# **Light Gradient Boosting Machine (Light GBM)**

## Introdução

* Modelo → relação matemática entre diversas variáveis;
  + *Machine Learning*: criação de modelos que aprendem com os dados
  + No caso de modelos parametrizados, usamos os dados para estimar quais são os melhores parâmetros que descrevem a relação entre as variáveis

### Tradeoff Viés-Variância - [StatQuest](https://www.youtube.com/watch?v=EuBBz3bI-aA)

* O ***tradeoff viés-variância*** está presente em todos os modelos de *Machine Learning* (ML).
  + IMPORTANTE: todos os modelos separam os dados em **teste** e **treino**, **antes** de qualquer imputação/modificação (para que não haja *spillovers*/contaminação)
    - Modelagem nos dados de treino, validação nos de teste
      * Métricas de validação: MAPE, MSE, ROC, AUC…
  + Basicamente, quanto maior a variabilidade captada pelo modelo (por exemplo, mais graus em um regressão polinomial), maior o viés que será transmitido dos dados de treino para os de teste.
    - **Viés**: inabilidade de um método de ML de capturar a relação verdadeira entre os dados
      * Exemplo: regressão linear simples em uma relação de dados logarítmica/exponencial
    - **Variância**: diferença de *fit* entre os dados de treino e de teste (ou entre diferentes conjuntos de dados de treino)
    - Algoritmo ideal: baixo viés e variância, encontrando o ponto de equilíbrio entre um modelo **simples** (com mais viés e menos variância) e um **complexo** que incorpora mais informação(com pouco viés, mas muita variância).
      * Métodos para encontrar esse ponto: *regularização*, *boosting*, *bagging*
      * Atenção!
        + Mesmo que haja um bom desempenho nos dados de treino e teste, há de se garantir que esses dados de fato são **representativos**
        + A escolha do “melhor modelo” não deve ocorrer apenas pelo melhor desempenho no conjunto de testes (“metatreinamento”, que define os conjuntos de testes como um segundo conjunto de treino)

Ideal: dividir os dados em três partes: treinamento para modelos em desenvolvimento, validação para a escolha do melhor modelo e um conjunto de testes para avaliar o modelo final

* + - * + “**Acurácia**” (% de acertos binários de classificação frente ao total) pode ser *misleading* em dados não-balanceados (teste idiota pode ter 98% de precisão, como o do *Data Science do Zero*, pág. 162)

Ideal: **Precisão/*Precision*** (previsões vdd positivas frente ao total de positivos preditos (falsos e verdadeiros)) + **Sensibilidade/*Recall*** (fração dos positivos verdadeiros identificadas pelo modelo)

Combinação das métricas: **F1 Score** (); quanto maior, melhor! (média harmônica das duas métricas)

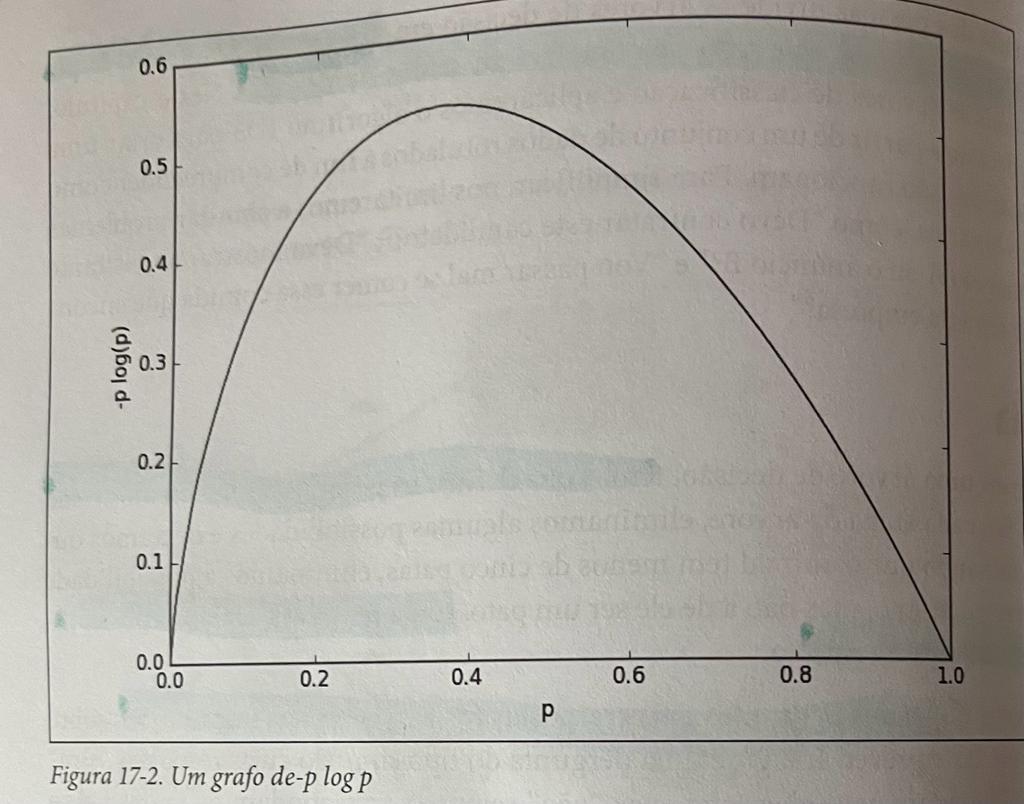
Um modelo que prevê “sim” pra tudo geralmente tem alta sensibilidade, mas baixa precisão, gerando muitos falsos positivos

* + - Alta variância → baixo viés: **overfitting**
      * Modelo se adequa apenas aos dados de treino (baixo viés), não sendo capaz de extrapolar/generalizar os resultados para o teste/novos dados/mundo real (alta variância)
    - Baixa variância → alto viés: **underfitting**

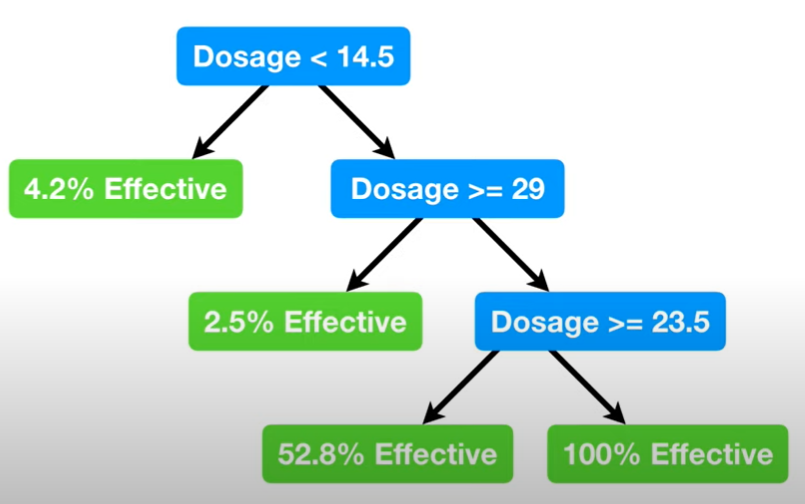
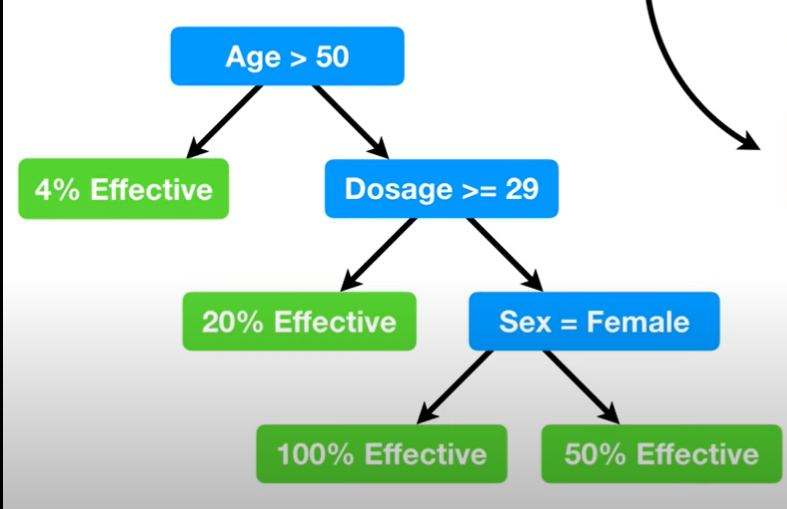
### 

### Decision Trees: Classification and Regression Trees

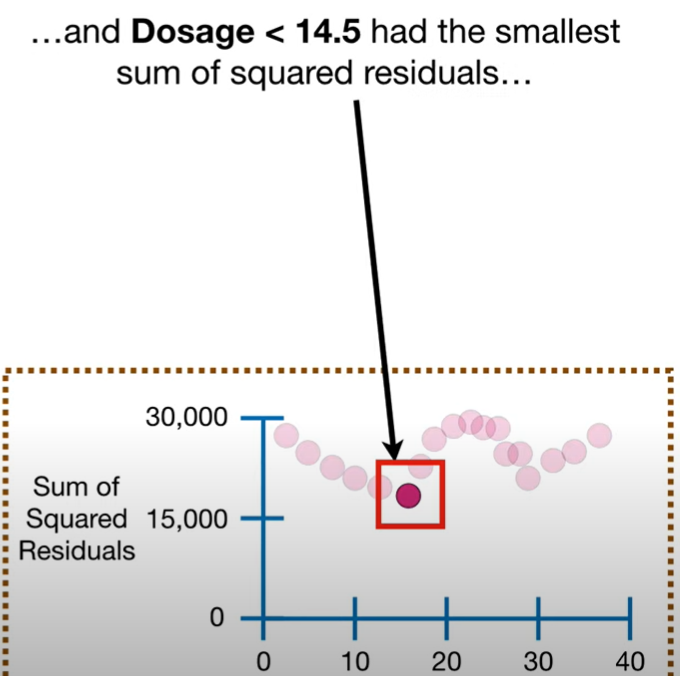
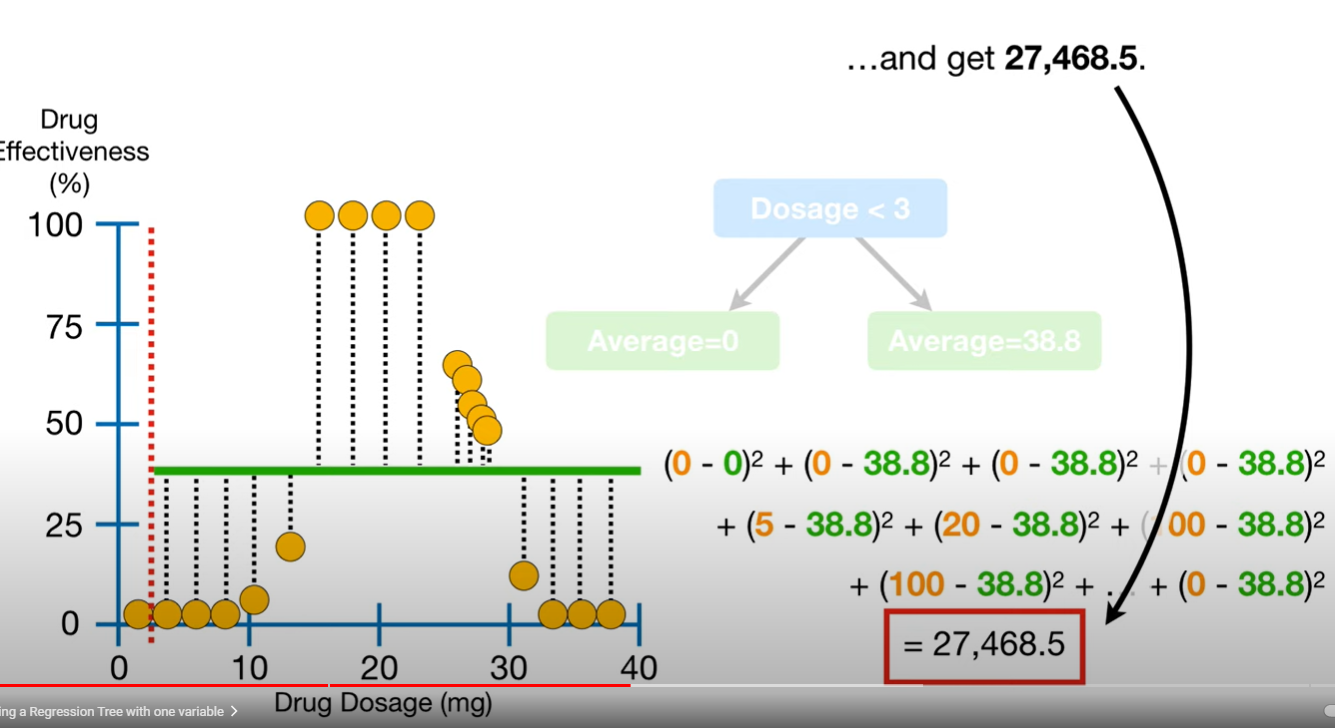
* **Árvores de decisão** → sequência de ifs (perguntas), muito usada para captar **não-linearidades** que modelos lineares não captam bem
  + Algoritmo supervisionado: dados rotulados previamente!
  + Cada cláusula if é um *fork* ou *node* na árvore, que dá origem a dois galhos (*branches*)
  + Cada *node* se baseia em uma *feature* (variável independente), seja ela numérica ou categórica
  + Como há infinitas sequências de perguntas (ainda mais com *datasets* com muitas *features*), encontrar a árvore ideal é computacionalmente muito complexo, havendo vários algoritmos que constroem uma árvore boa, mas não ideal
    - Geralmente, a ordem de perguntas/*features* é definida com base na **quantidade de informação** que ela fornece, o que é medido pela **entropia** ([Wordle 3b1b](https://www.youtube.com/watch?v=v68zYyaEmEA)), uma medida que representa a **incerteza** associada a um conjunto de dados (, onde é a proporção de observações de cada grupo frente ao total de observações).
      * Se todos os dados pertencem a uma só classe → 0 entropia
      * Se os dados estão igualmente distribuídos entre as classes → alta entropia
        + Dados com alta entropia tem uma alta **impureza de Gini**! ()
        + Uma partição dos dados gerados por um nó terá baixa entropia se dividir os dados em subconjuntos com baixa entropia (ou seja, com poucos indivíduos/classes; entropia 0: 1 indivíduo) e alta entropia se gerar subconjuntos grandes ou com alta entropia (***ver pág. 222 Data Science do 0***).



* + Atenção! Sem incorporar técnicas de *bagging* (como o *Random Forest*, que combina várias árvores de decisão), esse modelo tem bastante tendência ao overfitting
  + Exemplo: perfil/cara-a-cara!
* No limite, a árvore mais simples que existe olha para apenas uma característica (por exemplo, o sexo) e fala: "bem, se é homem, vai ser mau pagador; se for mulher, vai pagar direitinho".
  + Contudo, há mulheres que dão calote (e nosso modelo geraria erros do **Tipo II**, ou seja, **falsos negativos**) e homens que pagam em dia (erro **Tipo I** ou **falsos positivos**).
* Árvores de decisão podem ser divididas em duas:
  + **Classificação** (é ou não bom pagador; output *binário*) - [StatQuest](https://www.youtube.com/watch?v=_L39rN6gz7Y)
  + **Regressão** (probabilidade de pagar; output/*leafs* *numérico*) - [StatQuest](https://www.youtube.com/watch?v=g9c66TUylZ4)



* + Os valores dos *leafs* são a média de cada subconjunto (dose < 14.5, 14.5 < dose < 29…)
  + Como medir os pontos que usaremos como *cutoffs* nos *nós* (por que 14.5?)?
    - Usa-se aquele que **minimiza a soma dos quadrados dos resíduos da previsão** (SQR ou SSR, em inglês)**:**



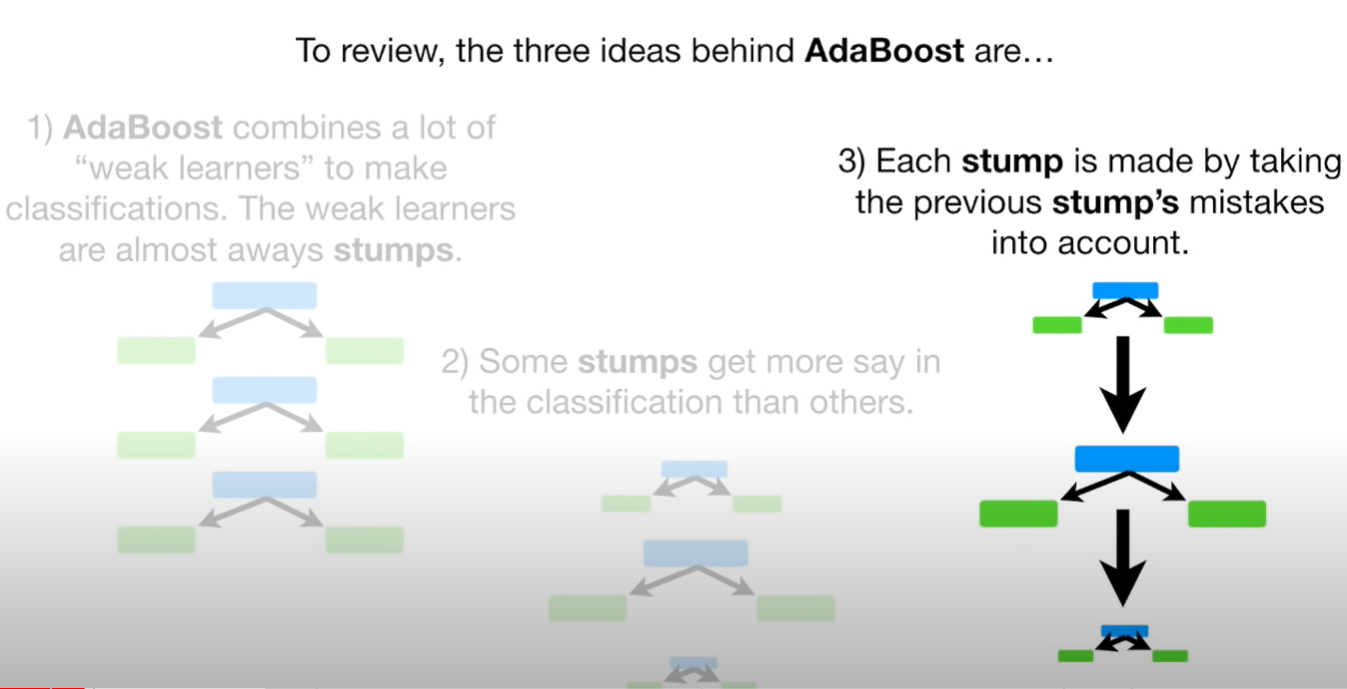
* + - O processo pode ser repetido para cada subconjunto formado (por exemplo, a otimização do *cutoff* do segundo nó levaria em conta apenas a SQR das observações com doses maiores que 14.5)
      * Cuidado com ***overfitting***! O ideal não é ter *leafs* cuja média é composta por apenas uma observação ou árvores com predições “perfeitas” (0 viés, mas possivelmente alta variância).
        + Evitar esse problema: só criar um nó quando houver um número mínimo de observações em cada partição resultante (geralmente 20); caso contrário, o nó será convertido em *leaf* e gerará o output médio da partição
    - E como construir uma árvore com várias *features*?
      * Para escolher a *feature* usada em cada nó, fazemos o processo acima de escolher o ponto/categoria da *feature* que minimiza a SQR das predições geradas em cada uma das duas partições;
      * Fazemos isso para cada *feature*, sendo que o ponto escolhido em cada uma se torna um **candidato** para o nó; o vencedor é aquele que, dentre os vencedores das *features*, possui o menor SQR.
      * O procedimento é replicado para cada nó, até chegar a um ponto em que o número mínimo de observações para realizar uma separação não é mais atingível.
      * Para uma **árvore de classificação**, a ideia é semelhante: **escolhemos para o primeiro nó e (cada nó posterior) a *feature* que melhor classifica (via menor entropia/menor impureza de Gini) os dados restantes**;
        + No caso de *features* numéricas, usamos o ponto de corte (dentre os valores que aparecem no nosso *dataset*) que melhor classifica os dados, o que é medido por aquele que gera a partição com **menor entropia/índice de impureza de Gini** (que é a soma ponderada pelas observações de cada subamostra/partição criada pelo nó)
        + O mesmo procedimento é feito para os nós seguintes, levando em conta apenas a amostra de dados que foi gerada pelo nó anterior da árvore
        + Para os *leafs*, o output será o valor mais recorrente, respeitando a amostra mínima necessária para prevenir o *overfitting*

### 

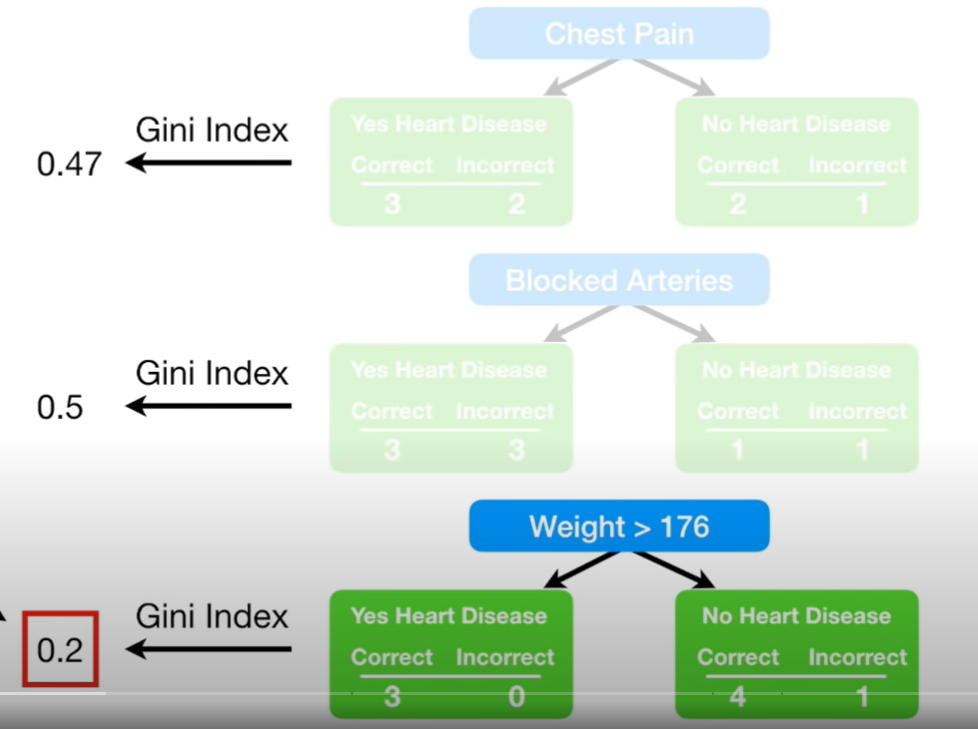
### Random Forests - [StatQuest](https://www.youtube.com/watch?v=LsK-xG1cLYA)

* Árvores aleatórias são usadas para contornar alguns dos problemas das árvores de decisão, que tendem a gerar *overfitting*, não generalizando suas predições a novos dados ou à incorporação de novas categorias na sua variável de interesse
  + Uma floresta aleatória, resumidamente, **cria várias árvores de decisão** (centenas ou milhares) com base em várias amostras de testes (tanto da variável dependente quanto de ***features*, usando conjuntos diferentes de preditoras para cada árvore**).
    - Importante: em cada árvore, cada *feature* pode aparecer mais de uma vez!
    - Ao invés de amostras, podemos usar *bootstraps*  do *dataset*, o que aumenta o número de dados em cada árvore.
      * ***Bootstrap***: a partir de observações aleatórias do *dataset* original (que podem se repetir), cria-se um banco de dados do mesmo tamanho do original, mas com observações diferentes (incluindo repetidas)
        + Na média, ⅓ dos dados originais não irão aparecer em cada *dataset* gerado via *bootstrap*
        + Cada *dataset* será usado por uma árvore!
  + O output do modelo é dado por *majority voting*: se mais de 50% (em caso de categorias binárias) das árvores classificam a entrada como determinado tipo, ele será classificado dessa forma; caso contrário, não.
  + Exemplo de algoritmo de *ensemble learning*, pois combina o resultado de outros algoritmos
    - Terminologia: *bootstrapping*  + *ensemble* (usando alguma agregação de resultados como *output* final): ***bagging!***
  + Como o *bootstrap* deixa de fora alguns dados em cada árvore de decisão, podemos usá-los para testar a acurácia da *random forest*
    - Terminologia: dados que ficaram de fora → ***out-of-bag sample***
    - Para cada *dataset* criado, usamos sua respectiva *out-of-bag sample* nas árvores que utilizam o mesmo conjunto de dados, vendo se a maioria das árvores (que têm a mesma *out-of-bag sample*) a classifica corretamente
      * Quanto maior o % de *out-of-bag samples* classificada corretamente, maior a acurácia da floresta (menor o ***out-of-bag error***)
      * O número ideal de *features* por árvore pode ser determinado sendo aquele que minimiza o *OoB Error*

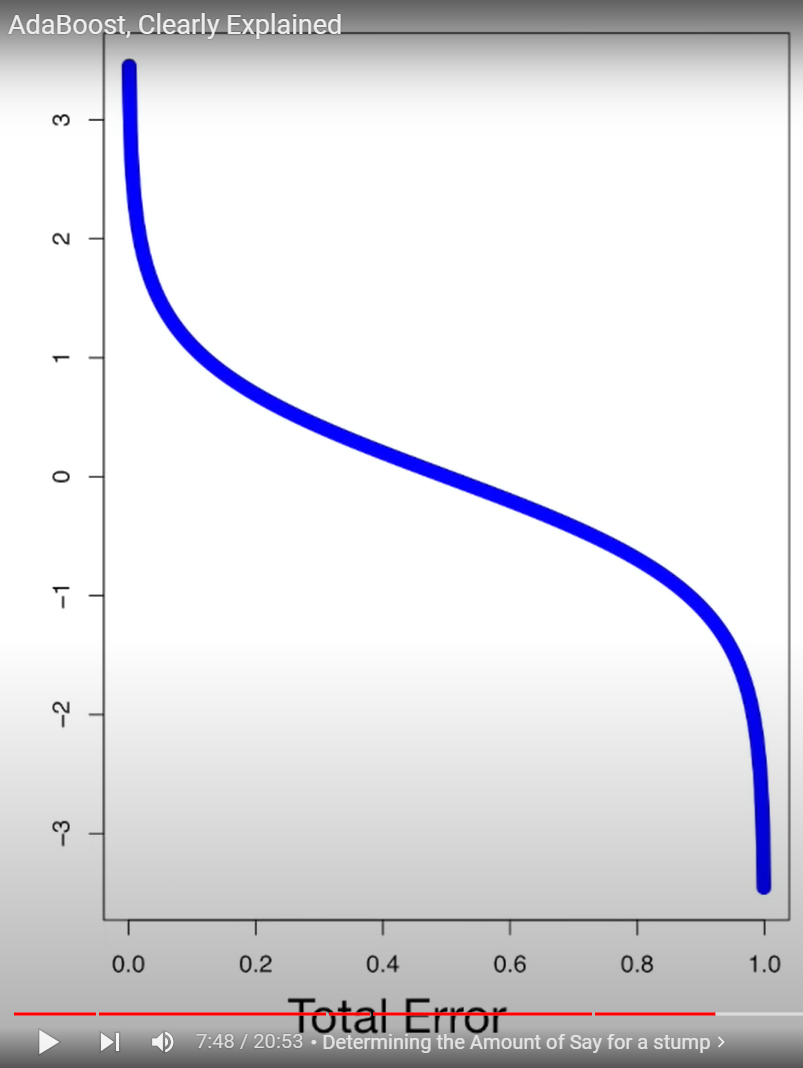
### AdaBoost - [StatQuest](https://www.youtube.com/watch?v=LsK-xG1cLYA)



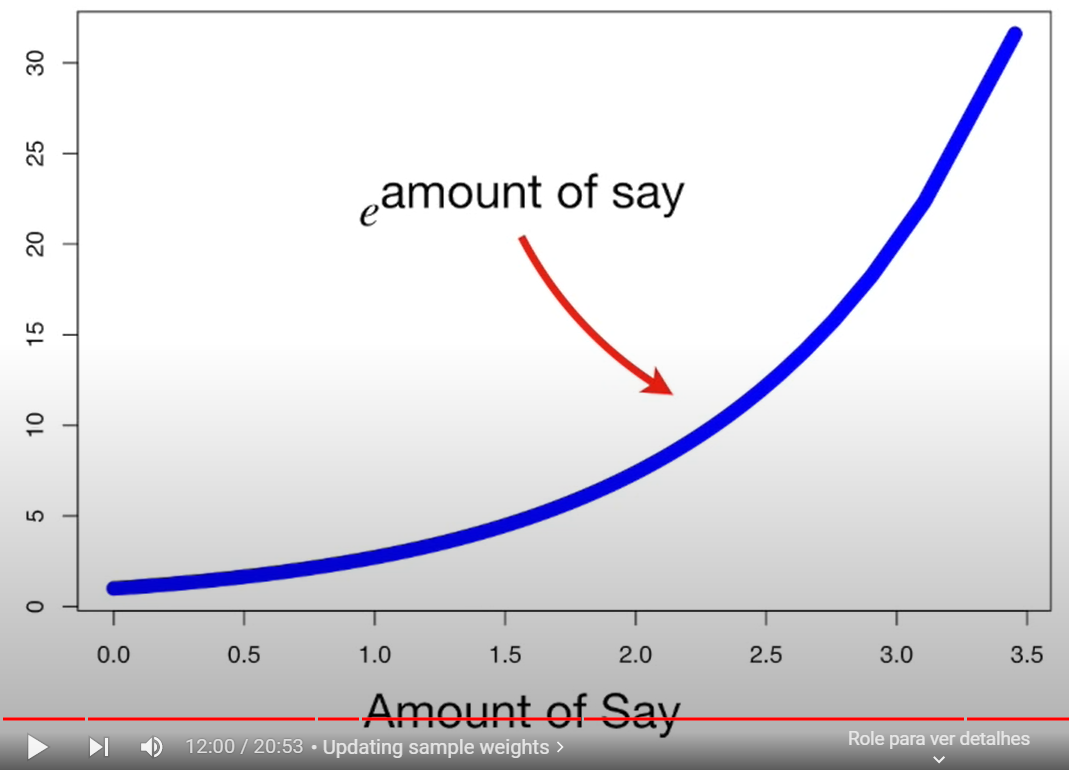
* ***Boosting***: transformar “*weak learners*” em “*strong learners*”
* Ao contrário de *Random Forests*, *AdaBoost* não possui árvores com mais de um nó (chamadas de “cotocos”/*stumps*) (“*weak learners*”)
* O primeiro passo após o recebimento/processamento dos dados é a atribuição de **pesos** para cada observação. Num primeiro momento, cada observação recebe um peso igual (), tornando-as igualmente importantes.
* O primeiro *stump* é feito de forma parecida ao primeiro nó em uma árvore de classificação: dentre as *features*, **escolhemos aquela que produz o menor índice de impureza de Gini** (), ou, em outras palavras, que **melhor classifica a nossa variável de interesse[[1]](#footnote-0)**.



* + Lembrete: o índice é calculado ponderando a entropia de cada uma das partições (em geral, duas) geradas pelo *stump* pelo número de observações em cada partição
  + Em tese, teríamos de calcular o índice *ponderado* de Gini, haja vista os pesos que cada observação tem (não é importante para o 1o *stump*).
  + Atenção para variáveis numéricas: o ponto de corte (dentre os observados no *dataset*) é aquele que minimiza a entropia/Gini.
* Após a criação do primeiro *stump* da floresta (que sempre conta com apenas uma *feature*), começamos a ver alguns dos princípios do *AdaBoost*:
  + ***Not all stumps have equal say***: ao contrário da *Random Forest*, que funciona com votação majoritária, cada *stump* tem um “voto” exponencialmente proporcional à sua taxa de acerto, ou seja, à proporção de classificações corretas feitas.
    - O **erro total (TE)** de uma *stump* é definido como a *soma dos pesos das observações incorretamente classificadas*
    - O ***Amount of Say*** (AS) é definido como
      * Se TE=50% (quase randômico), AS=0
      * AS < 0 (TE alto) indicam que o *AdaBoost* interpretará o contrário da classificação predita pela *stump*



* + ***Adaptação das Stumps***: a próxima *stump* fará um esforço adicional para acertar a classificação das observações que a *stump* anterior errou, o que é feito ajustando os pesos
    - Os pesos das observações classificadas erroneamente aumentam: ; quanto maior for o *Amount of Say* da *stump* anterior (menor o erro total), o peso da observação classificada de forma errada aumenta exponencialmente:



* + - Os das classificadas corretamente, diminuem: ; seu peso decresce de forma exponencial com o grau de acerto do *stump* anterior
    - Posteriormente, os pesos são regularizados/escalados para que sua soma dê 1;
    - Para o próximo *stump*, podemos fazer o mesmo processo de selecionar a *feature* que gera o menor índice de Gini, mas levando em conta agora os diferentes pesos das variáveis (um erro de classificação de observação que já eram erradas na rodada anterior penaliza mais a *feature*, ou seja, gera um aumento maior em seu Gini)
      * Alternativamente, podemos gerar um novo conjunto de dados via *bootstrap*, ou seja, preservando o tamanho original do *dataset* original, mas mudando as observações
      * Para isso, usamos os pesos como uma função de distribuição acumulada e geramos números aleatórios entre 0 e 1: como observações erradas na rodada anterior terão mais peso, elas terão maior frequência no novo *dataset* gerado por *bootstrap*
        + No novo *dataset*, todas as observações passam a ter pesos iguais e realizamos o procedimento de forma idêntica ao do primeiro *stump*, mas com esse novo conjunto de dados
    - O processo de adaptação continua ao longo dos *stumps* que formarão nossa “floresta”.
* A classificação/predição via *AdaBoost* ocorre a partir da soma das *Amount of Say* de cada *stump*: ao jogar no modelo uma nova observação que queremos prever, parte das *stumps* irá classificá-la como TRUE e parte como FALSE;
  + A classificação final dependerá da soma dos AS: se a soma for maior para as *stumps* que disseram TRUE, a nova observação terá sua classificação prevista como TRUE

### Gradient Boosting (for Regression) - [StatQuest1](https://www.youtube.com/watch?v=3CC4N4z3GJc) e [StatQuest2](https://www.youtube.com/watch?v=2xudPOBz-vs)

* Gradient Boosting é bem similar ao *AdaBoost*, mas com algumas diferenças notáveis:
  + A primeira parte da floresta não é um *stump* (que popula toda a floresta em um *AdaBoost*), mas sim **apenas uma *leaf***, que contém a **média** da variável dependente no *dataset* de treino
  + A partir da 2a árvore, o *Gradient Boosting* constrói estruturas que variam de 8 a 32 *leafs* (ou seja, com 3 a 5 níveis de nós), enquanto o *AdaBoost* continua com *stumps*
    - Além disso, enquanto o *AdaBoost* tenta prever sempre o valor da variável dependente — ajustando pelos pesos criados a partir do erro da *stump* anterior —, o *Gradient Boosting* tenta, a partir da 2a árvore, prever os ***pseudo-resíduos* gerados pela árvore anterior** (, para cada observação )
      * Idealmente, cada nova árvore deve reduzir todos os (ou a maioria dos) resíduos gerados pela previsão, melhorando o *fit* do modelo
        + De forma matematicamente mais rigorosa, o *Gradient Boosting* tenta, a cada árvore, minimizar sua ***função de perda***: (metade da soma dos quadrados dos resíduos, que é a função de perda de uma regressão linear)

A fração é usada para cancelar com o que surge da *derivada* da função com relação a

* + - * Assim como em uma *regression tree*, os resultados finais de uma *leaf* com mais de uma observação é a média delas
    - Para não gerar *overfitting*, cada árvore é multiplicada por uma ***learning rate*** (), que pode ser interpretada como a contribuição (fixa) de cada árvore para a previsão final
      * Vários pequenos passos no caminho certo geram melhores previsões (e menor variância)!
    - O resultado final do *Gradient Boosting* é uma combinação linear: partindo da média, soma-se as predições de cada árvore multiplicadas pela *learning rate* para se chegar à previsão.
      * O número final de árvores ou é predeterminado (geralmente, 100) ou é determinado pelo *Gradient Boost* quando árvores adicionais não melhoram o modelo
* **IMPORTANTE!** O modelo funciona melhor quando as *features* numéricas estão **escaladas/regularizadas** (z-score ou divisão min-max)
  + *Features* binárias: criação de categorias *dummies*!

## Referências (abrir com *Liner* no Chrome!)

* [Parâmetros LightGBM](https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc#id_token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjIwOWMwNTdkM2JkZDhjMDhmMmQ1NzM5Nzg4NjMyNjczZjdjNjI0MGYiLCJ0eXAiOiJKV1QifQ..cUF-xmakPVJLT3mbilPOoATFWVx7pBOBOjfm6gYIHKhFXVVqT6CuWTtN33BQbBmX2r6_MYkY2qF6UFE309ZJNOhbCIb0Ak2Drm6-2tnqMmi-EAUcGI6X_olJ5ZmMLnmktWveZvQxfblLPAJpqi5vjQY1EBl0KLZ8W8DFmgNHV73PyAwaHFjXp_8c0yibUw0JwiUWzM8pgbyoBSKcr7D0LiaSxHqQZdyFOF62iPqcw9lNFSLSfXcL1WDdAOSkMv8Qkdwgl88EM_QfUb8vgR1RfkZvY5ZM0LOIPbKZAo-OqBeOm_opPEPizpp6ySHXVmE1xW_V2B3Zu6qy_cdQXsJ8IQ)
* [XGBoost para previsões de séries temporais](https://machinelearningmastery.com/xgboost-for-time-series-forecasting/)
* Ideia de LightGBM para previsão: usar um *shift/lag* para usar os valores anteriores da série como uma *feature* (podemos usar o valor imediatamente anterior ou uma média dos últimos períodos, por exemplo, além de usar múltiplos *lags* como *inputs*), em conjunto com outras *features* (sazonalidade, calendário, renda…) com ou sem *lags*
  + No R: usar lubridate para coletar informações de sazonalidade, como ano, mês, dia, se é fim de semana ou não…

1. Apenas relembrando que estamos tratando de aprendizado supervisionado, ou seja, com dados rotulados, e, por isso, o ideal é separar os dados em conjuntos de *treino* e *teste*. [↑](#footnote-ref-0)