

FIAP

NABA

Agenda

1. **LightGBM**

1. Definição
2. Intuição
3. Otimizações
 1. GOSS
 2. EFB
4. Implementação

Bibliografia Básica

Artigo: pode ser encontrado [aqui](#)

Documentação oficial: [link](#)



Definição



Definição

LightGBM é um framework de *gradient boosting* que usa algoritmos de aprendizado baseados em árvores. Foi projetado para ser distribuído e eficiente seguindo as seguintes propriedades:

- Tempo de treinamento mais rápido e alta eficiência
- Menor uso de memória
- Melhor acurácia
- Suporte a aprendizado paralelo, distribuído e em GPU
- Capaz de lidar com dados em larga escala

Definição

Apesar de muitas otimizações terem sido feitas nos algoritmos baseados em GBDT (*Gradient Boosting Decision Tree*), **eficiência e escalabilidade** ainda estão insatisfatórias quando a dimensão das features é alta o dataset é grande.

Uma razão principal, como vimos em XGBoost, é que, para cada feature, é necessário percorrer todas as instâncias para estimar o ganho de informação de todos os possíveis splits, o que é muito custoso.

Para lidar com esse problema, LightGBM propõe duas novas técnicas:

- *Gradient-based On-Side Sampling (GOSSS)*
- *Exclusive Feature Bundling (EFB)*

Ambos serão explicados em detalhes futuramente.

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

Intuição



Intuição

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

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 usando o MSE:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

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

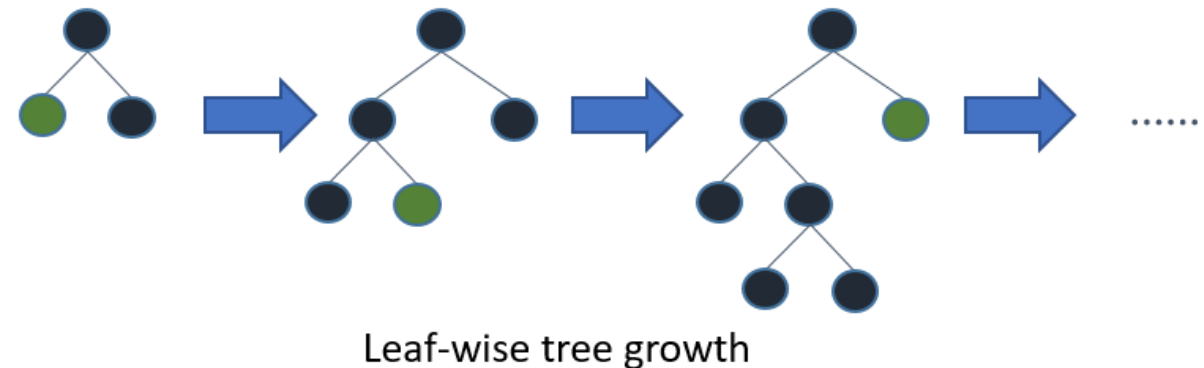
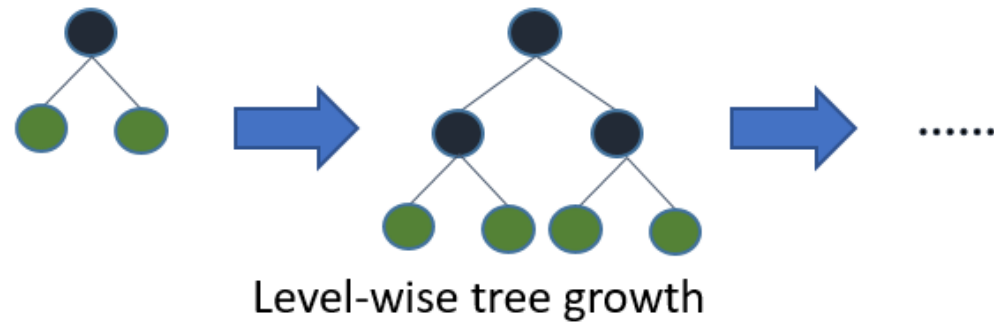
MSE = 150

Intuição

Passo 2: Cresça uma árvore

LightGBM cresce as árvores de uma maneira diferentes que o XGBoost. Enquanto este usa level-wise, LightGBM a técnica leaf-wise. A diferença está na ordem em que as árvores crescem, mas caso a árvore seja crescida completamente, ambas as abordagens resultarão na mesma árvore. Entretanto, como usualmente não crescemos uma árvore até sua profundidade máxima, a ordem importa.

A imagem abaixo apresenta um comparativo das duas abordagens:



Intuição

Passo 2: Cresça uma árvore

Visto que a abordagem leaf-wise escolhe o split baseado em sua contribuição para a loss global e não apenas a loss num particular galho, ela irá, geralmente, aprender árvores de erros menores quando comparada à abordagem level-wise. A abordagem leaf-wise foi descrita em detalhes na tese de dissertação de Haijan Shi ([link](#)).

Resumidamente, temos:

“A ideia básica de como uma árvore leaf-wise é construída é a seguinte. Primeiro, selecione um atributo para colocar no nó raiz e faça algumas ramificações para esse atributo com base em alguns critérios. Em seguida, divida as instâncias de treinamento em subconjuntos, um para cada ramificação que se estende do nó raiz. Nesta tese apenas árvores de decisão binárias são consideradas e, portanto, o número de ramos é exatamente dois. Em seguida, esta etapa é repetida para um ramo escolhido, usando apenas as instâncias que realmente o alcançam. Em cada etapa, escolhemos o “melhor” subconjunto entre todos os subconjuntos disponíveis para expansões. Este processo de construção continua até que todos os nós sejam puros ou um número específico de expansões seja alcançado.”

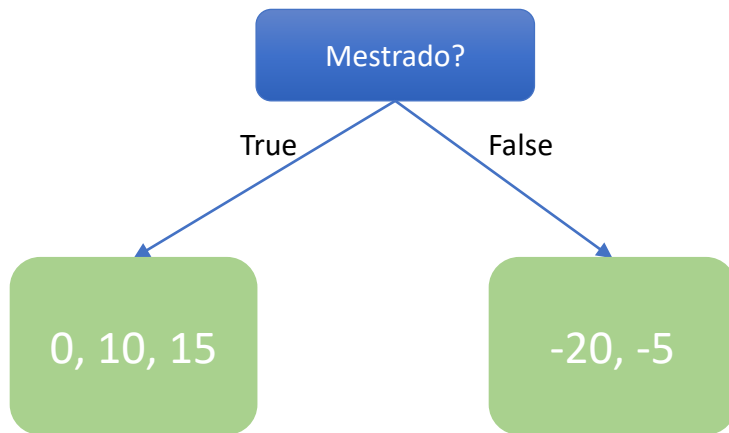
Vamos entender como crescer uma árvore usando a abordagem leaf-wise

Intuição

Passo 2: Cresça uma árvore

Importante notar que, na prática, LightGBM, assim como XGBoost, [usa histogramas para compartimentar os valores de cada feature](#). Isso ocorre especialmente quando há um grande número de linhas no dataset, o que não é nosso caso. Por isso, vamos analisar didaticamente a construção da árvore.

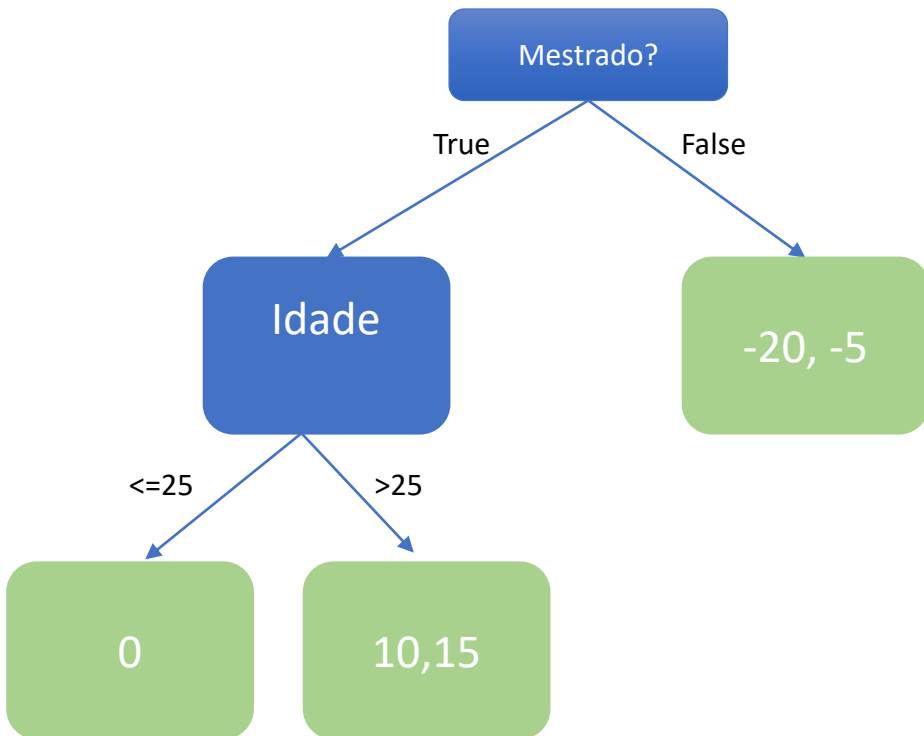
Vamos começar verificando a feature Mestrado?.



Intuição

Passo 2: Cresça uma árvore

Como é a abordagem leaf-wise, precisamos saber como a feature idade separa os dados de cada nó folha. Vamos começar pelo da esquerda:



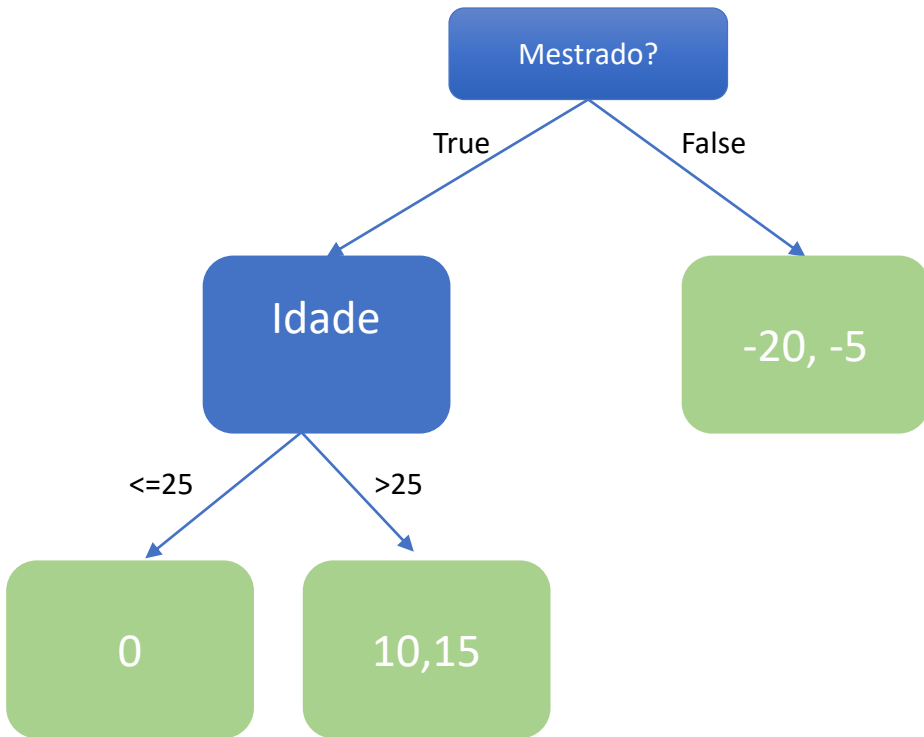
Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Nesse caso, há dois thresholds possíveis: $\text{Idade} \leq 25$ e $\text{Idade} \leq 26.5$. Vamos testar o primeiro

Intuição

Passo 2: Cresça uma árvore

Como é a abordagem leaf-wise, precisamos saber como a feature idade separa os dados de cada nó folha. Vamos começar pelo da esquerda:



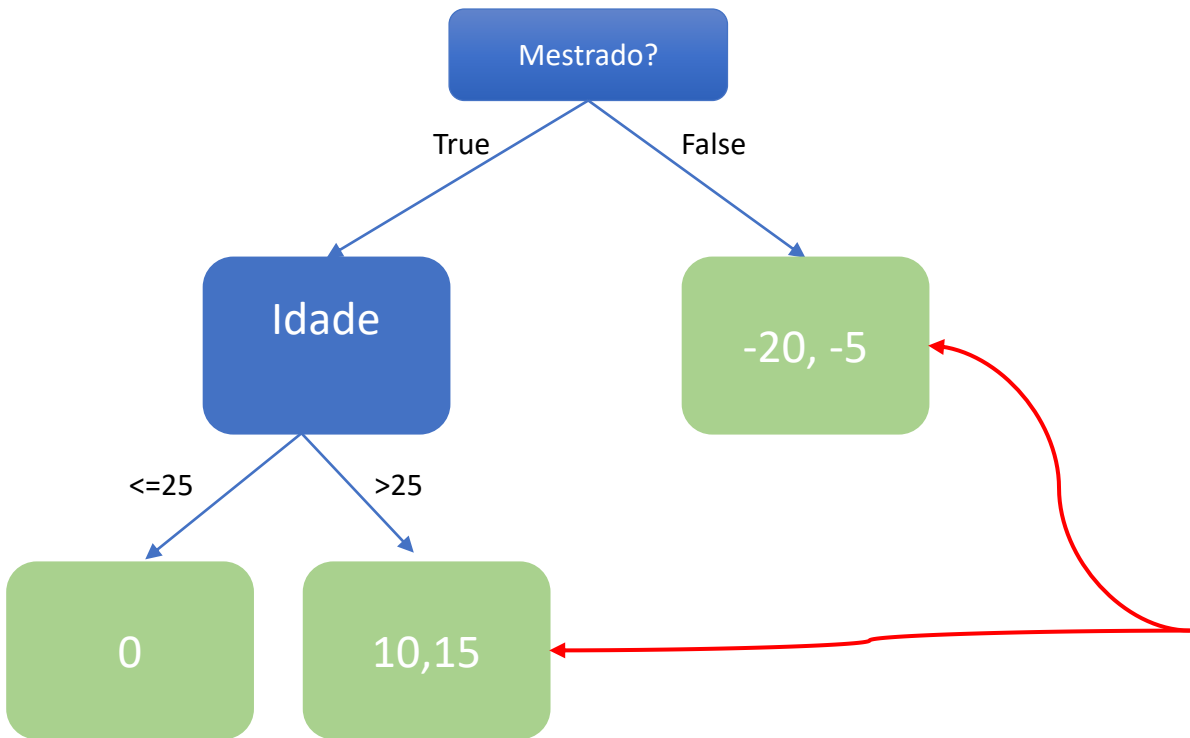
Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Importante notar que, no caso de Gradient Boost Machine, vamos tentar prever os residuals. Vou dar mais detalhes à frente.

Intuição

Passo 3: Calcule o MSE

Como não há mais features, não separamos a folha da direita. Agora calculamos o MSE para verificar se houve redução na loss.



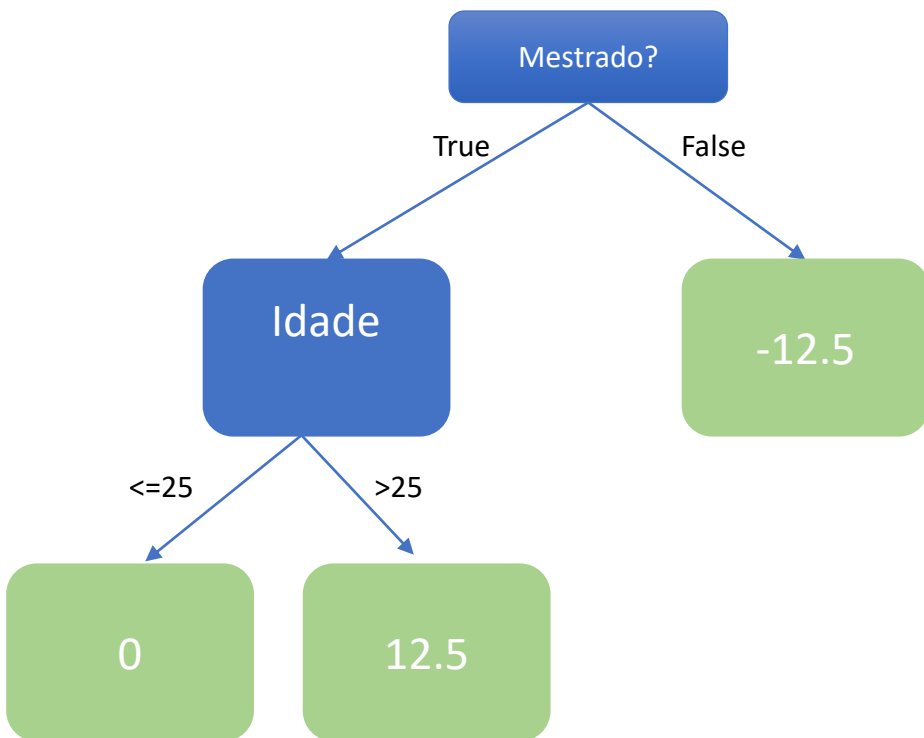
Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Para determinar o output values para nós folha com mais de dois valores, extraia a média deles.

Intuição

Passo 3: Calcule o MSE

Para calcular o MSE, fazemos a predição do salário usando a árvore obtida.



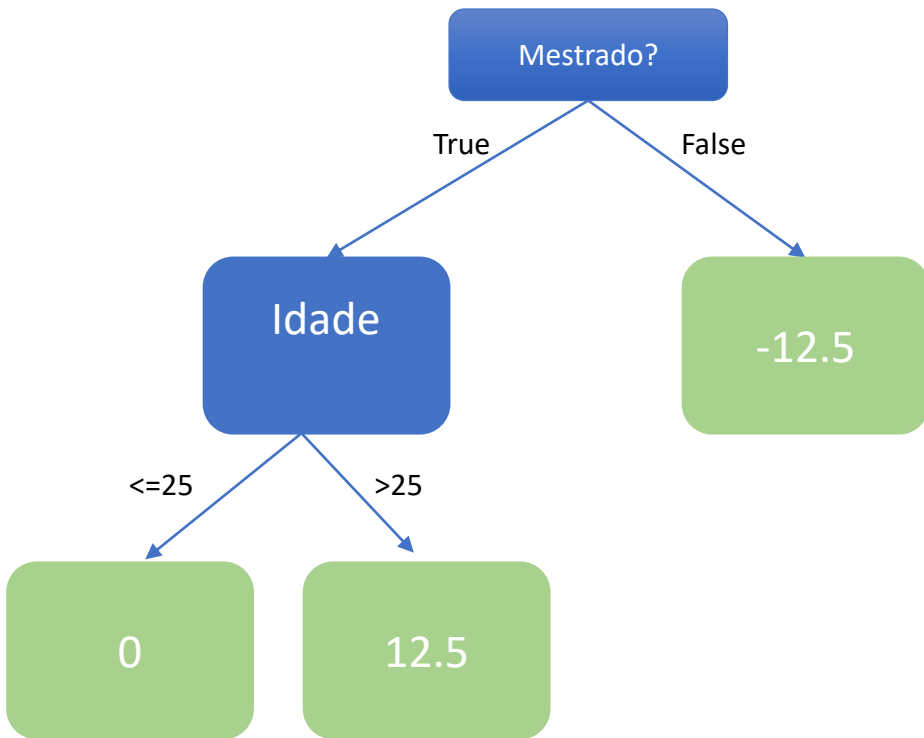
$$\text{Salário} = 70 + \text{learning rate} * \text{output value}$$

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

Intuição

Passo 3: Calcule o MSE

Para calcular o MSE, fazemos a predição do salário usando a árvore obtida.



$$\text{Salário} = 70 + \text{learning rate} * \text{output value}$$

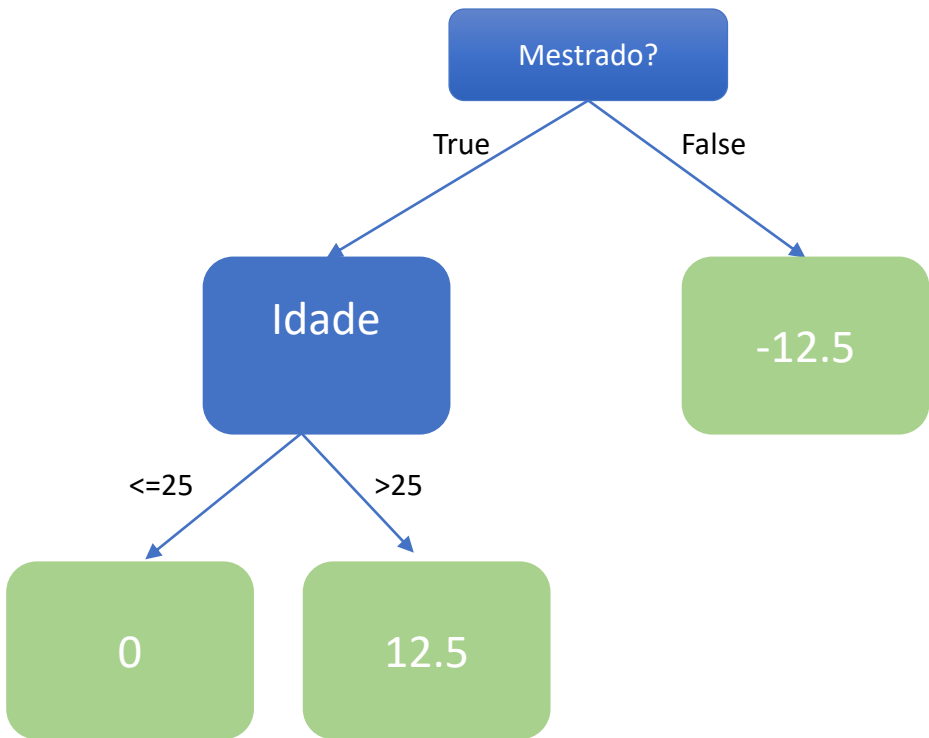
$$\text{Salário} = 70 + 0.1 * (-12.5) = 68.75$$

Idade	Mestrado?	Salário	Predição
23	Não	50	68.75
24	sim	70	70
26	Sim	80	71.25
26	Não	65	68.75
27	Sim	85	71.25

Intuição

Passo 3: Calcule o MSE

Com as novas previsões, podemos calcular os novos valores de residuals.

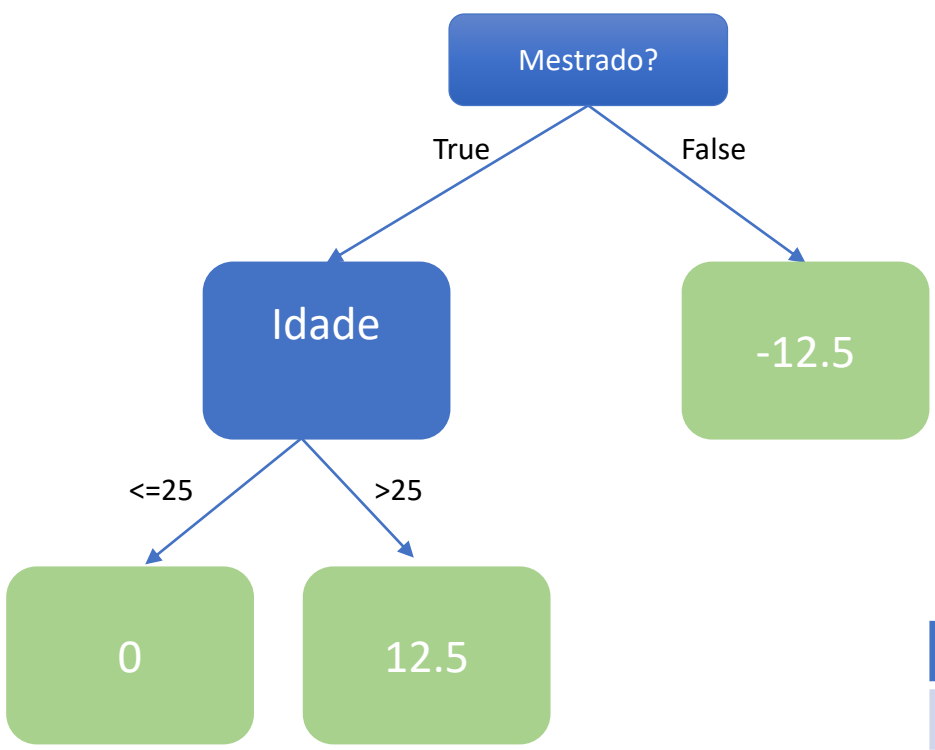


Idade	Mestrado?	Salário	Predição	Residual
23	Não	50	68.75	-18.75
24	sim	70	70	0
26	Sim	80	71.25	8.75
26	Não	65	68.75	-3.75
27	Sim	85	71.25	13.75

Intuição

Passo 3: Calcule o MSE

Podemos verificar que as novas predições diminuíram os residuals:



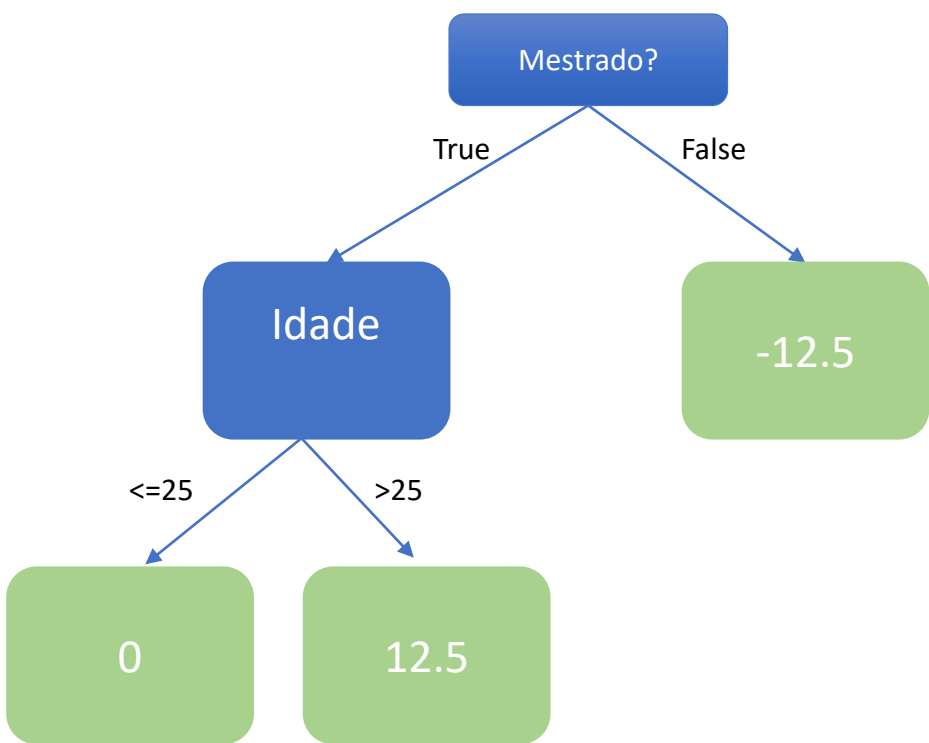
Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Idade	Mestrado?	Salário	Predição	Residual
23	Não	50	68.75	-18.75
24	sim	70	70	0
26	Sim	80	71.25	8.75
26	Não	65	68.75	-3.25
27	Sim	85	71.25	13.75

Intuição

Passo 3: Calcule o MSE

Agora calculamos o MSE



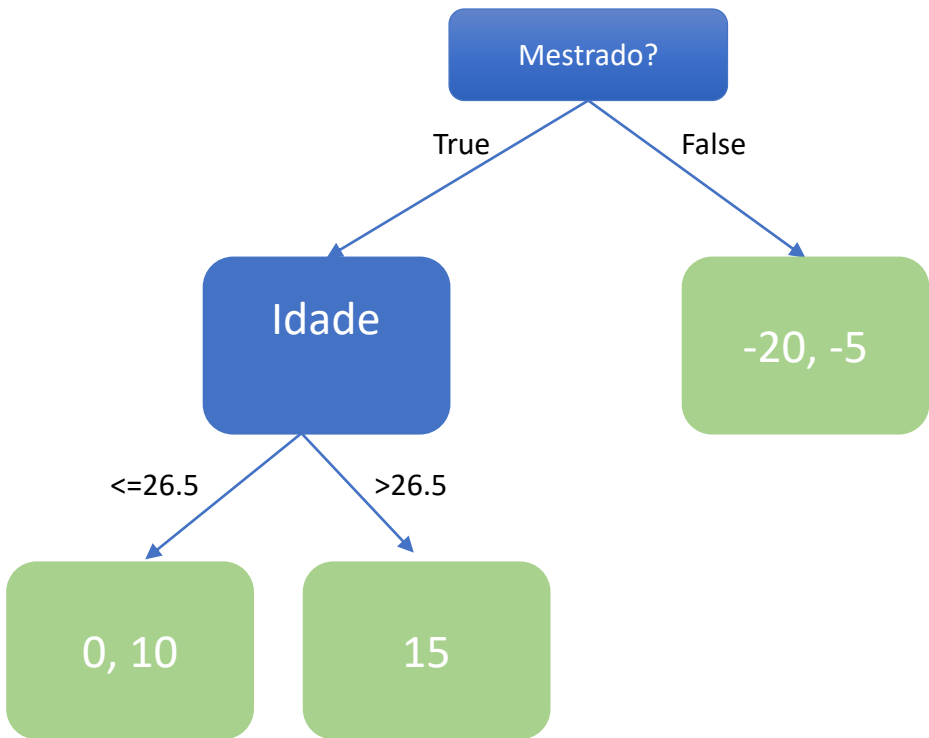
MSE = 125.55

Idade	Mestrado?	Salário	Predição	Residual
23	Não	50	68.75	-18.75
24	sim	70	70	0
26	Sim	80	71.25	8.75
26	Não	65	68.75	-3.25
27	Sim	85	71.25	13.75

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

Vimos que o atributo idade tinha dois possíveis thresholds: ≤ 25 ou ≤ 26.5 . Vamos usar o segundo valor agora e ver o MSE.

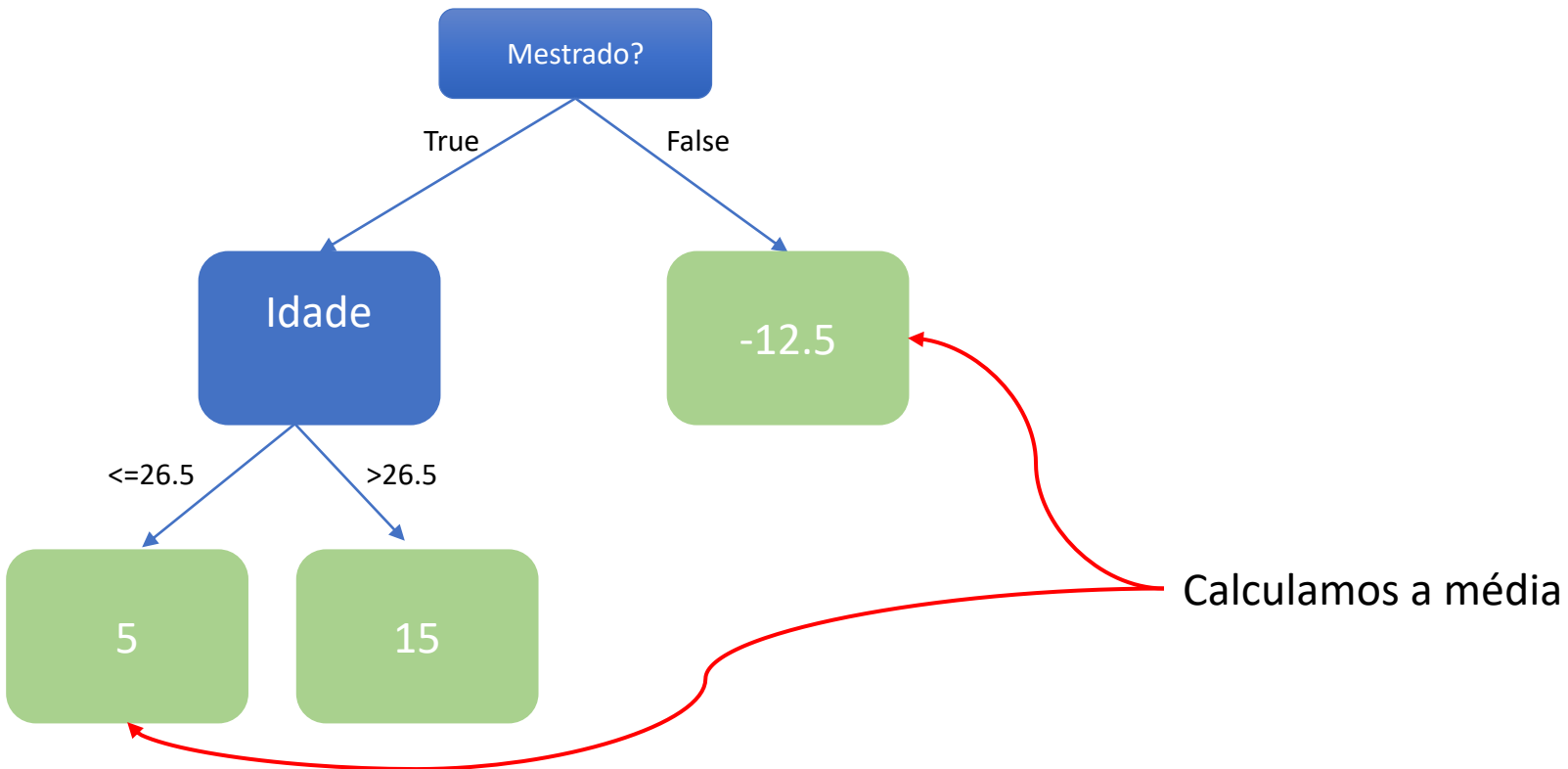


Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

Vimos que o atributo idade tinha dois possíveis valores: ≤ 25 ou ≤ 26.5 . Vamos usar o segundo valor agora e ver o MSE.

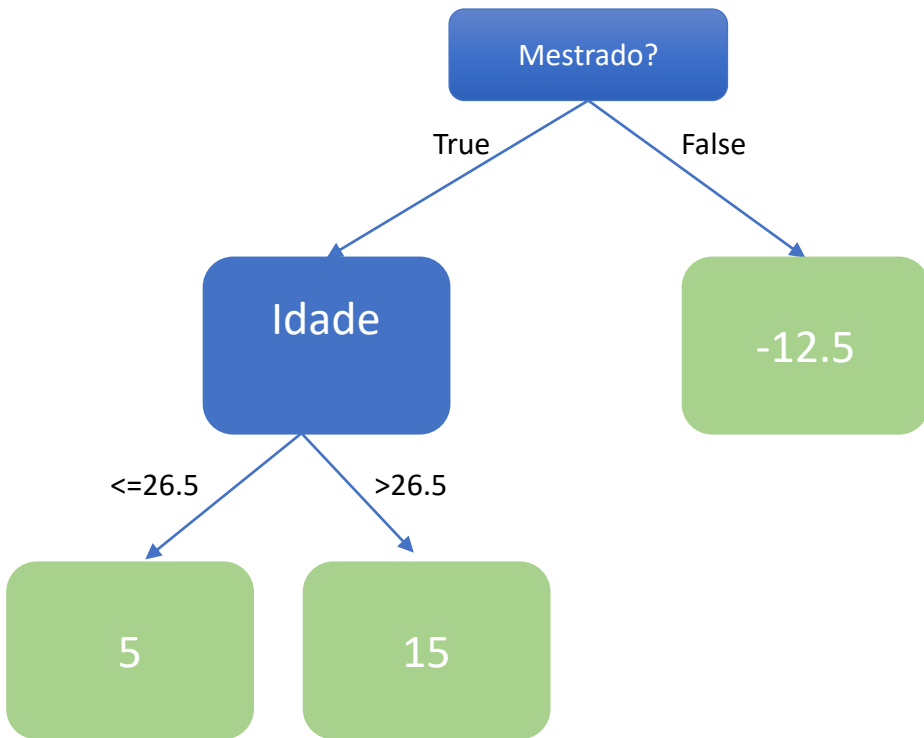


Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

Fazemos novas previsões



$$\text{Salário} = 70 + \text{learning rate} * \text{output value}$$

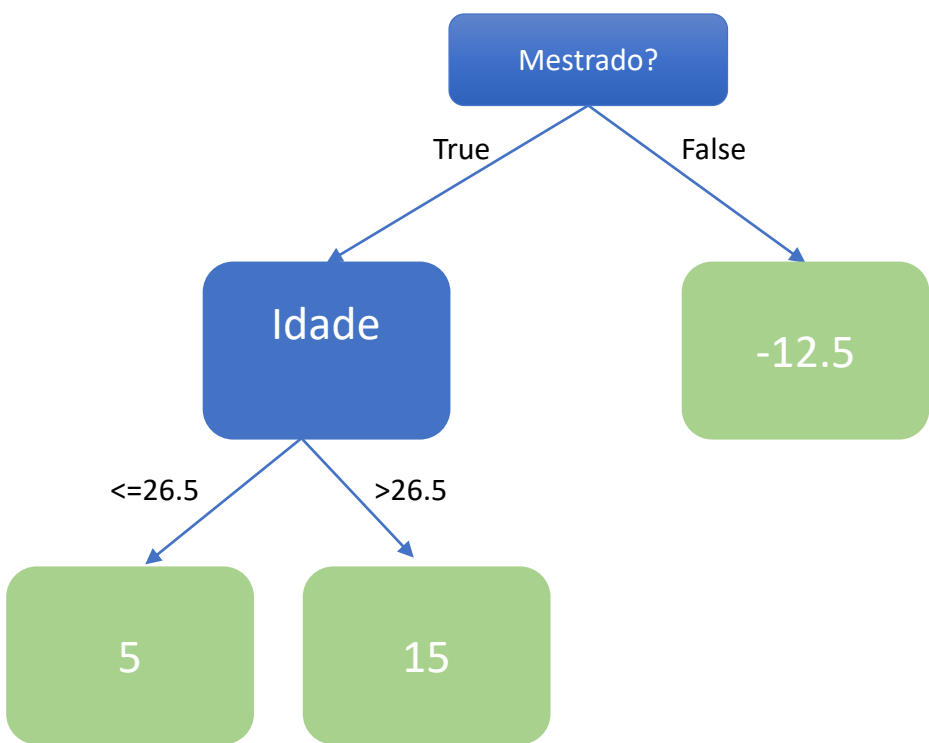
$$\text{Salário} = 70 + 0.1 * (-12.5) = 68.75$$

Idade	Mestrado?	Salário	Predição
23	Não	50	68.75
24	sim	70	70.5
26	Sim	80	70.5
26	Não	65	68.75
27	Sim	85	71.5

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

Calculamos os residuals

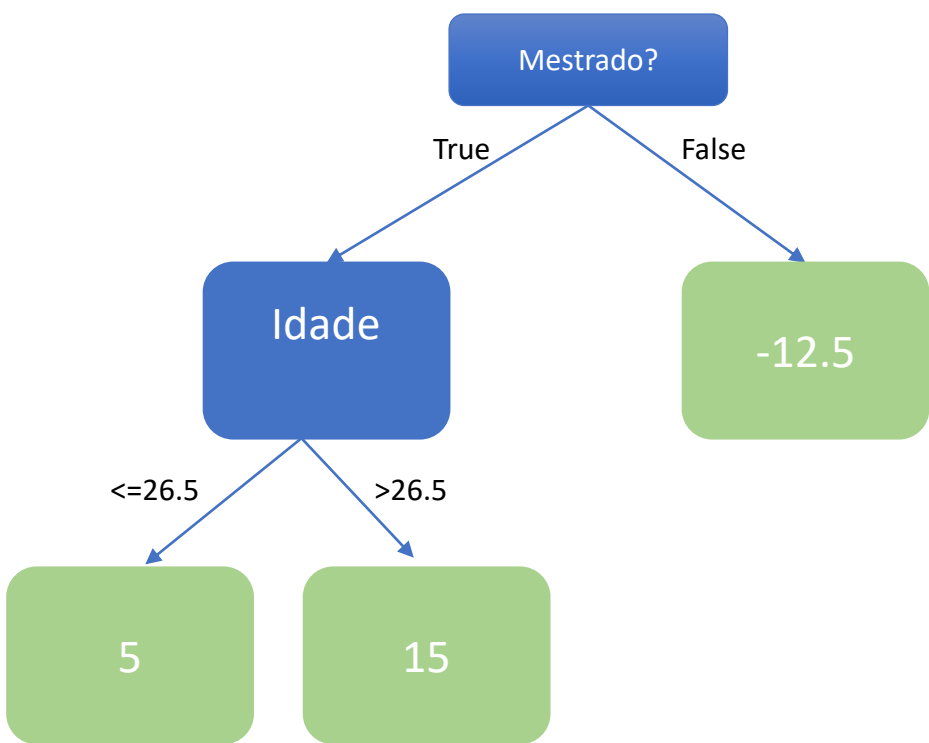


Idade	Mestrado?	Salário	Predição	Residuals
23	Não	50	68.75	-18.75
24	sim	70	70.5	-0.5
26	Sim	80	70.5	9.5
26	Não	65	68.75	-3.25
27	Sim	85	71.5	13.5

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

Calculamos o MSE



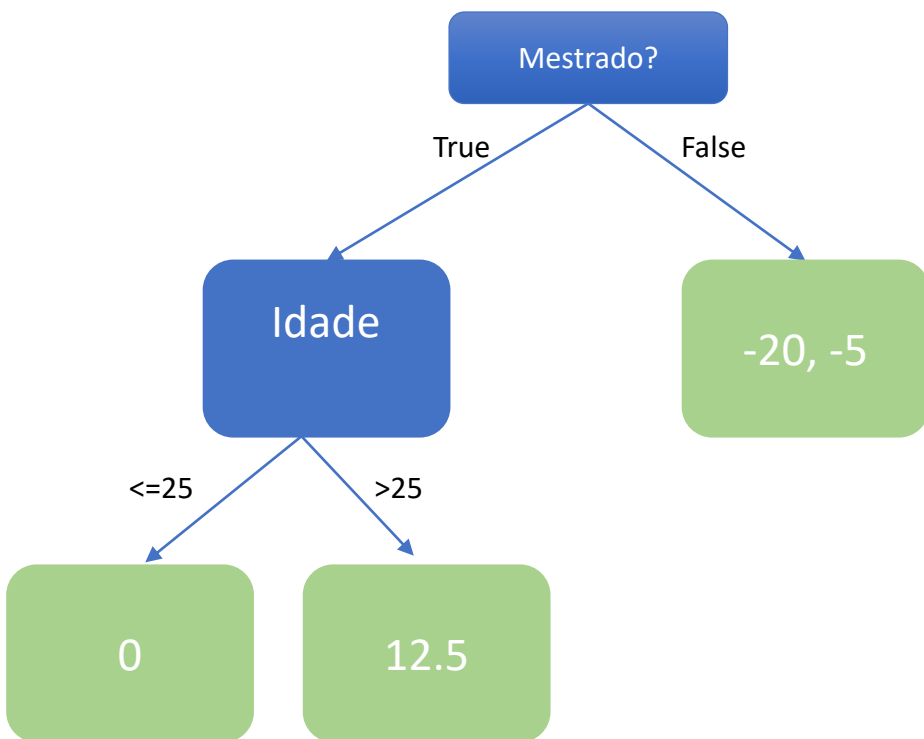
MSE = 126.975

Idade	Mestrado?	Salário	Predição	Residuals
23	Não	50	68.75	-18.75
24	sim	70	70.5	-0.5
26	Sim	80	70.5	9.5
26	Não	65	68.75	-3.25
27	Sim	85	71.5	13.5

Intuição

Passo 4: Repita o processo para o outro valor do atributo idade

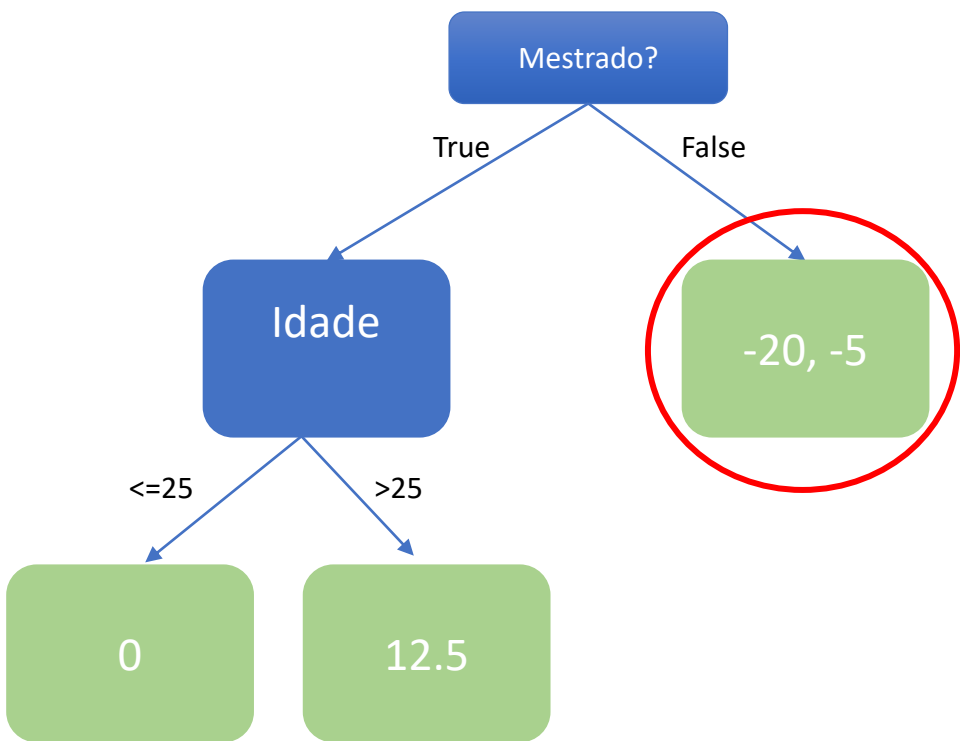
Como o MSE da primeira opção ($\text{Idade} \leq 25$) foi menor, usamos ele.



Intuição

Passo 5: Repita o processo para os demais nós

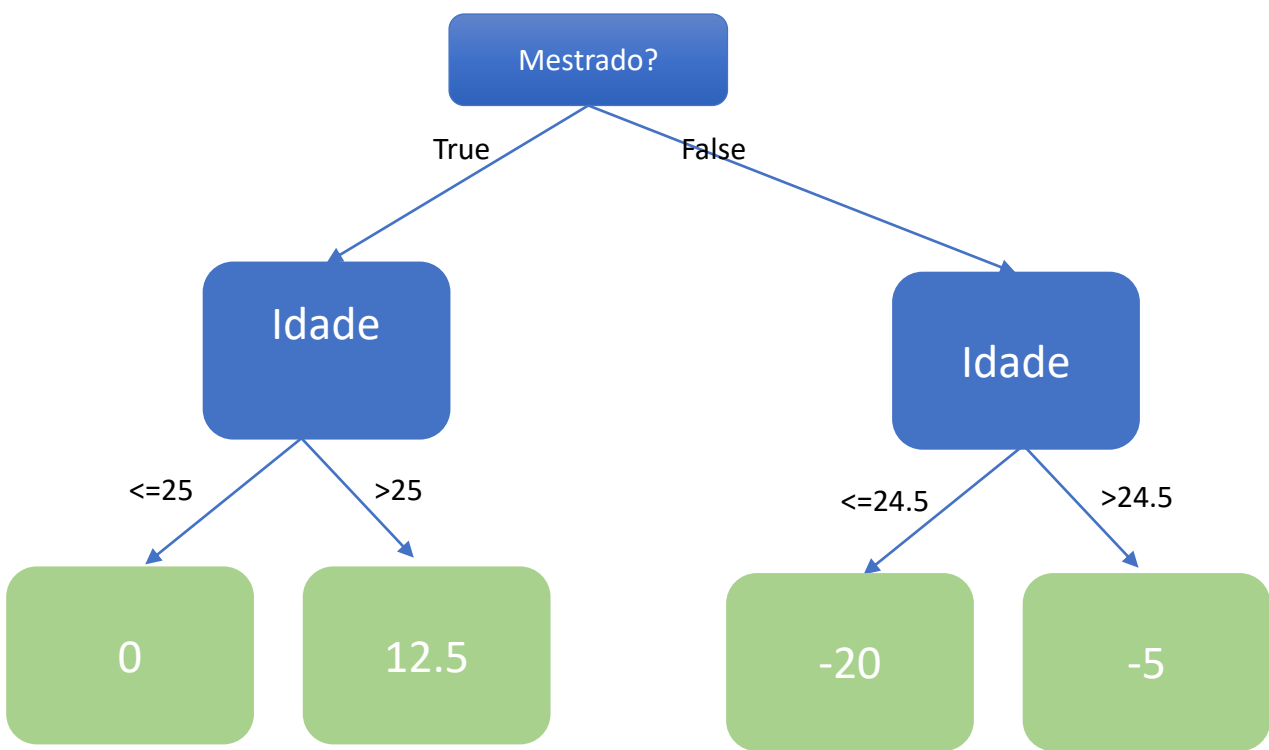
Agora precisamos saber se o nó em destaque retorna um menor MSE global assim ou quando fazemos o split dele.



Intuição

Passo 4: Repita o processo para os demais nós

Para esse caso, só temos um threshold possível: Idade ≤ 24.5

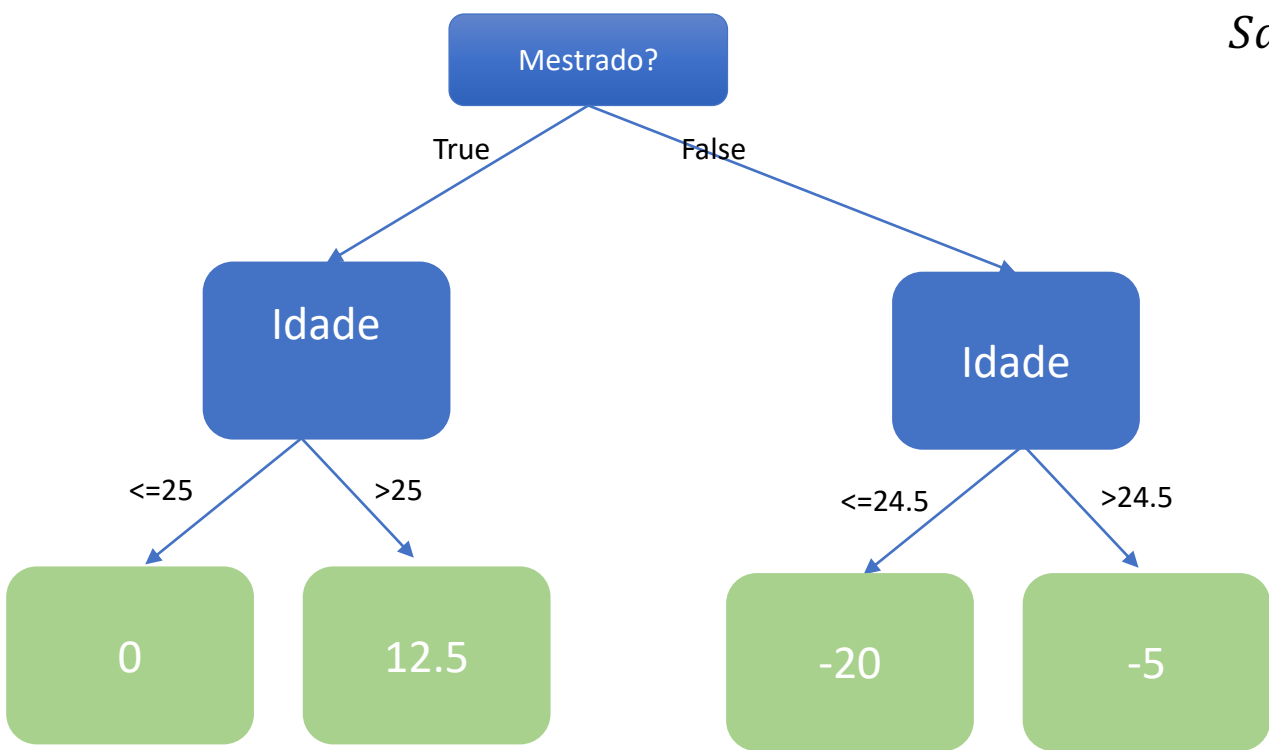


Idade	Mestrado?	Salário	Residuals
23	Não	50	-20
24	sim	70	0
26	Sim	80	10
26	Não	65	-5
27	Sim	85	15

Intuição

Passo 4: Repita o processo para os demais nós

Fazemos novas previsões



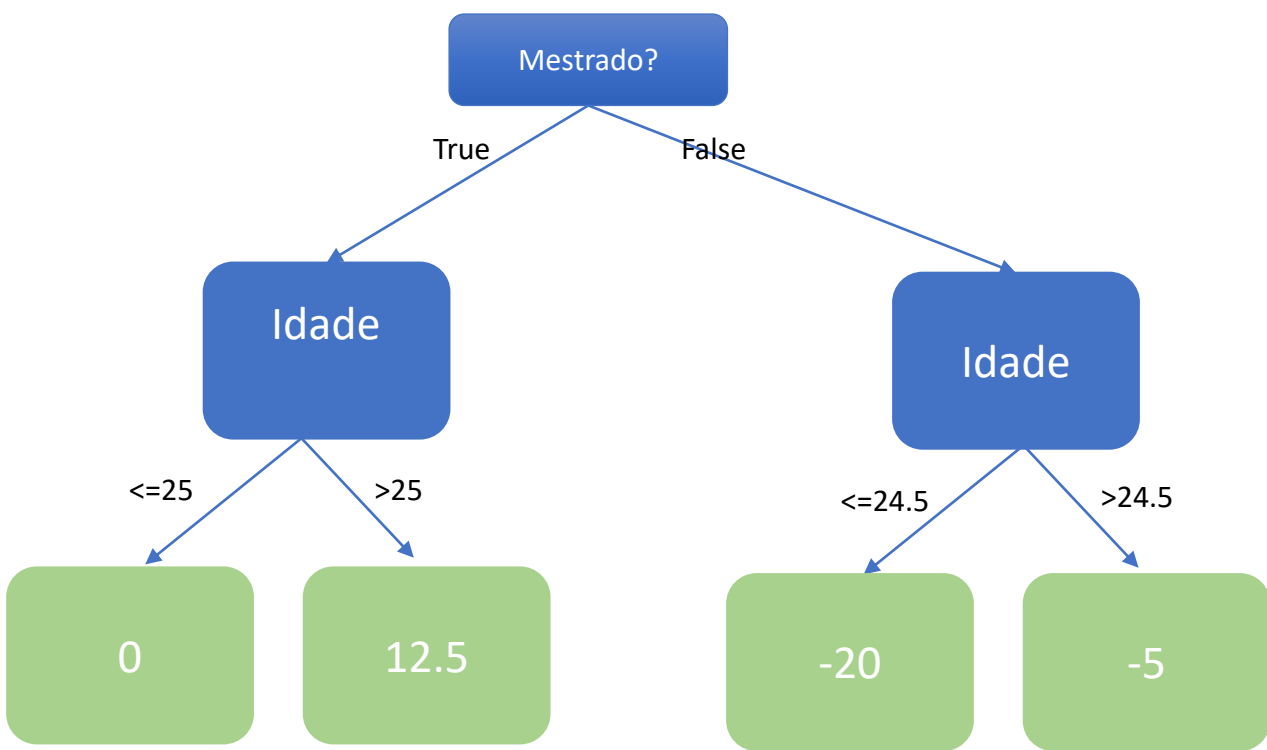
$Salário = 70 + learning\ rate * output\ value$

Idade	Mestrado?	Salário	Predição
23	Não	50	68
24	sim	70	70
26	Sim	80	71.25
26	Não	65	69.5
27	Sim	85	71.25

Intuição

Passo 4: Repita o processo para os demais nós

Calculamos os residuals

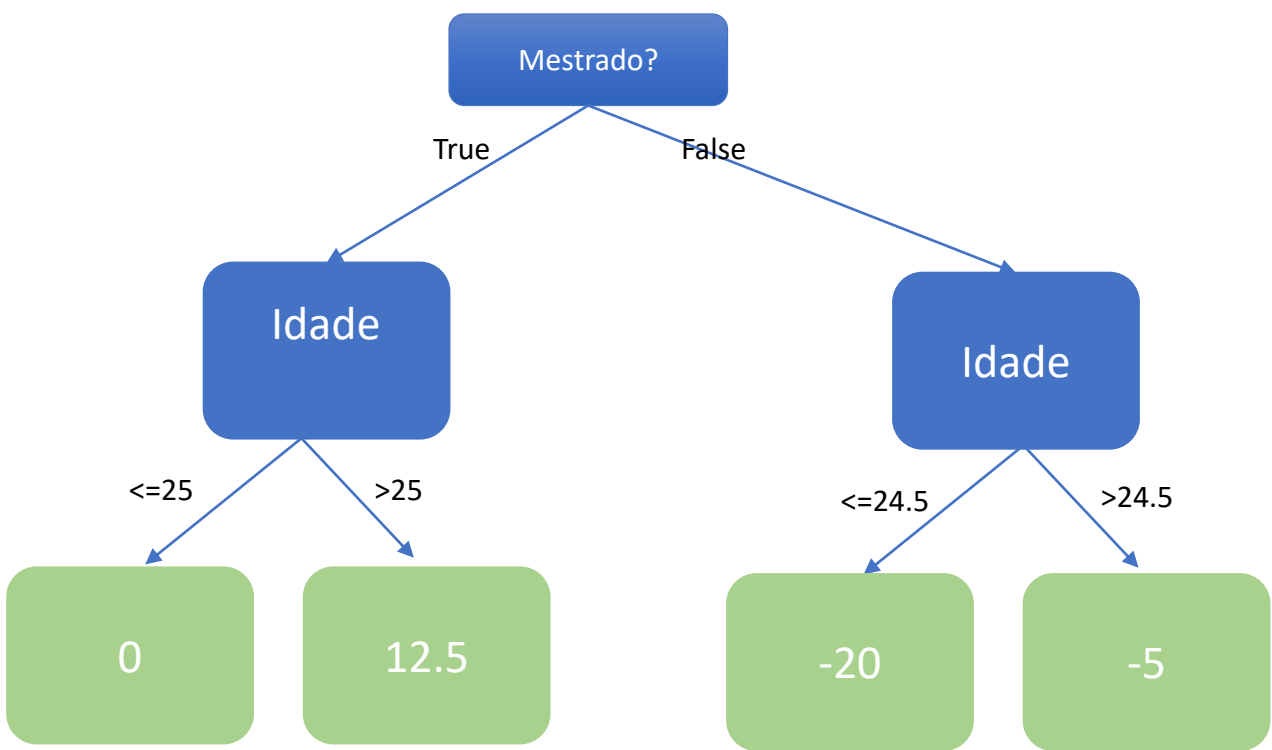


Idade	Mestrado ?	Salário	Predição	Residuals
23	Não	50	68	-18
24	sim	70	70	0
26	Sim	80	71.25	8.75
26	Não	65	69.5	-4.5
27	Sim	85	71.25	13.75

Intuição

Passo 4: Repita o processo para os demais nós

Calculamos o MSE



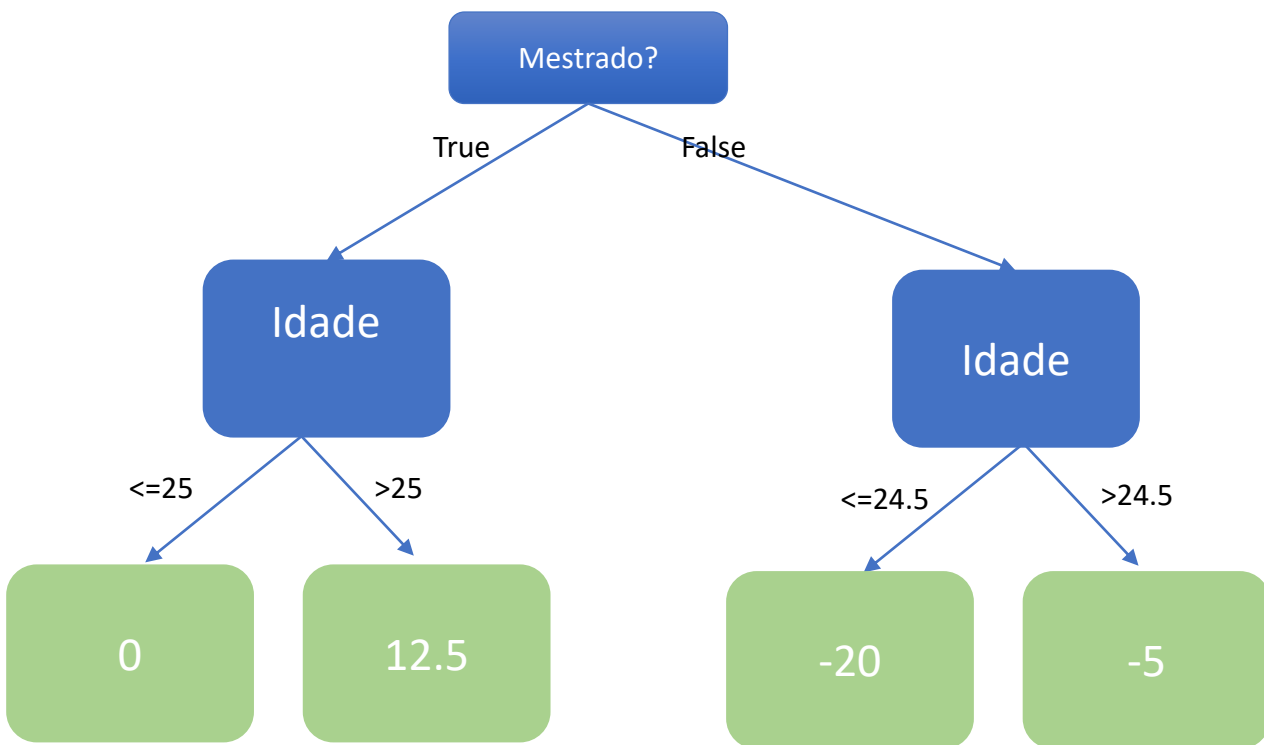
MSE = 121.975

Idade	Mestrado ?	Salário	Predição	Residuals
23	Não	50	68	-18
24	sim	70	70	0
26	Sim	80	71.25	8.75
26	Não	65	69.5	-4.5
27	Sim	85	71.25	13.75

Intuição

Passo 4: Repita o processo para os demais nós

Como reduzimos o MSE, essa é a árvore final.



Intuição

Passo 5: Crie novas árvores usando outros atributos como nó raiz. Por padrão, 100 árvores são construídas

Assim, a predição será o valor inicial (70) somado ao learning rating que multiplica cada árvore individual

Otimizações



Otimizações

As implementações convencionais de GBDT precisam, para cada feature, verificar todas as amostras para estimar o ganho de informação de todos os possíveis pontos de split.

Portanto, **a complexidade computacional será proporcional ao número de features e ao número de amostras**. Isto torna o tempo de treinamento um gargalo quando lidamos com big data.

Para lidar com essas limitações, os autores propuseram duas técnicas novas:

1. *Gradient-based One-Side Sampling (GOSS)*
2. *Exclusive Feature Bundling (EFB)*

Ambos serão descritos em detalhes agora.

Otimizações

As aplicações real que usam datasets de larga escala são usualmente esparsos. Usando a abordagem gulosa, um algoritmo GBDT pode reduzir o custo de treinamento ignorando features com valores iguais a zero. Entretanto, a abordagem baseada em histogramas não possui soluções otimizadas para lidar com esparsidade. A razão é que o algoritmo baseado em histogramas precisa recuperar os valores dos bins para cada ponto de dado, não importa se seu valor é zero ou não.

A primeira maneira que o LightGBM se propõe a resolver esse problema é usando o algoritmo GOSS.

Otimizações

O pseudo-código abaixo apresenta o GOSS. Seu objetivo é prover um bom equilíbrio entre reduzir o número de amostras a serem treinadas e manter a acurácia para árvores aprendidas.

Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations

Input: a : sampling ratio of large gradient data

Input: b : sampling ratio of small gradient data

Input: $loss$: loss function, L : weak learner

$models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$

$topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$

for $i = 1$ **to** d **do**

$preds \leftarrow models.predict(I)$

$g \leftarrow loss(I, preds)$, $w \leftarrow \{1, 1, \dots\}$

$sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$

$topSet \leftarrow sorted[1:topN]$

$randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)],$
 $randN)$

$usedSet \leftarrow topSet + randSet$

$w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the
 small gradient data.

$newModel \leftarrow L(I[usedSet], -g[usedSet],$
 $w[usedSet])$

$models.append(newModel)$

Segundo os autores:

“... Notamos que o gradiente (residual errors) para cada amostra fornece informação útil para amostragem dos dados, isto é, *se uma amostra é associada com um pequeno gradiente, o erro de treinamento dessa amostra é pequena e ela já foi bem treinada. Logo, uma ideia óbvia é descartar tais amostras.* Entretanto, fazer isso de maneira direta irá alterar a distribuição dos dados, o que afeta diretamente a acurácia do modelo aprendido. Para evitar isso, surgiu o GOSS.”

Otimizações

Vamos entender o passo a passo do algoritmo:

Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations

Input: a : sampling ratio of large gradient data

Input: b : sampling ratio of small gradient data

Input: $loss$: loss function, L : weak learner

$models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$

$topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$

for $i = 1$ **to** d **do**

$preds \leftarrow models.predict(I)$

$g \leftarrow loss(I, preds)$, $w \leftarrow \{1,1,...\}$

$sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$

$topSet \leftarrow sorted[1:topN]$

$randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)],$
 $randN)$

$usedSet \leftarrow topSet + randSet$

$w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the
 small gradient data.

$newModel \leftarrow L(I[usedSet], -g[usedSet],$

$w[usedSet])$

$models.append(newModel)$

- I : conjunto de treinamento
- d : número de iterações
- a : taxa de amostras com valor de gradiente alto (20)
- b : taxa de amostras com valor de gradiente baixo (10)
- $fact$: constante que amplifica amostras com valor de gradiente baixo quando for calcular o ganho de informação.
- $topN$: amostras com valor de gradiente alto que serão selecionadas para treino
- $randN$: amostras com valor de gradiente baixo que serão selecionadas para treino

Otimizações

Vamos entender o passo a passo do algoritmo:

Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations

Input: a : sampling ratio of large gradient data

Input: b : sampling ratio of small gradient data

Input: $loss$: loss function, L : weak learner

$models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$

$topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$

for $i = 1$ **to** d **do**

$preds \leftarrow models.predict(I)$

$g \leftarrow loss(I, preds)$, $w \leftarrow \{1, 1, \dots\}$

$sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$

$topSet \leftarrow sorted[1:topN]$

$randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)],$
 $randN)$

$usedSet \leftarrow topSet + randSet$

$w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the
 small gradient data.

$newModel \leftarrow L(I[usedSet], -g[usedSet],$
 $w[usedSet])$

$models.append(newModel)$

Até o número d de iterações, faça:

- Predição do modelo no conjunto de treino
- Calcula a loss e armazena em g e cria um vetor de pesos (w), com 1's
- Ordene os valores dos gradientes de maneira descendente
- $topSet$ será as amostras com maior valor de gradiente
- $randSet$ (usualmente 10%) será o valor de amostras com menor valor de gradiente
- $usedSet$ é a soma dos dois anteriores. Será usado para treinar um novo modelo

Otimizações

Vamos entender o passo a passo do algoritmo:

Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations

Input: a : sampling ratio of large gradient data

Input: b : sampling ratio of small gradient data

Input: $loss$: loss function, L : weak learner

$models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$

$topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$

for $i = 1$ **to** d **do**

$preds \leftarrow models.predict(I)$

$g \leftarrow loss(I, preds)$, $w \leftarrow \{1, 1, \dots\}$

$sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$

$topSet \leftarrow sorted[1:topN]$

$randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)],$
 $randN)$

$usedSet \leftarrow topSet + randSet$

$w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the
 small gradient data.

$newModel \leftarrow L(I[usedSet], -g[usedSet],$

$w[usedSet])$

$models.append(newModel)$

Até o número d de iterações, faça:

- Amplifica o dataset com valores pequenos de gradiente
- Treine um novo modelo no subconjunto criado

Otimizações

Resumidamente, temos:

Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations

Input: a : sampling ratio of large gradient data

Input: b : sampling ratio of small gradient data

Input: $loss$: loss function, L : weak learner

$models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$

$topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$

for $i = 1$ **to** d **do**

$preds \leftarrow models.predict(I)$

$g \leftarrow loss(I, preds)$, $w \leftarrow \{1, 1, \dots\}$

$sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$

$topSet \leftarrow sorted[1:topN]$

$randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)],$
 $randN)$

$usedSet \leftarrow topSet + randSet$

$w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the
 small gradient data.

$newModel \leftarrow L(I[usedSet], -g[usedSet],$
 $w[usedSet])$

$models.append(newModel)$

GOSS manterá todas as instancias com valor de gradiente grande e aleatoriamente selecionará amostras com valor de gradiente pequeno.

Primeiramente, o GOSS ordena as amostras de acordo com o valor absoluto de seu gradiente e seleciona as top $a \times 100\%$ amostras. Então, aleatoriamente selecionará $b \times 100\%$ amostras do resto dos dados. Depois disso, GOSS amplificará as amostras com pequeno valor de gradiente por uma constante $\frac{1-a}{b}$ quando calcular o ganho de informação.

Fazendo isso, um foco maior é posto em amostras não tão bem treinadas sem alterar muito a distribuição original dos dados.

Otimizações

Com o GOSS, reduzimos o número de amostras usadas para treinar em cada iteração. Para lidar com o número de features, o LightGBM introduz um novo método: *Exclusive Feature Bundling*.

Nas palavras dos autores:

“Dados de alta dimensão são geralmente muito esparsos. A esparsidade do espaço de características nos oferece a possibilidade de projetar uma abordagem quase sem perdas para reduzir o número de features. Especificamente, em um esperso espaço de características, muitas delas são mutuamente exclusivas, ou seja, elas nunca assumem valores diferentes de zero simultaneamente. Podemos agrupar com segurança features exclusivas em uma única feature. Por meio de um algoritmo de scanning de features cuidadosamente projetado, podemos construir os mesmos histogramas de features dos grupos de features como aqueles de features individuais.”

Há dois problemas a serem endereçados: O primeiro é determinar quais features devem ser combinadas (empacotadas). O segundo é como construir essa combinação.

Otimizações

O pseudo código abaixo ilustra como resolver o primeiro problema (**determinar quais features devem ser combinadas (empacotadas)**):

Algorithm 3: Greedy Bundling

Input: F : features, K : max conflict count

Construct graph G

$\text{searchOrder} \leftarrow G.\text{sortByDegree}()$

$\text{bundles} \leftarrow \{\}, \text{bundlesConflict} \leftarrow \{\}$

for i **in** searchOrder **do**

$\text{needNew} \leftarrow \text{True}$

for $j = 1$ **to** $\text{len}(\text{bundles})$ **do**

$\text{cnt} \leftarrow \text{ConflictCnt}(\text{bundles}[j], F[i])$

if $\text{cnt} + \text{bundlesConflict}[j] \leq K$ **then**

$\text{bundles}[j].\text{add}(F[i])$, $\text{needNew} \leftarrow \text{False}$

break

if needNew **then**

 Add $F[i]$ as a new bundle to bundles

Output: bundles

De maneira geral, é difícil encontrar features totalmente exclusivas.

Mesmo assim, os autores identificaram que se alguns conflitos forem permitidos, é possível obter um número ainda menor de pacotes de features e melhorar a eficiência computacional.

Otimizações

O pseudo código abaixo ilustra como resolver o primeiro problema (**determinar quais features devem ser combinadas (empacotadas)**):

Algorithm 3: Greedy Bundling

Input: F : features, K : max conflict count

Construct graph G

$\text{searchOrder} \leftarrow G.\text{sortByDegree}()$

$\text{bundles} \leftarrow \{\}, \text{bundlesConflict} \leftarrow \{\}$

for i **in** searchOrder **do**

$\text{needNew} \leftarrow \text{True}$

for $j = 1$ **to** $\text{len}(\text{bundles})$ **do**

$\text{cnt} \leftarrow \text{ConflictCnt}(\text{bundles}[j], F[i])$

if $\text{cnt} + \text{bundlesConflict}[j] \leq K$ **then**

$\text{bundles}[j].\text{add}(F[i])$, $\text{needNew} \leftarrow \text{False}$

break

if needNew **then**

 Add $F[i]$ as a new bundle to bundles

Output: bundles

A tarefa de encontrar os pacotes de features pode ser reduzida ao problema de colorir um grafo e é **NP-hard**, ou seja, **o tempo e a complexidade computacional aumentam exponencialmente com o número de features.**

O desafio num problema de colorir um grafo é preencher os vértices de um grafo com o mínimo número de cores únicas possíveis sem que dois vértices conectados por uma aresta tenham a mesma cor.

Otimizações

O pseudo código abaixo ilustra como resolver o primeiro problema (**determinar quais features devem ser combinadas (empacotadas)**):

Algorithm 3: Greedy Bundling

Input: F : features, K : max conflict count

Construct graph G

$\text{searchOrder} \leftarrow G.\text{sortByDegree}()$

$\text{bundles} \leftarrow \{\}, \text{bundlesConflict} \leftarrow \{\}$

for i **in** searchOrder **do**

$\text{needNew} \leftarrow \text{True}$

for $j = 1$ **to** $\text{len}(\text{bundles})$ **do**

$\text{cnt} \leftarrow \text{ConflictCnt}(\text{bundles}[j], F[i])$

if $\text{cnt} + \text{bundlesConflict}[i] \leq K$ **then**

$\text{bundles}[j].\text{add}(F[i])$, $\text{needNew} \leftarrow \text{False}$

break

if needNew **then**

 Add $F[i]$ as a new bundle to bundles

Output: bundles

Variáveis de inicialização.

Neste grafo, cada feature é um vértice. Duas features que não são mutuamente exclusivas são conectadas por uma aresta.

As features são ordenadas pelo seu grau (# de arestas que apontam pra feature) de maneira decrescente (um grau mais elevado significa mais conflitos e menos esparsidade).

Otimizações

O pseudo código abaixo ilustra como resolver o primeiro problema (**determinar quais features devem ser combinadas (empacotadas)**):

Algorithm 3: Greedy Bundling

Input: F : features, K : max conflict count

Construct graph G

$\text{searchOrder} \leftarrow G.\text{sortByDegree}()$

$\text{bundles} \leftarrow \{\}, \text{bundlesConflict} \leftarrow \{\}$

```
for  $i$  in  $\text{searchOrder}$  do
   $\text{needNew} \leftarrow \text{True}$ 
  for  $j = 1$  to  $\text{len}(\text{bundles})$  do
     $\text{cnt} \leftarrow \text{ConflictCnt}(\text{bundles}[j], F[i])$ 
    if  $\text{cnt} + \text{bundlesConflict}[j] \leq K$  then
       $\text{bundles}[j].\text{add}(F[i])$ ,  $\text{needNew} \leftarrow \text{False}$ 
      break
  if  $\text{needNew}$  then
    Add  $F[i]$  as a new bundle to  $\text{bundles}$ 
```

Output: bundles

Agora, checamos diferentes combinações de features e criamos um pacote se a combinação possui uma contagem de conflitos menores que o threshold pre-definido (K).

Otimizações

Para resolver o segundo problema, [como construir essa combinação](#), considere o pseudo-código abaixo:

Algorithm 4: Merge Exclusive Features

Input: *numData*: number of data

Input: *F*: One bundle of exclusive features

binRanges \leftarrow {0}, *totalBin* \leftarrow 0

for *f* **in** *F* **do**

totalBin += *f.numBin*

binRanges.append(*totalBin*)

newBin \leftarrow new Bin(*numData*)

for *i* = 1 **to** *numData* **do**

newBin[*i*] \leftarrow 0

for *j* = 1 **to** *len(F)* **do**

if *F*[*j*].*bin*[*i*] \neq 0 **then**

newBin[*i*] \leftarrow *F*[*j*].*bin*[*i*] + *binRanges*[*j*]

Output: *newBin*, *binRanges*

Variáveis de entrada. Visto que o algoritmo baseado em histograma armazena bins (compartimentos) discretos, podemos construir um pacote de features deixando features exclusivas residir em diferentes bins.

Otimizações

Para resolver o segundo problema, [como construir essa combinação](#), considere o pseudo-código abaixo:

Algorithm 4: Merge Exclusive Features

Input: *numData*: number of data

Input: *F*: One bundle of exclusive features

binRanges \leftarrow {0}, *totalBin* \leftarrow 0

for *f* **in** *F* **do**

totalBin += *f.numBin*
 binRanges.append(*totalBin*)

newBin \leftarrow new Bin(*numData*)

for *i* = 1 **to** *numData* **do**

newBin[*i*] \leftarrow 0
 for *j* = 1 **to** *len(F)* **do**
 if *F*[*j*].*bin*[*i*] \neq 0 **then**
 newBin[*i*] \leftarrow *F*[*j*].*bin*[*i*] + *binRanges*[*j*]

Output: *newBin*, *binRanges*

Isto é feito adicionando “compensações” ao valor original das features (*f.numBin*).

Por exemplo, suponha que você possua duas features num pacote de features.

Originalmente, a feature A assume valores no intervalo [0,10) e a feature B assume valores no intervalo [10,30). Adicionamos uma “compensação” de 10 aos valores da feature B de modo que a feature refinada possua um range [0,30].

Otimizações

Para resolver o segundo problema, [como construir essa combinação](#), considere o pseudo-código abaixo:

Algorithm 4: Merge Exclusive Features

Input: *numData*: number of data

Input: *F*: One bundle of exclusive features

binRanges \leftarrow {0}, *totalBin* \leftarrow 0

for *f* **in** *F* **do**

totalBin += *f.numBin*

binRanges.append(*totalBin*)

newBin \leftarrow new Bin(*numData*)

for *i* = 1 **to** *numData* **do**

newBin[*i*] \leftarrow 0

for *j* = 1 **to** *len(F)* **do**

if *F*[*j*].*bin*[*i*] \neq 0 **then**

newBin[*i*] \leftarrow *F*[*j*].*bin*[*i*] + *binRanges*[*j*]

Output: *newBin*, *binRanges*

O exemplo abaixo ilustra a ideia:

feature1	feature2	feature_bundle
0	2	6
0	1	5
0	2	6
1	0	1
2	0	2
3	0	3
4	0	4

Otimizações

Para resolver o segundo problema, [como construir essa combinação](#), considere o pseudo-código abaixo:

Algorithm 4: Merge Exclusive Features

Input: *numData*: number of data

Input: *F*: One bundle of exclusive features

binRanges \leftarrow {0}, *totalBin* \leftarrow 0

for *f* **in** *F* **do**

totalBin $+=$ *f.numBin*

binRanges.append(*totalBin*)

newBin \leftarrow new Bin(*numData*)

for *i* = 1 **to** *numData* **do**

newBin[*i*] \leftarrow 0

for *j* = 1 **to** *len(F)* **do**

if *F*[*j*].*bin*[*i*] \neq 0 **then**

newBin[*i*] \leftarrow *F*[*j*].*bin*[*i*] + *binRanges*[*j*]

Output: *newBin*, *binRanges*


Depois disso, é seguro fazer um merge das features A e B e usar o pacote de features no intervalo [0,30] para substituir as features originais A e B.

Implementação



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