_{\text{Termo de Regularização}}$$

Vamos analisar a *loss function* e o termo de regularização de maneira separada:

Matemática explicada

Começaremos pelo termo de regularização:

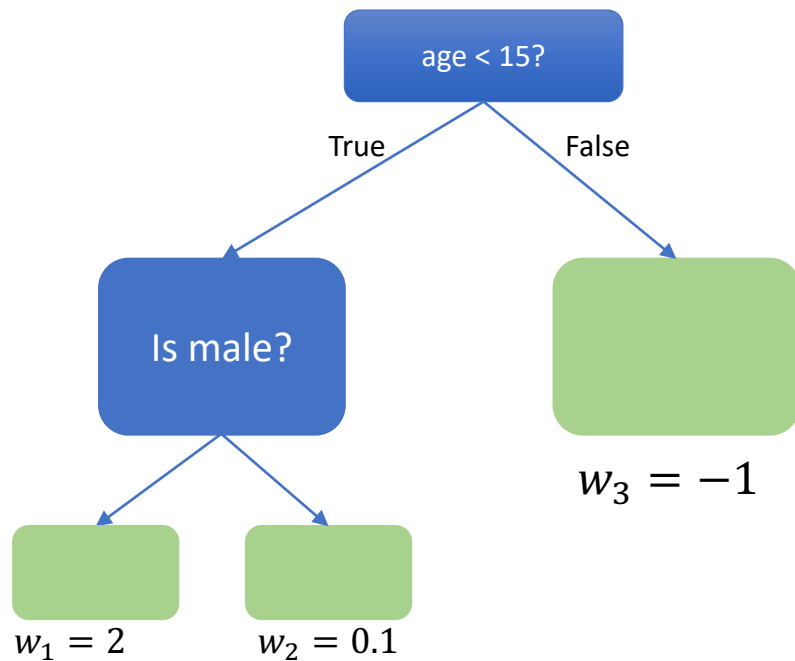
$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

γ = redução mínima requirida na *loss* para fazer um novo split num nó de uma árvore. Varia de $[0, \infty]$ (padrão = 0)

λ = termo de regularização L2 nos pesos (scores).

Matemática explicada

Considere o seguinte exemplo:



$$\begin{aligned}\Omega &= \gamma 3 + \frac{1}{2} \lambda (2^2 + 0.1^2 + (-1)^2) \\ &= \gamma 3 + \frac{1}{2} \lambda (4 + 0.01 + 1) = \gamma 3 + \frac{1}{2} \lambda (5.01)\end{aligned}$$

Vamos entender a *loss function* agora.

Matemática explicada

No XGBoost, exploramos muitos *base learners* e escolhemos uma função que minimiza a *loss*. Existem dois problemas com essa abordagem:

1. Explorar diferentes *base learners*
2. Calcular o valor da *loss function* para todos eles

XGBoost usa a Série de Taylor para aproximar o valor da *loss function* para um *base learner*, reduzindo a necessidade de calcular a *loss* exata para todos os diferentes possíveis *base learners*.

A Série de Taylor pode ser definida da seguinte forma:

$$f(a + h) = f(a) + f'(a)h + \frac{1}{2}f''(a)h^2 + \dots + f^n(a)\frac{h^n}{n!}$$

Matemática explicada

Continuando na Série de Taylor

$$f(a + h) = \boxed{f(a)} + \boxed{f'(a)h} + \boxed{\frac{1}{2}f''(a)h^2} + \dots + f^n(a)\frac{h^n}{n!}$$

Em que:

$$a = \hat{y}_i^{(t-1)}$$

$$h = f_t(x_i)$$


$$f(a) = l(y_i, \hat{y}_i^{(t-1)})$$

Portanto:

$$\mathcal{L}^t = \sum_{i=1}^n \boxed{l(y_i, \hat{y}_i^{(t-1)})} + \boxed{\left(\frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}} \right) f_t(x_i)} + \boxed{\left(\frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)^2}} \right) f_t(x_i)^2}$$

Matemática explicada

Continuando na Série de Taylor

$$\mathcal{L}^t = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)}) + \left(\frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}} \right) f_t(x_i) + \left(\frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)^2}} \right) f_t(x_i)^2$$


Este termo é constante em
relação a qualquer função

Chamando a derivada de primeira ordem de g_i e a derivada de segunda ordem de h_i (ambas com respeito as previsões da iteração anterior) e eliminando os termos constantes, obtemos a seguinte função objetivo simplificada no passo t :

$$\mathcal{L}^t = \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

Isso nos ajuda a resolver o problema 2, mas ainda temos o problema de explorar diferentes *base learners*.

Matemática explicada

Considere que cada árvore f_t possui k nós folhas; I_j é o conjunto de instâncias que pertencem ao nó j e w_j as previsões (scores) para o nó j :

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

$$\mathcal{L}^t = \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \gamma K + \frac{1}{2} \lambda \sum_{j=1}^K w_j^2$$

$$\mathcal{L}^t = \sum_{j=1}^k \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma K$$

Matemática explicada

Para cada folha j , obtenha a derivada com respeito a w_j e iguale a zero (minimização):

$$\frac{\partial \mathcal{L}^t}{\partial w_j^*} = 0$$

$$\sum_{i \in I_j} g_i + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) 2w_j^* = 0 \quad \longrightarrow \quad - \sum_{i \in I_j} g_i = \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^* \quad \longrightarrow \quad w_j^* = \frac{- \sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Matemática explicada

Substituindo $w_j^* = \frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$ aqui $\mathcal{L}^t = \sum_{j=1}^k \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma K$ Obtemos:






$$\mathcal{L}^t = \sum_{j=1}^k \left[\left(\sum_{i \in I_j} g_i \right) \frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) \left(\frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \right)^2 \right] + \gamma K =$$

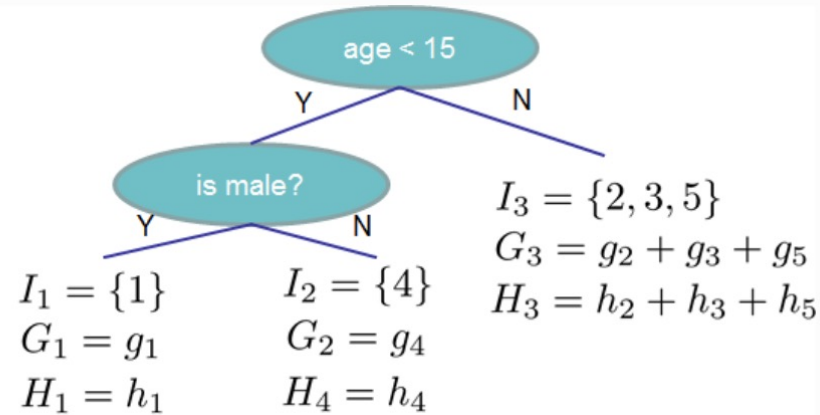
$$\mathcal{L}^t = -\frac{1}{2} \sum_{j=1}^K \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma K$$

Está é equação que fornece o melhor valor de *loss* para um *base learner* com k nós. Dito de outra forma, esta equação mensura quão boa a estrutura da árvore $q(x)$ é. Observe o seguinte exemplo:

Matemática explicada

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

Aqui, considere $G_j = \sum_{i \in I_j} g_i$ e
 $H_j = \sum_{i \in I_j} h_i$

Matemática explicada

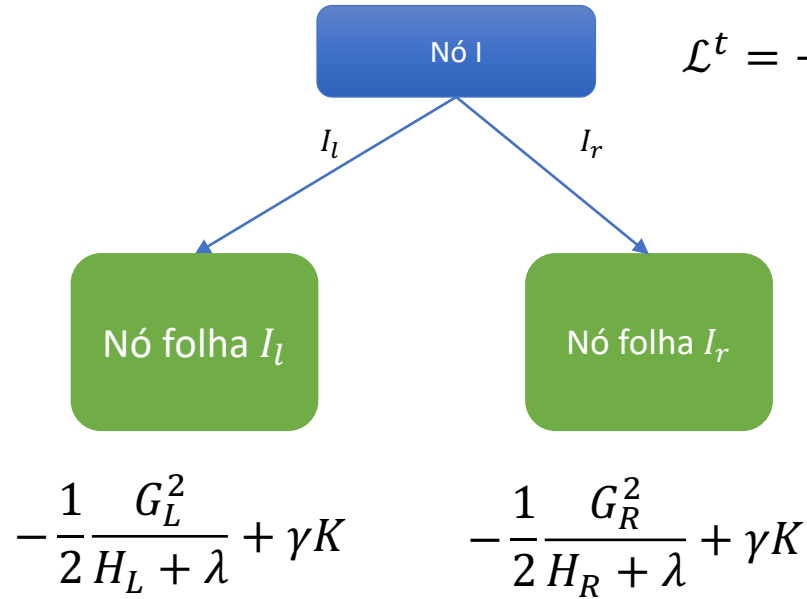
A equação $\mathcal{L}^t = -\frac{1}{2} \sum_{j=1}^K \frac{\left(\sum_{i \in I_j} g_i\right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma K$ nos fornece o valor ótimo de *loss* para uma estrutura fixa de árvore

Idealmente, deveríamos enumerar todas as árvores possíveis e escolher a melhor. Entretanto, isto é impraticável.

Recordando o que já vimos, usamos a Série de Taylor para facilitar o cálculo da *loss*, conseguimos determinar os valores ótimos do score num nó folha e agora precisamos determinar uma maneira de explorar todas as diferentes possíveis estruturas de árvores.

Vamos ver o seguinte exemplo:

Matemática explicada



$$\mathcal{L}^t = -\frac{1}{2} \sum_{j=1}^K \frac{\left(\sum_{i \in I_j} g_i\right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma K = -\frac{1}{2} \frac{G^2}{H + \lambda} + \gamma K$$

Anteriormente, vimos que, se $Gain - \gamma$ é positivo, realizamos o split. Caso contrário, podemos a árvore. Para isso, verificamos a *loss* antes do split e a *loss* depois do split.

Antes do split: $-\frac{1}{2} \frac{G^2}{H + \lambda} + \gamma K = -\frac{1}{2} \left[\frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] + \gamma K$

Depois do split: $-\frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} \right] + \gamma K$

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

Otimizações



Otimizações

A diferença mais importante entre XGBoost para o Gradiente Boosting é que a performance foi amplificada através de várias melhorias na abordagem algorítmica. Podemos elencar as principais contribuições:

1. Approximate Greedy Algorithm
2. Parallel Learning
3. Weighted Quantile Sketch
4. Sparsity-Aware Split Finding
5. Cache-Aware Access
6. Compressed Sparse column (CSC) data format

Otimizações

Discutimos anteriormente que um modelo baseado em árvores tem como principal tarefa para obter a melhor performance encontrar o melhor split para fazer uma clara distinção entre as amostras. Bom, uma maneira de fazer isso é avaliar todos os possíveis splits, calcular seus scores e escolher o melhor. Este é o método usado em modelos de boosting e é chamado de *Basic Exact Greedy Aalgorithm*, definido da seguinte maneira:

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I , instance set of current node

Input: d , feature dimension

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I , by \mathbf{x}_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split with max score

Otimizações

Entretanto, como vimos, isso se torna impraticável para grandes datasets. Por isso, XGBoost usa um algoritmo aproximado para realizar o split, o que torna ele bastante rápido. O *Approximate Greedy Algorithm* é definido da seguinte forma:

Algorithm 2: Approximate Algorithm for Split Finding

```
for  $k = 1$  to  $m$  do
    | Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature  $k$ .
    | Proposal can be done per tree (global), or per split(local).
end
for  $k = 1$  to  $m$  do
    |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$ 
    |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$ 
end
Follow same step as in previous section to find max
score only among proposed splits.
```

Vamos entender seu funcionamento:

Otimizações

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ **to** m **do**

 Propose $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$ by percentiles on feature k .
 Proposal can be done per tree (global), or per split(local).

end

for $k = 1$ **to** m **do**

$G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
 $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

end

Follow same step as in previous section to find max score only among proposed splits.

Para da uma das features

Otimizações

Algorithm 2: Approximate Algorithm for Split Finding

```
for  $k = 1$  to  $m$  do
  | Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature  $k$ .
  | Proposal can be done per tree (global), or per split(local).
end
for  $k = 1$  to  $m$  do
  |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$ 
  |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$ 
end
```

Follow same step as in previous section to find max score only among proposed splits.

Divida o range das features em percentis.

Esta abordagem irá dividir a distribuição da feature x em n buckets.

Imagine que queremos dividir x em 10 buckets diferentes. Visto que estamos usando percentis, cada bucket terá um número idêntico de amostras. Ex: [1 -> 10], [11 -> 20], etc.

Otimizações

Algorithm 2: Approximate Algorithm for Split Finding

```
for  $k = 1$  to  $m$  do
    Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature  $k$ .
    Proposal can be done per tree (global), or per split(local).
```

```
end
```

```
for  $k = 1$  to  $m$  do
```

```
     $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$ 
     $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$ 
```

```
end
```

```
Follow same step as in previous section to find max
score only among proposed splits.
```

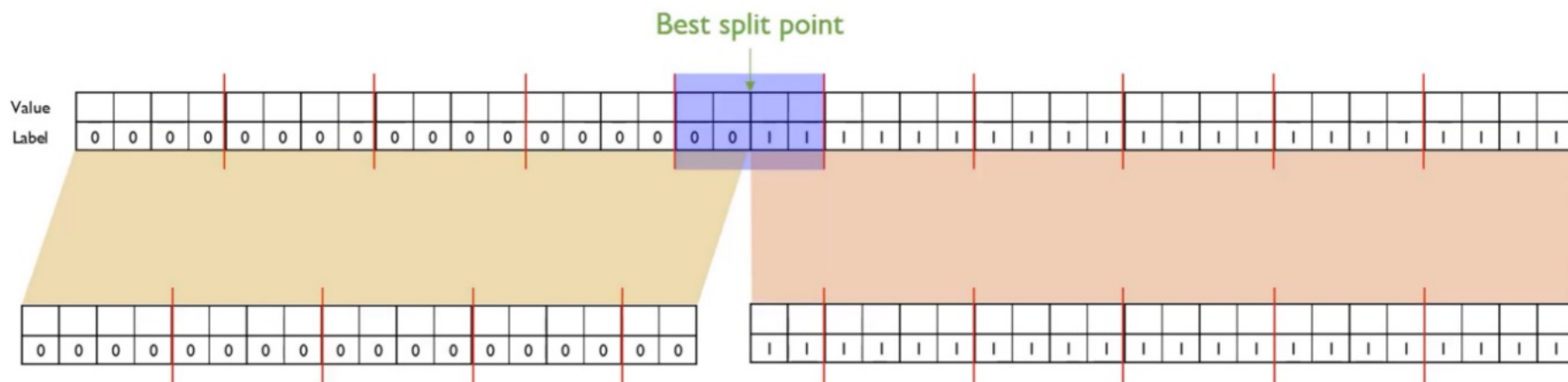
Calculo G e H para a feature K
para cada bucket.

Aqui, entra mais uma
otimização que o XGBoost
implementou: **parallel learning**.
O cálculo do score para cada
bucket pode ser realizado de
forma paralela.

Otimizações

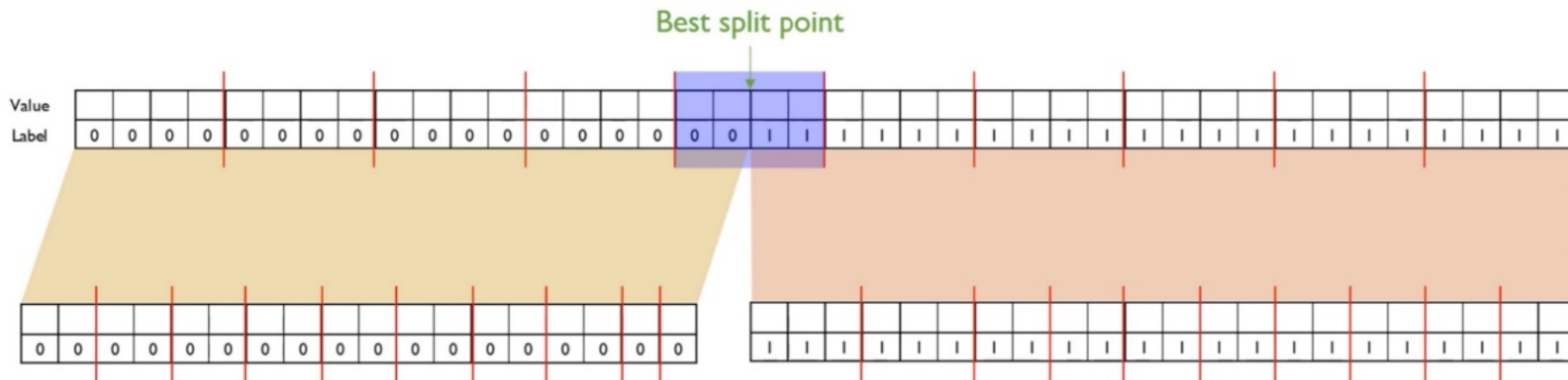
Vimos que a criação dos buckets pode ser feita através de dois métodos diferentes: global e local. Isto significa que essa criação pode ser feita uma vez para a árvore toda ou toda vez quando ocorre um split.

Quando ocorre pelo método global, o nó pai será dividido em dois nós filhos a partir do melhor ponto para split e os pontos de criação dos buckets serão mantidos, conforme vemos imagem abaixo:



Otimizações

Por outro lado, sempre quando ocorre um split local, a criação do bucket acontece novamente. Ex: se o pai tinha 10 buckets, cada filho também terá 10 buckets, conforme vemos abaixo:



O número de buckets é definido pelo hiperparametro eps . Por default, $eps = 0.03$

$$\frac{1}{eps} = \text{número de pontos candidatos}. \frac{1}{0.03} = 33$$

Otimizações

Mas o XGBoost promoveu mais uma otimização quando se trata de encontrar o melhor split. Ao invés de usar percentis, quartis são usados. A ideia permanece a mesma, mas isso acelera ainda mais o processo de calcular o melhor ponto para split .

Podemos resumir o processo da seguinte maneira:

1. Divida um grande dataset em datasets menores; execute-os em paralelo para encontrar um histograma aproximado do dataset.
2. Use o histograma aproximado para aproximar os quantis. Usando *eps*, crie *n* buckets para armazenar cada quartil.

Otimizações

Podemos elencar as principais contribuições:

- ~~1. Approximate Greedy Algorithm~~
- ~~2. Parallel Learning~~
- ~~3. Weighted Quantile Sketch~~
4. Sparsity-Aware Split Finding
5. Cache-Aware Access
6. Compressed Sparse column (CSC) data format

Vamos entender como o XGBoost lida com datasets esparsos agora.

Otimizações

No mundo real, os datasets são sempre esparsos, isto é, com muitas entradas nulas ou com valor zero. Esparsidade pode ser gerada também como resultado de um processo de feature engineering.

XGBoost também lida com dados esparsos de uma maneira bem eficiente. Vejamos o psedo-código:

Algorithm 3: Sparsity-aware Split Finding

Input: I , instance set of current node

Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

Input: d , feature dimension

Also applies to the approximate setting, only collect statistics of non-missing entries into buckets

$\text{gain} \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

// enumerate missing value goto right

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I_k , ascent order by \mathbf{x}_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

// enumerate missing value goto left

$G_R \leftarrow 0, H_R \leftarrow 0$

for j in sorted(I_k , descent order by \mathbf{x}_{jk}) **do**

$G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split and default directions with max gain

Otimizações

Algorithm 3: Sparsity-aware Split Finding

Input: I , instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d , feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
 $\text{gain} \leftarrow 0$
 $G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ **to** m **do**
 // enumerate missing value goto right
 $G_L \leftarrow 0, H_L \leftarrow 0$
 for j in sorted(I_k , ascent order by \mathbf{x}_{jk}) **do**
 $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$
 $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$
 $\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$
 end
 // enumerate missing value goto left
 $G_R \leftarrow 0, H_R \leftarrow 0$
 for j in sorted(I_k , descent order by \mathbf{x}_{jk}) **do**
 $G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$
 $G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$
 $\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$
 end
end
Output: Split and default directions with max gain

Iniciamos da mesma forma que o algoritmo usado para fazer o melhor split. Entretanto, agora, para calcular as estatísticas (G e H), usamos somente amostras não nulas

Otimizações

Algorithm 3: Sparsity-aware Split Finding

Input: I , instance set of current node

Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

Input: d , feature dimension

Also applies to the approximate setting, only collect statistics of non-missing entries into buckets

$\text{gain} \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

// enumerate missing value goto right

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I_k , ascent order by x_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

// enumerate missing value goto left

$G_R \leftarrow 0, H_R \leftarrow 0$

for j in sorted(I_k , descent order by x_{jk}) **do**

$G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split and default directions with max gain

Considere o seguinte exemplo:

Value	1.3		1.1	0.2		1.9	0.5		1.5	1.8
Class	1	0	1	0	0	1	0	0	1	1



Todos os valores faltantes vão para a direita quando o split é executado. No caso, o melhor split é obtido no seguinte ponto:

Value	0.2	0.5	0.8	1.1	1.3	1.5	1.9			
Class	0	0	1	1	1	1	1	0	0	0

Otimizações

Algorithm 3: Sparsity-aware Split Finding

Input: I , instance set of current node

Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

Input: d , feature dimension

Also applies to the approximate setting, only collect statistics of non-missing entries into buckets

$\text{gain} \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

// enumerate missing value goto right

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I_k , ascent order by x_{jk}) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

// enumerate missing value goto left

$G_R \leftarrow 0, H_R \leftarrow 0$

for j in sorted(I_k , descent order by x_{jk}) **do**

$G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split and default directions with max gain

Considere o seguinte exemplo:

Value	1.3		1.1	0.2		1.9	0.5		1.5	1.8
Class	1	0	1	0	0	1	0	0	1	1



Repita o processo, agora levando os missing values para a esquerda, gerando o seguinte split:

Value				0.2	0.5	0.8	1.1	1.3	1.5	1.9
Class	0	0	0	0	0	1	1	1	1	1

Como podemos ver, esse split gerou um split melhor ele será usado para toda feature em questao

Otimizações

Podemos elencar as principais contribuições:

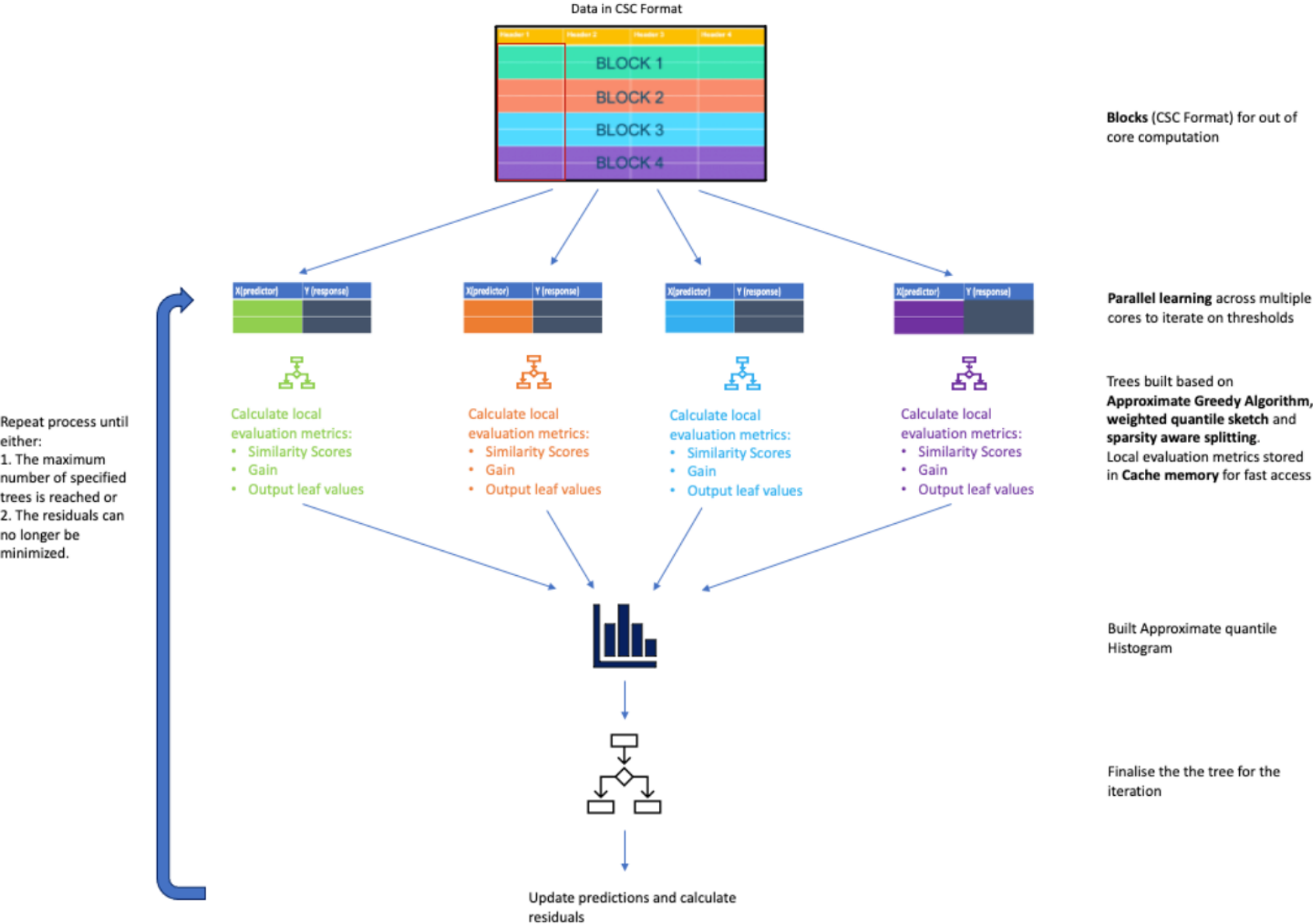
- ~~1. Approximate Greedy Algorithm~~
- ~~2. Parallel Learning~~
- ~~3. Weighted Quantile Sketch~~
- ~~4. Sparsity Aware Split Finding~~
5. Cache-Aware Access
6. Compressed Sparse column (CSC) data format

Otimizações

Para rapidamente calcular a primeira e segunda (G e H) derivadas na etapa de treino do modelo, o XGBoost usa a memória cache.


Por fim, quando os dados são grandes demais para caber em memória, XGBoost usa um formato comprimido dos dados em disco para acelerar a leitura e, quando mais de um HD está disponível (Parallel Learning), o acesso é feito a cada um deles, otimizando o tempo. O formato usado é o CSC (*Compressed Sparse Column*)

Otimizações - Resumindo



Obrigado!

profdheny.fernandes@fiap.com.br

 /dhenyfernandes

FIAP MBA⁺

Copyright © 2022 | Professor Dheny R. Fernandes

Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente proibido sem consentimento formal, por escrito, do professor/autor.

FIAP