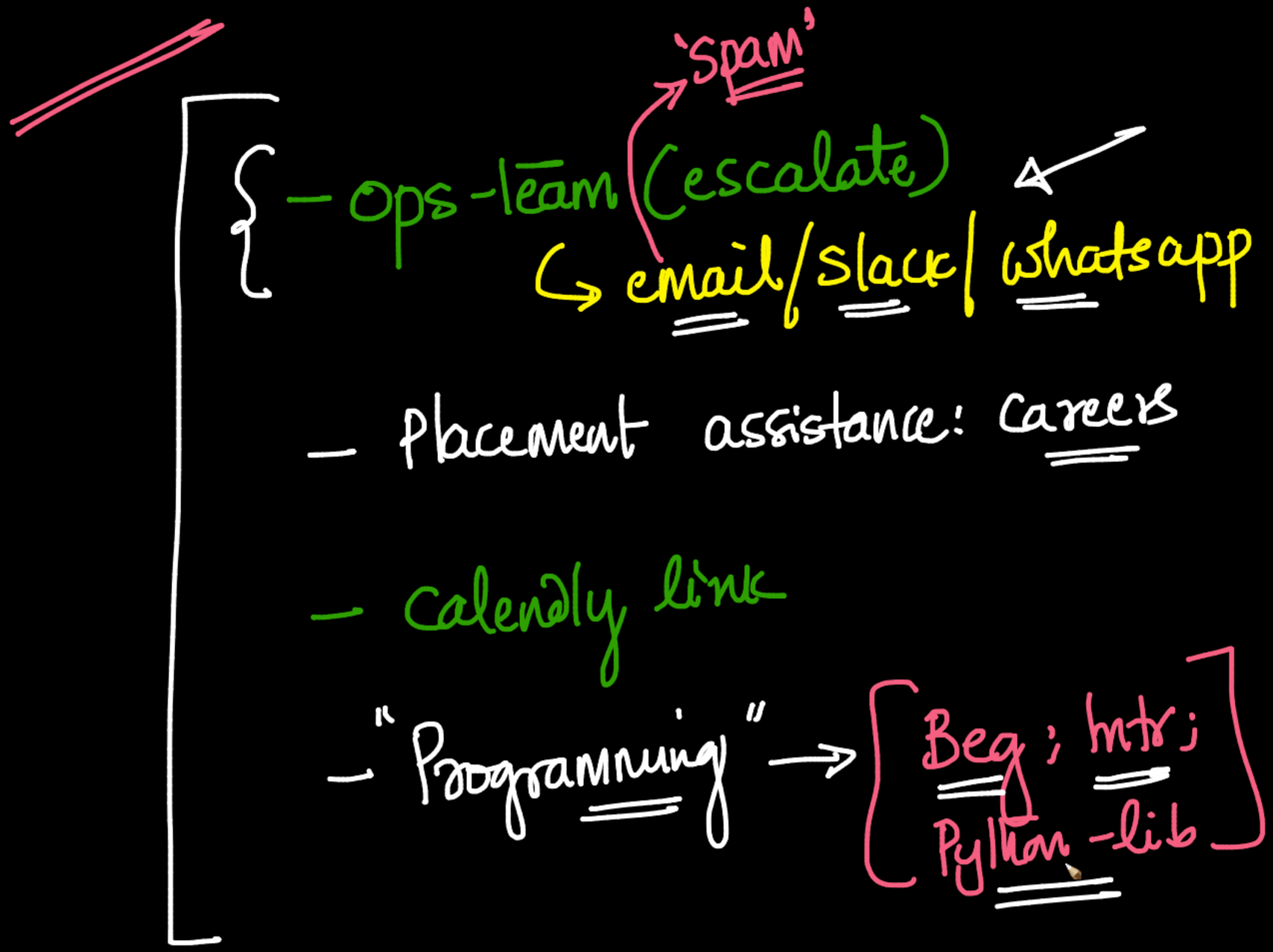
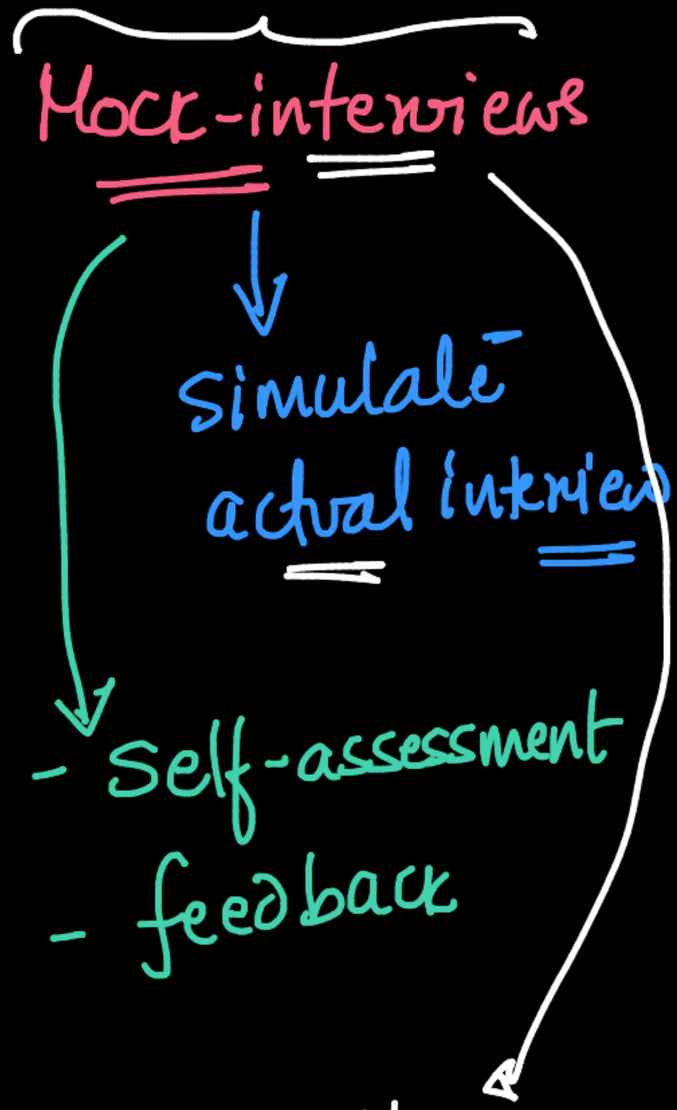
Topics:

- ✓ Mock-interviews
- [revised]
  - GBDT
    - Training algo ✓
    - Bias-variance tradeoff ✓
    - variations ✓
  - Intuition behind AdaBoost
- ✓
  - XgBoost ✓
  - Light GBM ✓


'Spam'
- ops-team (escalate)
↳ email/slack/whatsapp
- Placement assistance: Careers
- Calendly link
- "Programming" →
Beg ; Intr ;
Python -lib


Mock-interviews
Simulate actual interview
- Self-assessment
- feedback
E-M-H

Optional Module: Adv DSA for MLE → Greedy; DP; Graphs

pseudo-residual

{already trained $m-1$ classifiers} → $f_{m-1}(x_i)$

$i : 1 \rightarrow n$ train points

$1 \rightarrow M$

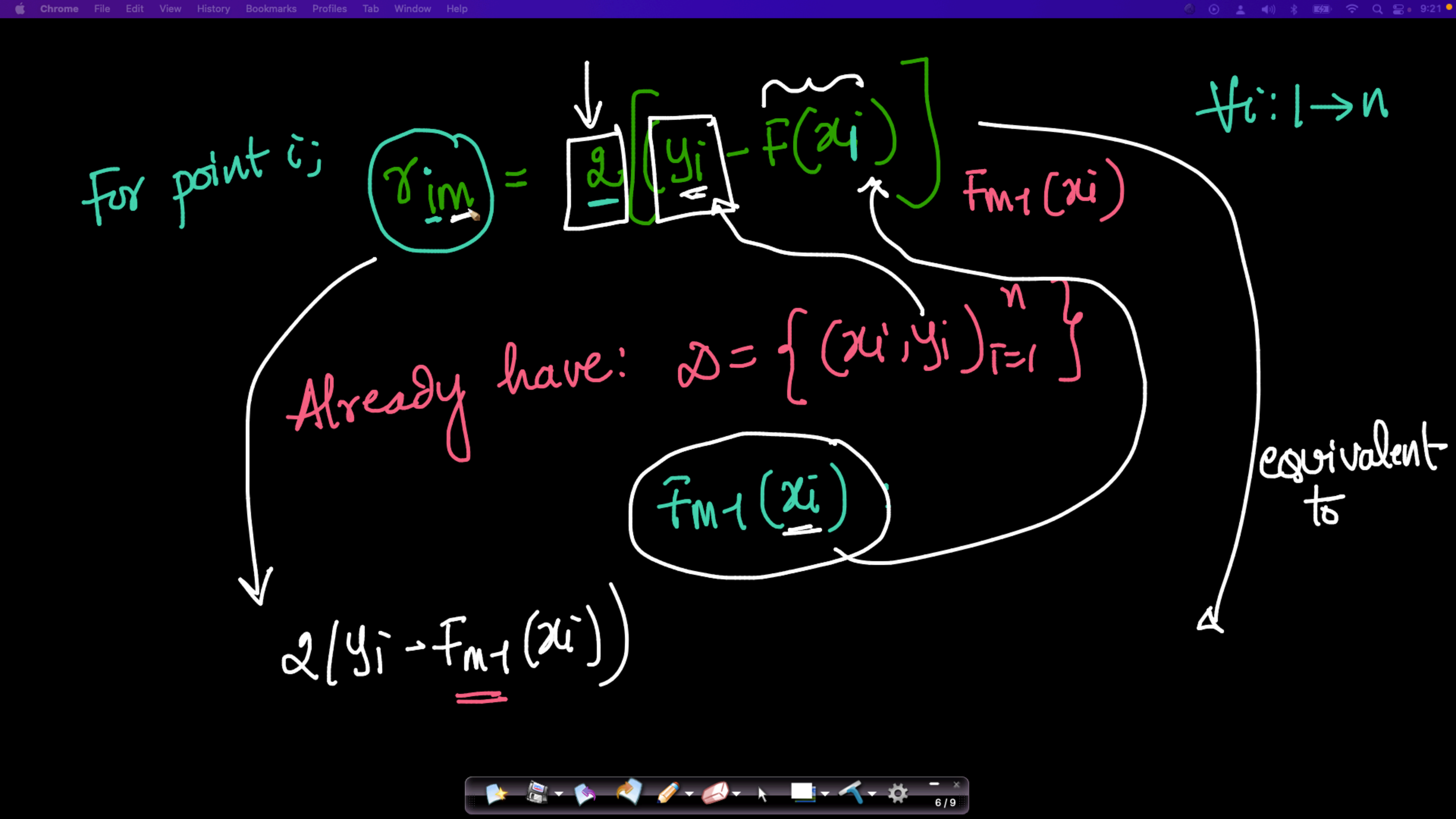
m^{th} classifier

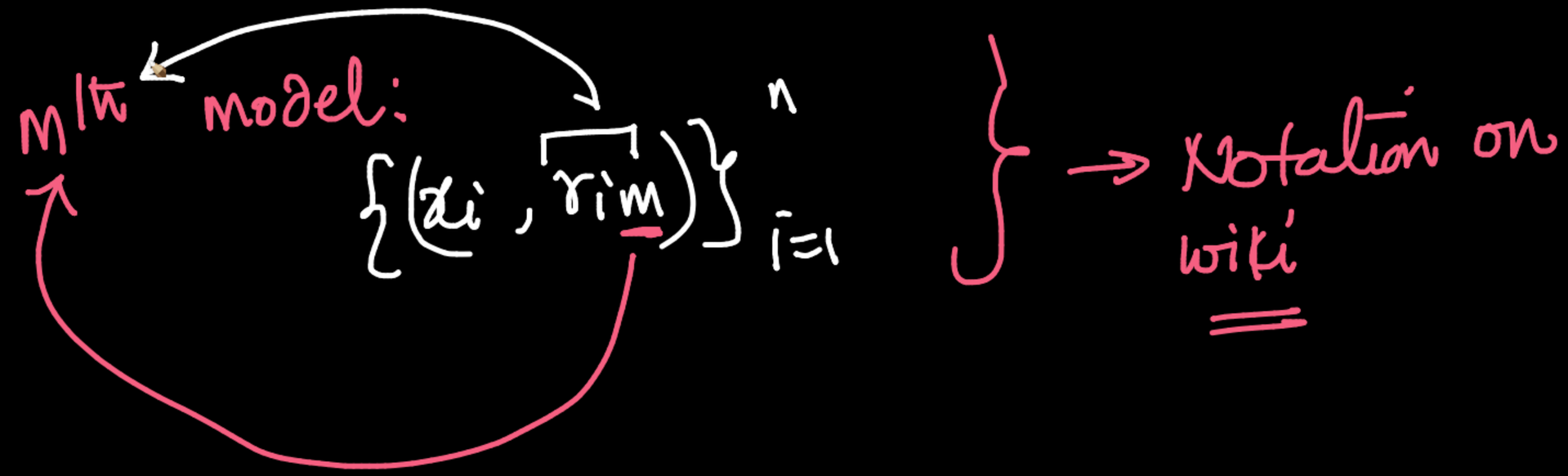
$$\gamma_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{f_{m-1}(x_i)}$$

Loss : regression ⇒ Squared - loss

$$L(y_i, f(x_i)) = \{y_i - f(x_i)\}^2$$

$$\frac{\partial L}{\partial f(x_i)} = -2(y_i - f(x_i)) \quad (\text{Prev Class})$$


For point i;
r_{im} =
2
y_i - $F(x_i)$
$\forall i: 1 \to n$
$F_{m-1}(x_i)$
Already have: $\mathcal{D} = \{(x_i, y_i)_{i=1}^{n}\}$
$F_{m-1}(x_i)$
equivalent to
$2(y_i - F_{m-1}(x_i))$



$x_i \in \mathbb{R}^d$

$y_i \in \mathbb{R}^1$ ← scalar

Idea:

$$\text{pseudo-residual} = -\frac{\partial L}{\partial f(x_i)} \approx \text{residual}$$

any Loss function that is differentiable

$$F_{m-1}(x_i) = \alpha_0 h_0(x_i) + \alpha_1 h_1(x_i) + \cdots + \alpha_{m-1} h_{m-1}(x_i)$$

$(m-1)^{th}$ iteration

$(m-1)^{th}$ base-model

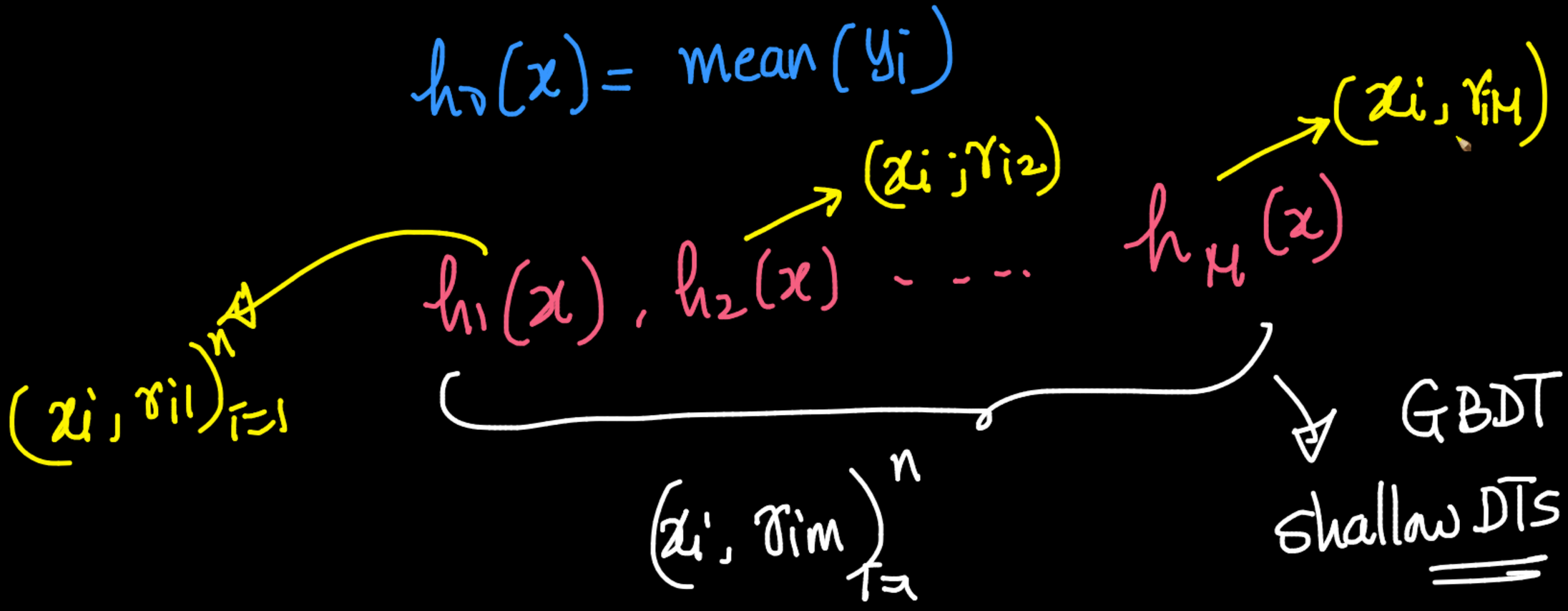
$$F_m(x_i) = \frac{\gamma_0 h_0(x) + \gamma_1 h_1(x) + \gamma_2 h_2(x) \cdots}{\gamma_{m-1} h_{m-1}(x) + \gamma_m h_m(x)}$$

m^{th} - iteration

$\hookrightarrow (x_i, r_{im})_{i=1}^{n} \longrightarrow \boxed{h_m(x)}$ ✓

also compute γ_m (later)

$\boxed{= 1}$


$h_0(x) = \text{mean}(y_i)$
$(x_i, r_{i1})_{i=1}^{n}$
$h_1(x), h_2(x) \sim \cdots$
(x_i, r_{i2})
$h_M(x)$
(x_i, r_{iM})
$(x_i, r_{im})_{i=1}^{n}$
GBDT
Shallow DTs

pseudo residuals are used to train

M - base learners

$$F_M(x) = \sum_{i=0}^{M} \gamma_i h_i(x) = y$$

h_i: shallow DT

Test time:

$$x_q \rightarrow \sum_{i=0}^{M} \gamma_i h_i(x_q) = \hat{y}_q$$

Let

$$F_M(x) = \underset{12}{\gamma_0 h_0(x)} + \underset{1}{\gamma_1 h_1(x)} + \underset{1}{\gamma_2 h_2(x)} + \cdots + \underset{1}{\gamma_M h_M(x)}$$

$h_0(x_i) = 12.0$

$h_1(x_i) = 4.2$

14.2

10

$h_2(x_i)$

$\vdots$

$= 0$

$h_M(x_i)$ ← hyper-param

$i = 10$ (let)

$x_i = [1, 2, 1, 0]$

$y_i = 26.2$ ✓

residuals

$y_i \approx \hat{y}_{iM}$

$M \uparrow$ overfit $\uparrow$

Overfitting vs underfitting

Let
- $h_0(x)$ = extremely simple = mean-model
- $h_1(x)$: deep DT $\Rightarrow$ overfitting
- $\vdots$
- $h_5(x)$: "

$h_0(x)$: Mean-model

↑ train data only

No CV or test data

Classification

$$(x_i, y_i)_{i=1}^{n}$$

$y_i =$ 0 or 1

$p_i = P(y_i = 1)$

$f_M(x_i) \longrightarrow p_i$

if $x_i \in$ +ve

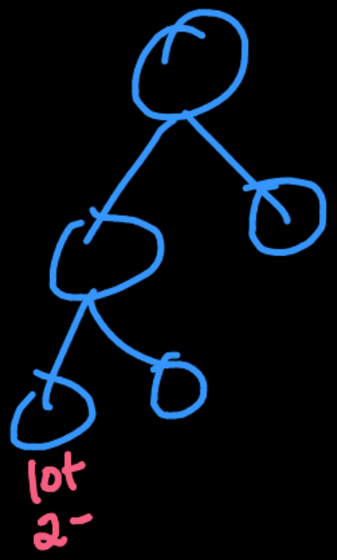
$p_i \to 1$

else $p_i \to 0$

Loss-function for classfn: log-loss

$$\mathcal{L}(y_i, F(x_i)) = \text{log-loss}$$

$$\frac{-\partial \mathcal{L}}{\partial F(x)} = p_i - y_i \quad \text{(proved in past-read vides)}$$



$h_0(x_i) = 0.4$ 0.6

$h_1(x_i) = 0.2$ 0.4

$h_2(x_i) = 0.31$

$x_i = [0, 1, 3, 1]$

$y_i = 1$

$x_i = [1, 2, 0, 1]$

$y_i = 0$

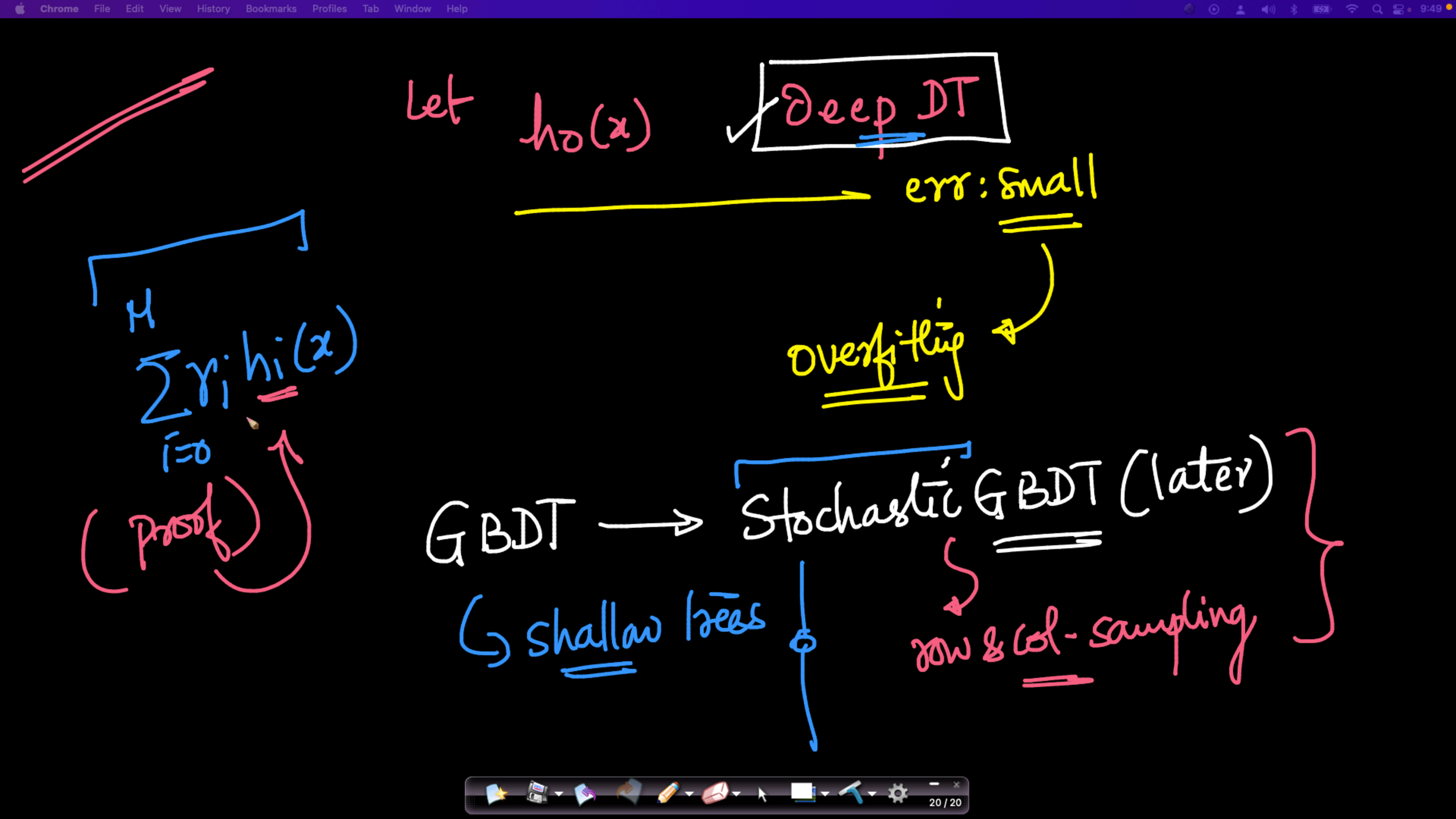
$h_0(x) = 0.4$ → -0.4

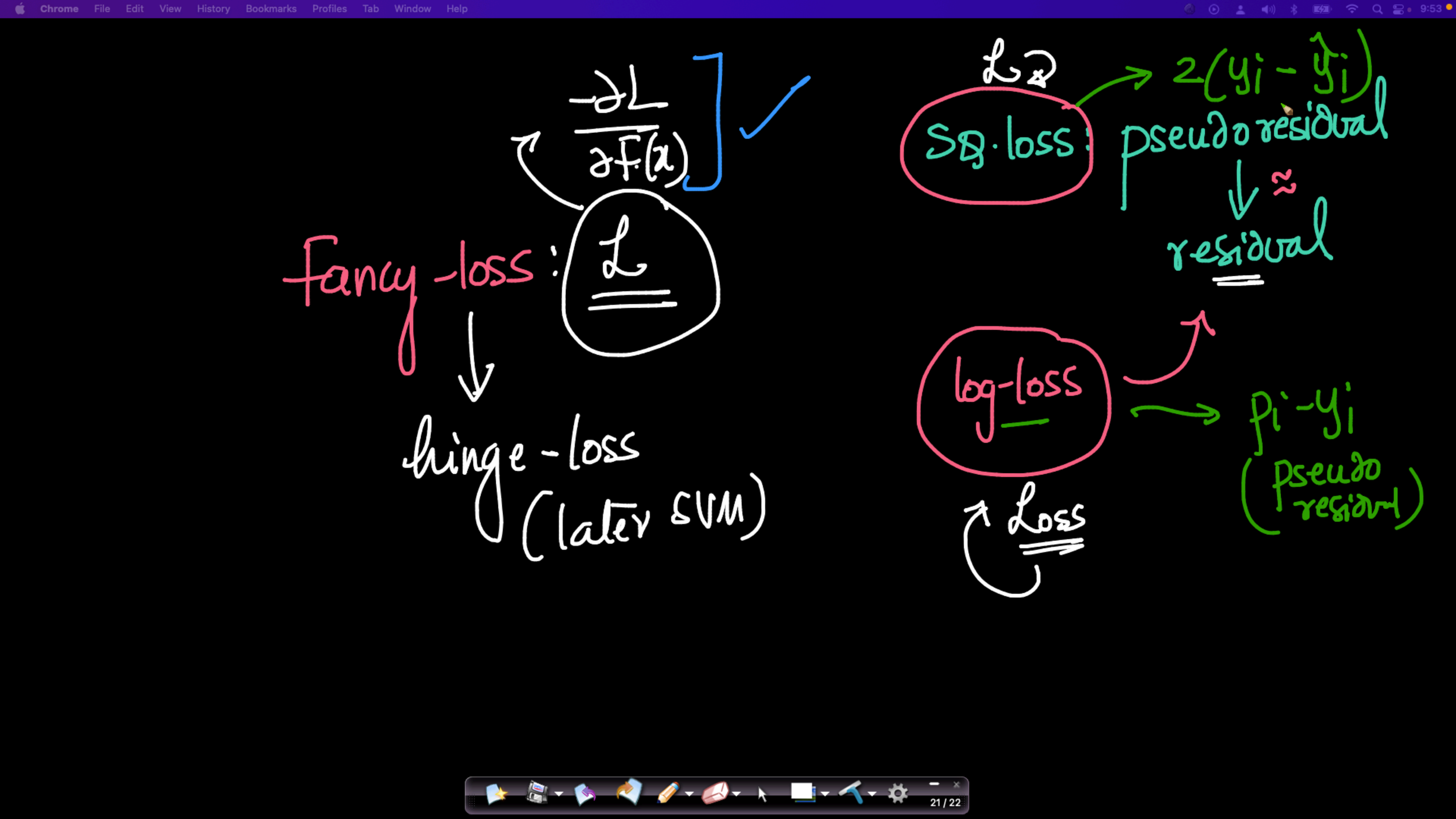
$h_1(x) = -0.1$ → -0.3

$h_2(x) = -0.21$ → -0.09

$h_3(x) =$

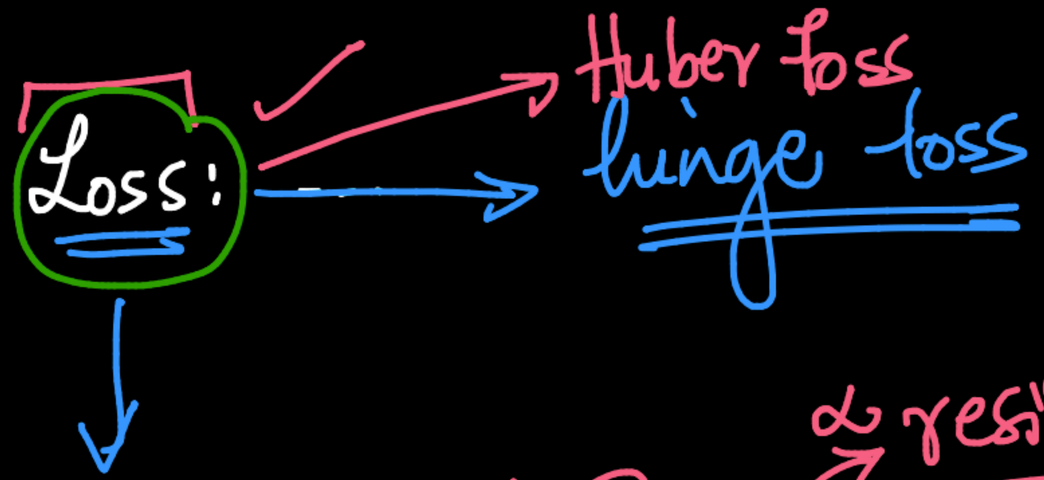
⋮


Let $h_0(x)$
Deep DT
err: small
overfitting
H
$\sum_{i=0} \gamma_i h_i(x)$
(proof)
GBDT → Stochastic GBDT (later)
shallow trees
row & col-sampling


$\frac{\partial L}{\partial F(x)}$
Fancy-loss : L
hinge-loss
(later SVM)
L2
SE-loss
$2(y_i - \hat{y}_i)$
pseudo residual
≈
residual
log-loss
$p_i - y_i$
(pseudo residual)
Loss

Real-world:

Data:



$$\frac{-\partial L}{\partial f(x_i)} = \text{function}$$

→ residual

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

$h_0(x) = \text{mean}$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

mean fw SQ.loss

n/3 +ve
2n/3 -ve
p(y_i=1) fw log-loss
↳ 1/3 0.33

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

$$f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

$$= \arg\min_{\gamma} \underbrace{\sum_{i=1}^{n} (y_i - \gamma)^2}_{g(\gamma)}$$

γ ↓ scalar

$$\frac{\partial g(\gamma)}{\partial \gamma} = \sum_{i=1}^{n} -2(y_i - \gamma) = 0$$

$$\Rightarrow \sum_{i=1}^{n} y_i = n\gamma$$

$$\Rightarrow \frac{1}{n} \sum_{i=1}^{n} y_i = \gamma$$

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

$$\frac{\partial L}{\partial \gamma} = 0$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

m: 1 → M

2. For $m = 1$ to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

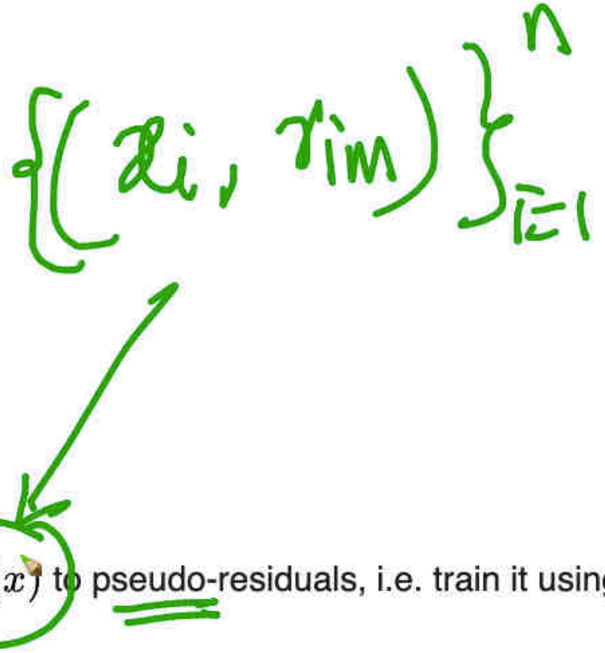
   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.



Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

$$f_{M-1}(x) = f_0(x) + \sum_{i=1}^{M-1} \gamma_i h_i(x)$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

$f_m = 1 \cdot h_0(x) + \gamma_1 h_1(x) + \ldots + \gamma_{m-1} h_{m-1}(x) + \gamma_m h_m(x)$

Gradient tree

regression

$g(\gamma)$

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} \mathcal{L}\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right)$$

γ_m = weight on $h_m(x)$

$\mathcal{L}$ → dataset → 12.2 → fw i=0

$F_{m-1}(x_i)$ → 10.2

γ → only variable

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree boosting [edit]

Gradient boosting is typically used with decision trees (especially CART trees) of a fixed size as base learners. For this special case, Friedman proposed a modification to gradient boosting method which improves the quality of fit of each base

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree boosting [edit]

Gradient boosting is typically used with decision trees (especially CART trees) of a fixed size as base learners. For this special case, Friedman proposed a modification to gradient boosting method which improves the quality of fit of each base

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:

   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree boosting [edit]

Gradient boosting is typically used with decision trees (especially CART trees) of a fixed size as base learners. For this special case, Friedman proposed a modification to gradient boosting method which improves the quality of fit of each base

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

Loss!

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

g(...)

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

A(γ)

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient tree boosting [edit]

Gradient boosting is

es) of a fixed size as base learners. For this

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

GBDT

Loss: → Yes → hack in some cases :-)

SVM

Gradient tree boosting [edit]

Gradient boosting is 44 / 44 es) of a fixed size as base learners. For this

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

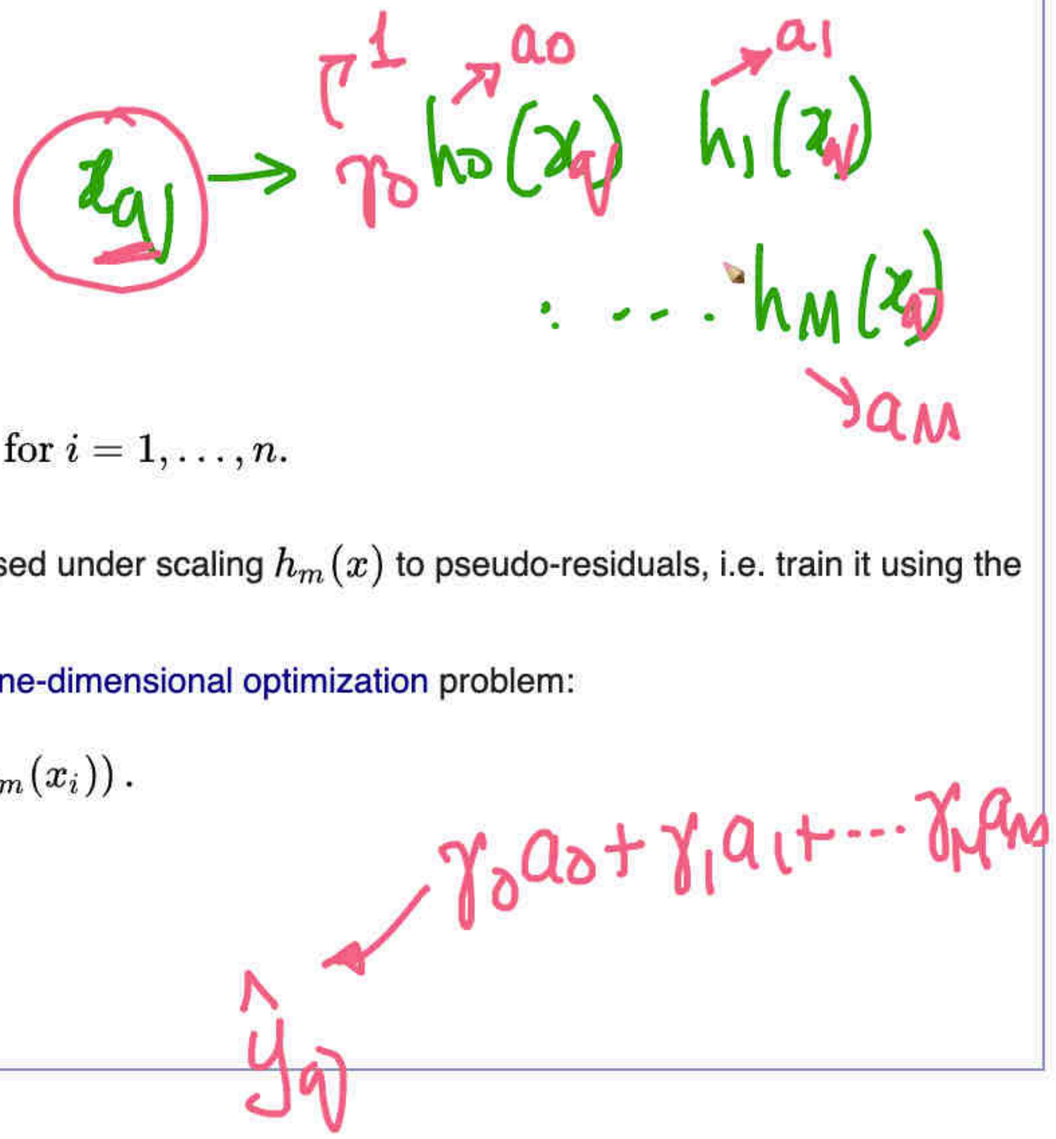
   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.



Gradient tree boosting [edit]

Gradient boosting is typically used with decision trees (especially es) of a fixed size as base learners. For this

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

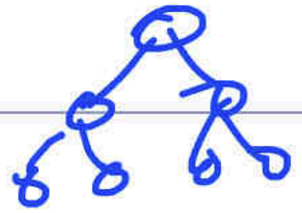
$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

GBDT → all rows & cols

all the features

$h_1(x)$ $h_2(x)$ ... h



Gradient tree boosting [edit]

Gradient boosting is [redacted] es) of a fixed size as base learners. For this

Overfitting - Underfitting

Regularization → avoid overfitting

1. M = # base learners — $M \uparrow$ overfit ↑
2. depth of base-learners ↑ — overfit ↑
   - ↳ n_samples / n_leaf_nodes

Risk of boosting → overfit very easily if not careful

3 reg/ Shrinkage/learning-rate

additional levey

Shrinkage: ν : hyperparam $0 \le \nu \le 1$

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m h_m(x)$$

$\gamma_m h_m(x)$ → pseudo residuals

derivatives

$\nu \uparrow$ overfitting ↑

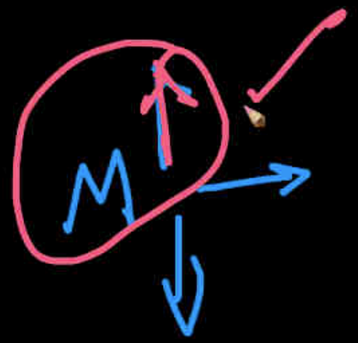
$\nu = 0$ → $h_0(x)$ ⇒ Underfit Model

$\nu = 1$

overfitting increases

GBDT ⟶ M, Max depth of tree : v } ✓

hyperparams



same performance

$\gamma = [\gamma_1 \checkmark \; \gamma_2 \checkmark \; \cdots \; \gamma_M]$ L_2-reg on (γ_M's) $\longrightarrow$ Yes

reg: $\mathcal{L}(y_i, f_M(x_i)) + \alpha \|\gamma\| + \beta(\#\text{leaf nodes})$

multi-model $\longrightarrow$ Control/regularize γ_M

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

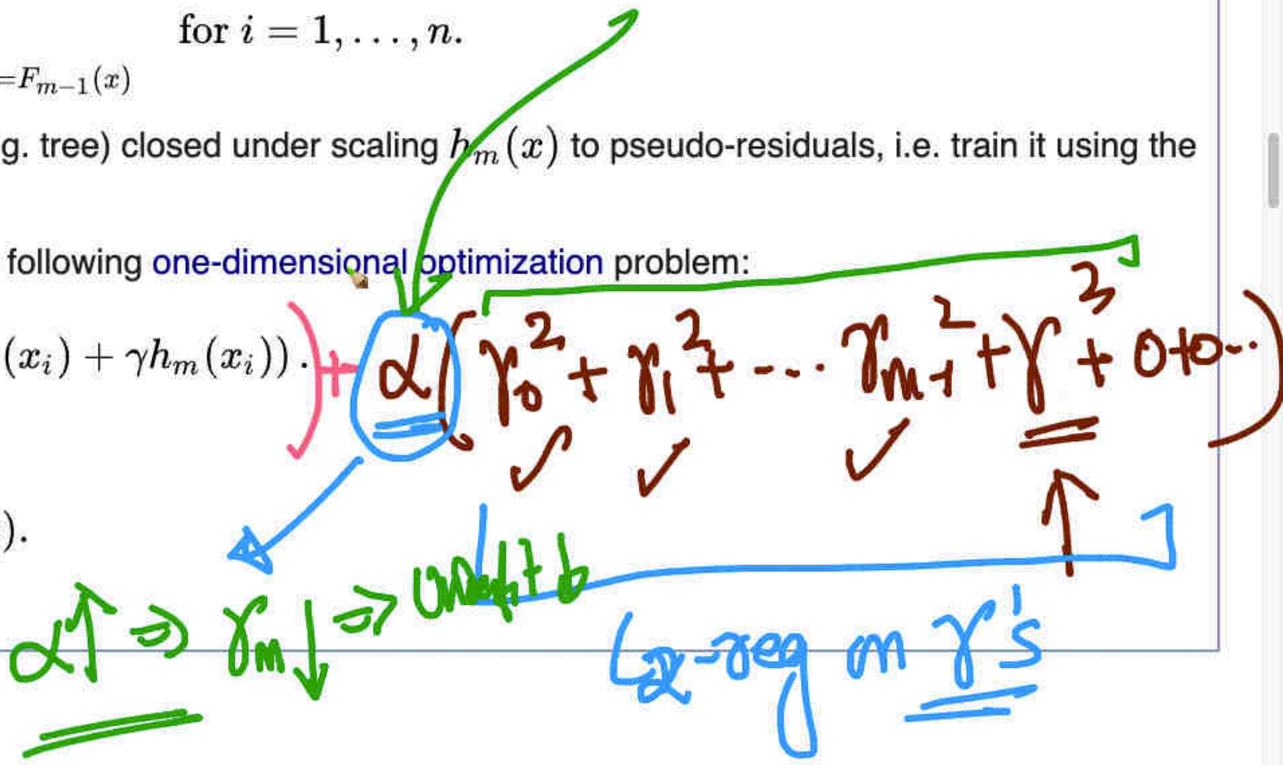
   2. Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

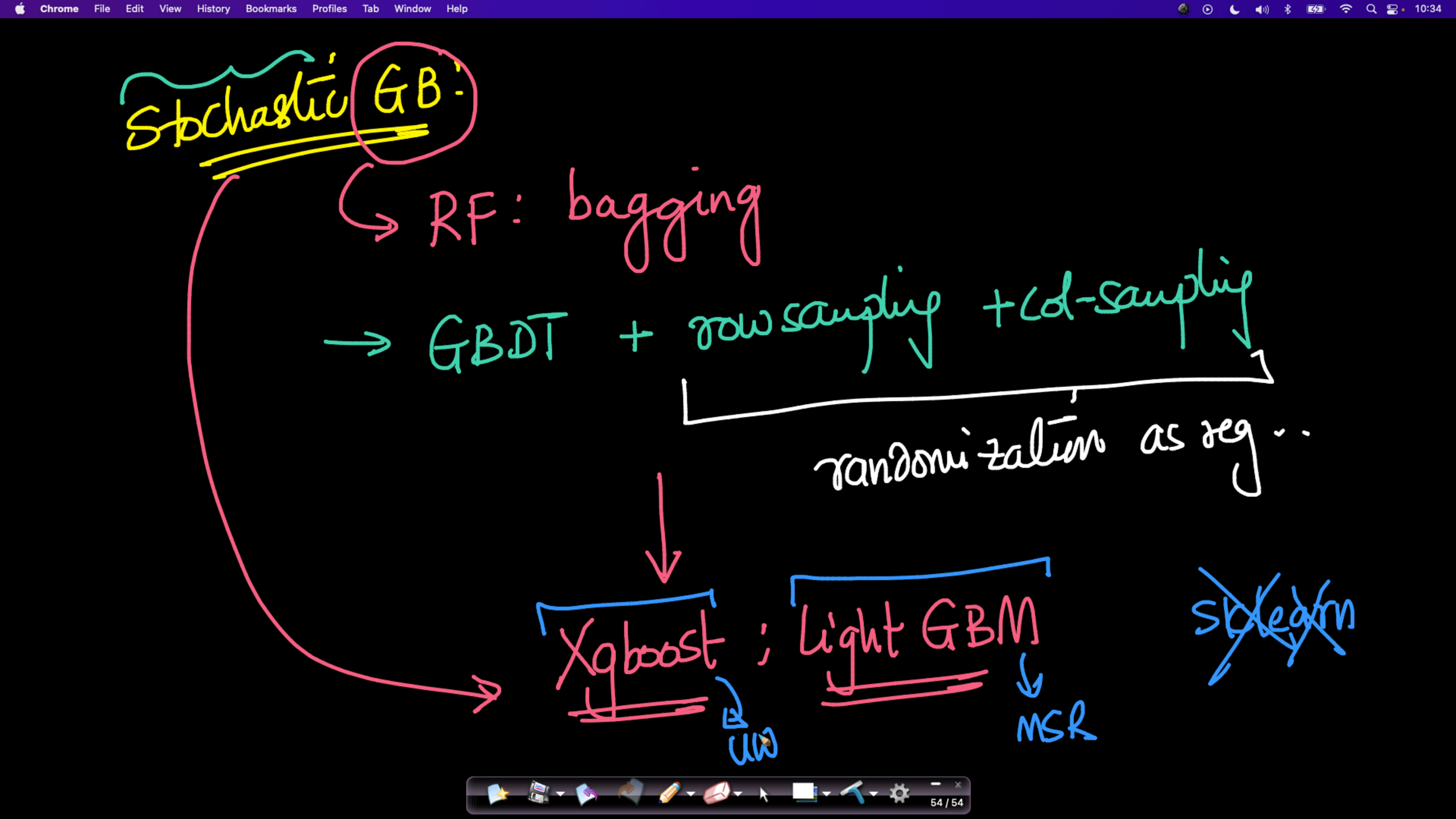
$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.



Gradient tree boosting [edit]

Gradient boosting is t es) of a fixed size as base learners. For this special case, Friedman proposes a modification to gradient boosting method which improves the quality of fit of each base


Stochastic GB:
RF: bagging
GBDT + row sampling + col-sampling
randomization as reg..
Xgboost ; Light GBM
MSR
54 / 54

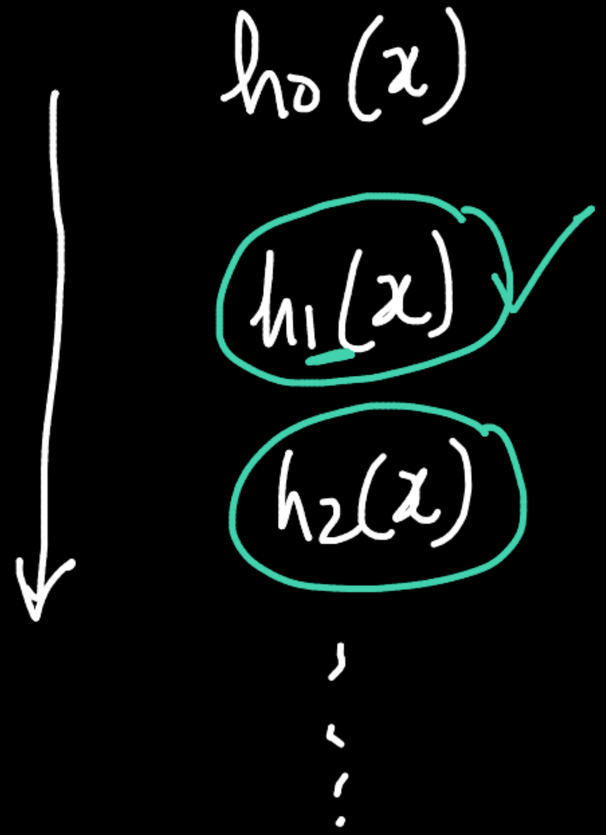
Train time

much faster [RF :- trivially parallelize

M_1 M_2 ---- M_K → different cores / system ✓

GBDT → Sequential - model

Stochastic GBDT

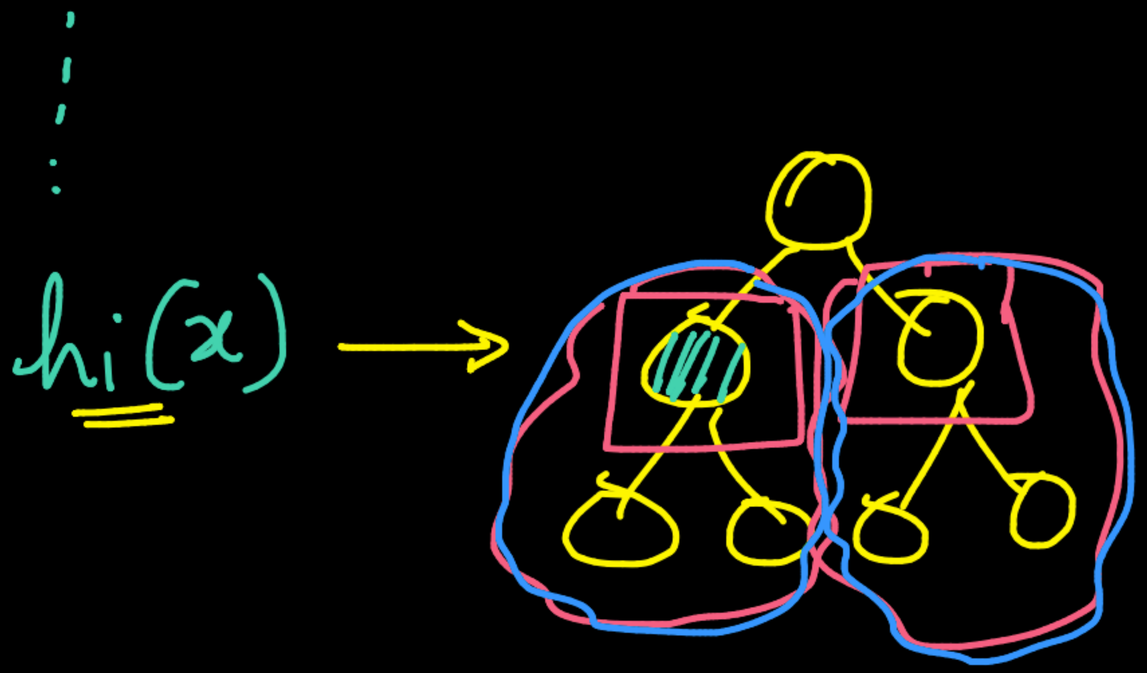


(let) $M = 100$

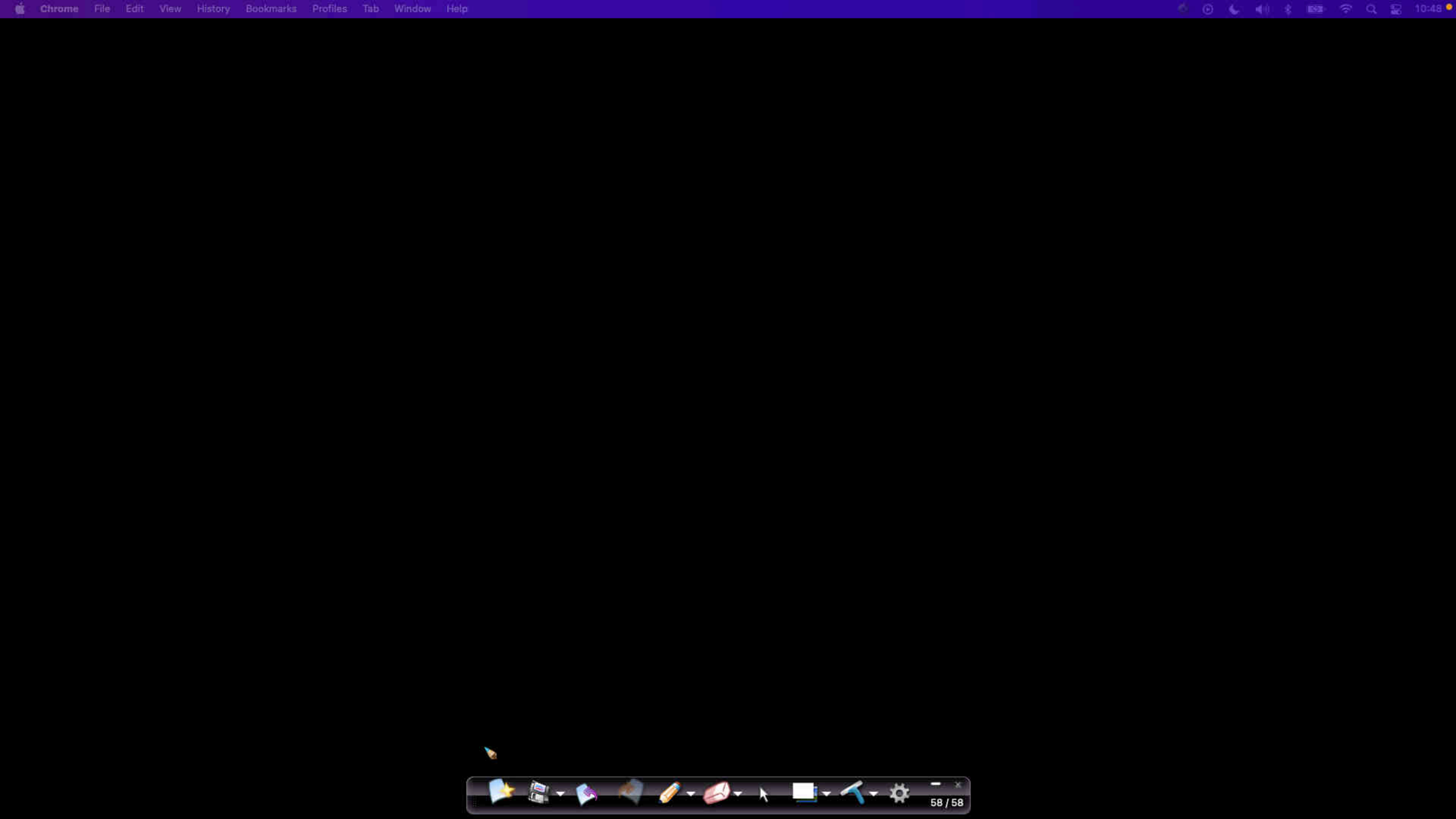
slow to train

Q how can we speed up GBDT

- row & column: small sampling : speed up



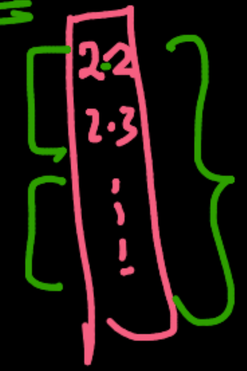
- individual tree level parallelization ✓
- Use for best split ✓


Chrome File Edit View History Bookmarks Profiles Tab Window Help
10:48
58 / 58

- official documentation
- top implementation → RF; GBDT ✓
- lots of code & algo optimization →
  - C++; Java; Scala; Python, R ...
- Multi core
- Spark / Dask ...

Xgboost: Millions of users

fi numerical



Histo

data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

Contents

XGBoost Documentation

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

Contents

61 / 61

Python API Reference

This page gives the Python API reference of xgboost, please also refer to Python Package Introduction for more information about the Python package.

Global Configuration

xgboost.config_context(***new_config*)

Context manager for global XGBoost configuration.

Global configuration consists of a collection of parameters that can be applied in the global scope. See Global Configuration for the full list of parameters supported in the global confi

- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.
- `num_feature` [set automatically by XGBoost, no need to be set by user]
  - Feature dimension used in boosting, set to maximum dimension of the feature

Parameters for Tree Booster

- `eta` [default=0.3, alias: `learning_rate`]
  - Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results

- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.
- `num_feature` [set automatically by XGBoost, no need to be set by user]
  - Feature dimension used in boosting, set to maximum dimension of the feature

Parameters for Tree Booster

- `eta` [default=0.3, alias: `learning_rate`]
  - Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results

Parameters for Tree Booster

- `eta` [default=0.3, alias: `learning_rate`]
  - Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger `min_child_weight` is, th

- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger `min_child_weight` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_delta_step` [default=0]
  - Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.
  - ra

- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger `min_child_weight` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_delta_step` [default=0]
  - Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.
  - ra

- range: [0,∞]

- `max_delta_step` [default=0]
  - Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.
  - range: [0,∞]
- `subsample` [default=1]
  - Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.
  - range: (0,1]
- `sampling_method` [default= `uniform`]
  - The method to use to sample the training instances.
  - `uniform` : each training instance has an equal probability of being selected. Typically set `subsample` >= 0.5 for good results.
  - `gradient_based` : the selection probability for each training instance is proportional to the *regularized absolute value* of gradients (more specifically, $\sqrt{g^2 + \lambda h^2}$). `subsample` may be set to as low as 0.1 without loss of model accuracy. Note that this sampling method is only supported when `tree_method` is set to `gpu_hist` ; other tree methods only support `uniform` sampling.
- `cols` ...]

- `colsample_bynode` is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
- `colsample_by*` parameters work cumulatively. For instance, the combination `{'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5}` with 64 features will leave 8 features to choose from at each split.

  Using the Python or the R package, one can set the `feature_weights` for DMatrix to define the probability of each feature being selected when using column sampling. There's a similar parameter for `fit` method in sklearn interface.

- `lambda` [default=1, alias: `reg_lambda`]
  - L2 regularization term on weights. Increasing this value will make model more conservative.
- `alpha` [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative.
- `tree_method` string [default= `auto`]
  - The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.
  - XGBoost supports `approx`, `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist`.
  - Choices: `auto`, `exact`, `approx`, `hist`, `gpu_hist`, this is a combination of commonly used updaters. For other updaters like `refresh`, set the parameter `updater` directly.
    - `auto`: Use heuristic to choose the fastest method

dmlc XGBoost
stable
Search docs
Installation Guide
Building From Source
Get Started with XGBoost
XGBoost Tutorials
Frequently Asked Questions
XGBoost User Forum
GPU Support
XGBoost Parameters
Prediction
Tree Methods
Python Package
R Package
JVM Package
Read the Docs v: stable

parameter for `fit` method in sklearn interface.

- `lambda` [default=1, alias: `reg_lambda`]
  - L2 regularization term on weights. Increasing this value will make model more conservative.
- `alpha` [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative.
- `tree_method` string [default= `auto`]
  - The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.
  - XGBoost supports `approx` , `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist` .
  - Choices: `auto` , `exact` , `approx` , `hist` , `gpu_hist` , this is a combination of commonly used updaters. For other updaters like `refresh` , set the parameter `updater` directly.
    - `auto` : Use heuristic to choose the fastest method.
      - For small dataset, exact greedy (`exact`) will be used.
      - For larger dataset, approximate algorithm (`approx`) will be chosen. It's recommended to try `hist` and `gpu_hist` for higher performance with large dataset. (`gpu_hist`)has support for `external memory` .
      - Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.
    - `exact` : Exact greedy algorithm. Enumerates all split candidates.
    - `approx` : Approximate greedy algorithm using quantile sketch and gradient histogram.
    - `hist` : Faster histogram optimized approximate greedy algorithm.

- `alpha` [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative.
- `tree_method` string [default= `auto`]
  - The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.
  - XGBoost supports `approx`, `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist`.
  - Choices: `auto`, `exact`, `approx`, `hist`, `gpu_hist`, this is a combination of commonly used updaters. For other updaters like `refresh`, set the parameter `updater` directly.
    - `auto`: Use heuristic to choose the fastest method.
      - For small dataset, exact greedy (`exact`) will be used.
      - For larger dataset, approximate algorithm (`approx`) will be chosen. It's recommended to try `hist` and `gpu_hist` for higher performance with large dataset. (`gpu_hist`)has support for `external memory`.
      - Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.
    - `exact`: Exact greedy algorithm. Enumerates all split candidates.
    - `approx`: Approximate greedy algorithm using quantile sketch and gradient histogram.
    - `hist`: Faster histogram optimized approximate greedy algorithm.
    - `gpu_hist`: GPU implementation of `hist` algorithm.
- `sketch_eps` [default=0.03]
  - Only used for `updater=grow_local_histmaker`.
  - T… ompared to directly select

71 / 71

General Parameters

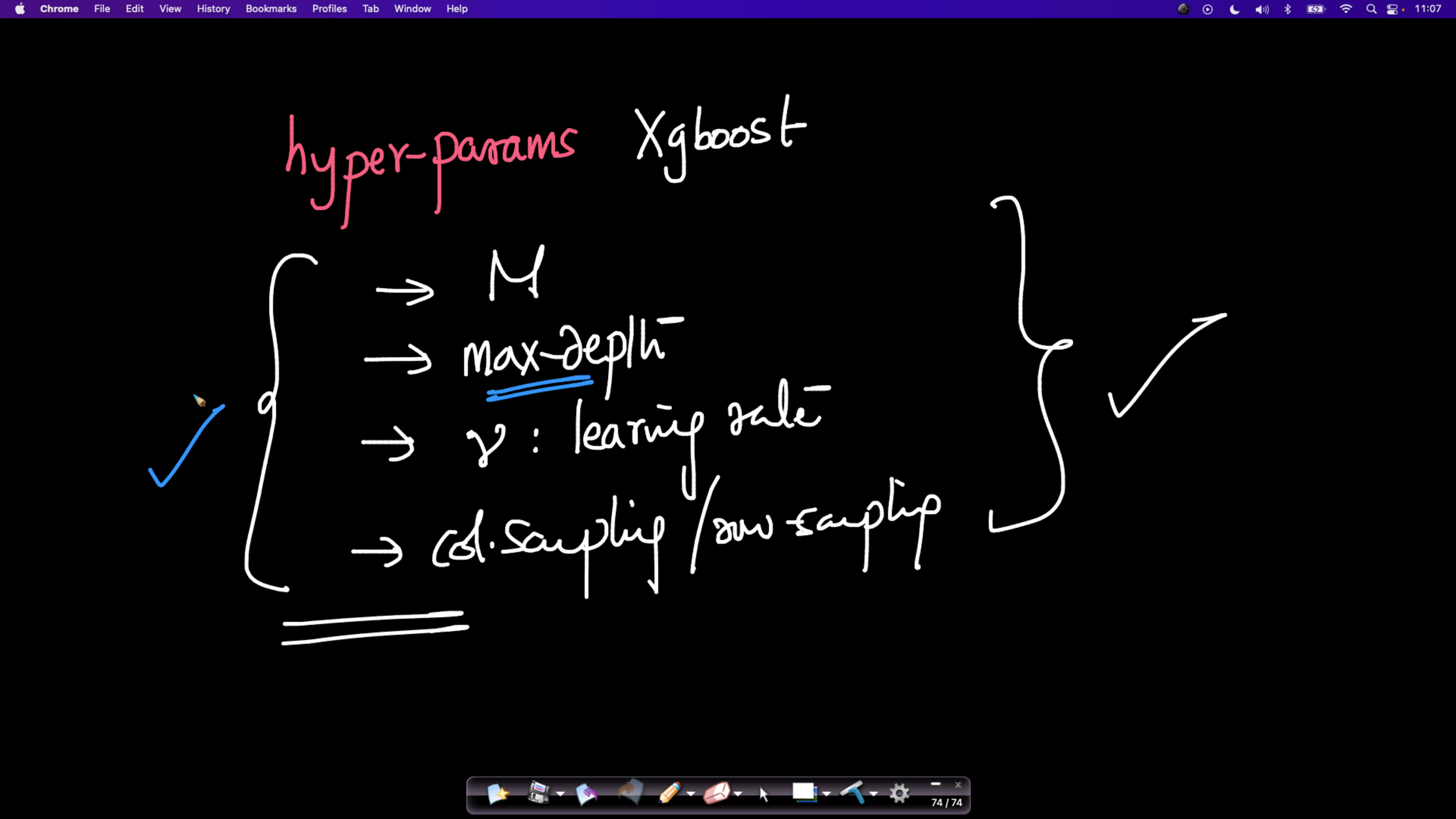
- `booster` [default= `gbtree`]
  - Which booster to use. Can be `gbtree` , `gblinear` or `dart` ; `gbtree` and `dart` use tree based models while `gblinear` uses linear functions.
- `verbosity` [default=1]
  - Verbosity of printing messages. Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug). Sometimes XGBoost tries to change configurations based on heuristics, which is displayed as warning message. If there's unexpected behaviour, please try to increase value of verbosity.
- `validate_parameters` [default to `false` , except for Python, R and CLI interface]
  - When set to True, XGBoost will perform validation of input parameters to check whether a parameter is used or not. The feature is still experimental. It's expected to have some false positives.
- `nthread` [default to maximum number of threads available if not set]
  - Number of parallel threads used to run XGBoost. When choosing it, please keep thread contention and hyperthreading in mind.
- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.
- `num_feature` [set automatically by XGBoost, no need to be set by user]
  - Feature dimension used in boosting, set to maximum dimension of the feature

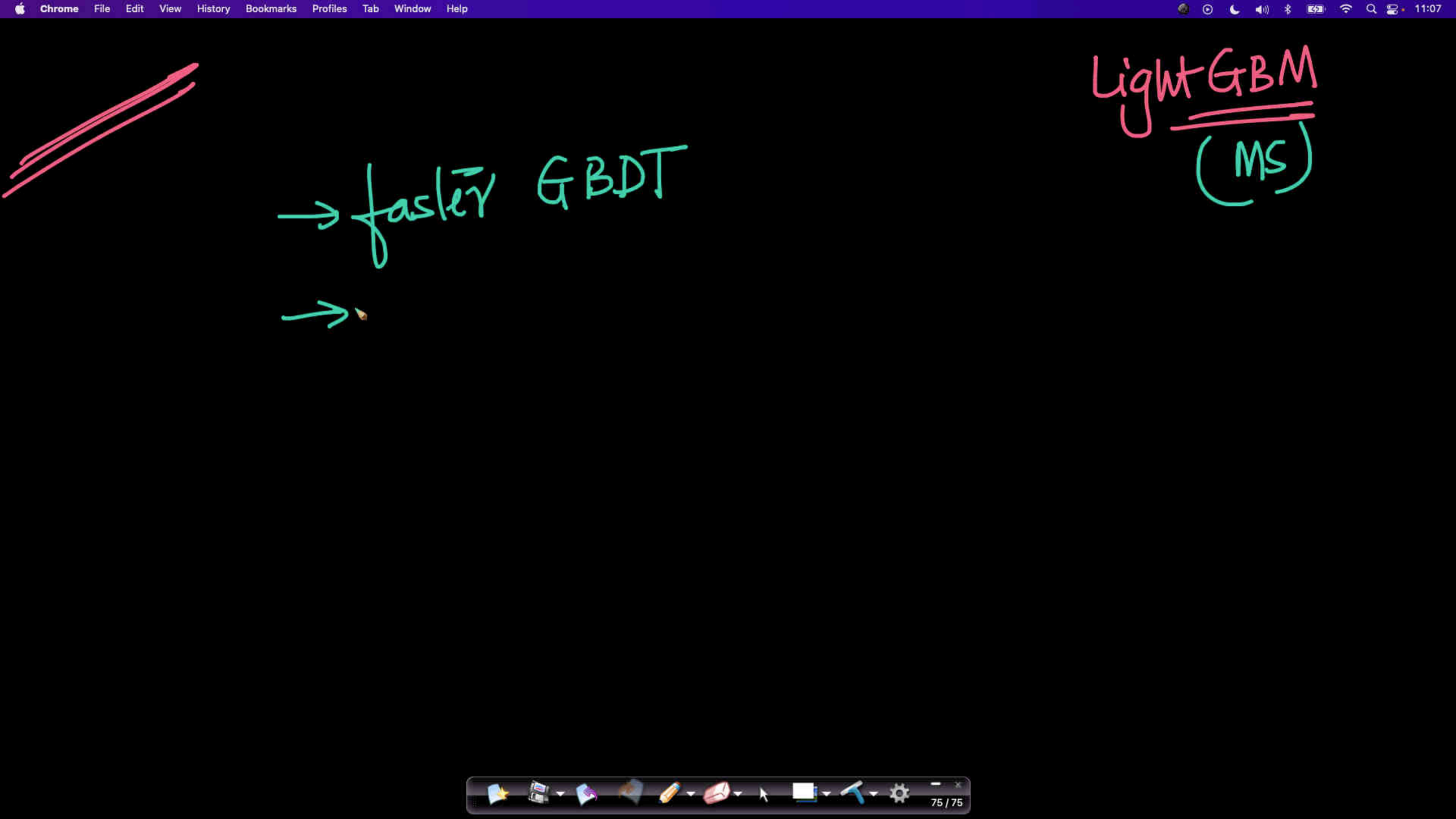
Parameters for Tree Booster

General Parameters

- `booster` [default= `gbtree`]
  - Which booster to use. Can be `gbtree` , `gblinear` or `dart` ; `gbtree` and `dart` use tree based models while `gblinear` uses linear functions.
- `verbosity` [default=1]
  - Verbosity of printing messages. Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug). Sometimes XGBoost tries to change configurations based on heuristics, which is displayed as warning message. If there's unexpected behaviour, please try to increase value of verbosity.
- `validate_parameters` [default to `false` , except for Python, R and CLI interface]
  - When set to True, XGBoost will perform validation of input parameters to check whether a parameter is used or not. The feature is still experimental. It's expected to have some false positives.
- `nthread` [default to maximum number of threads available if not set]
  - Number of parallel threads used to run XGBoost. When choosing it, please keep thread contention and hyperthreading in mind.
- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.
- `num_feature` [set automatically by XGBoost, no need to be set by user]
  - Feature dimension used in boosting, set to maximum dimension of the feature

Parameters for Tree Booster


hyper-params Xgboost
M
max-depth
γ : learning rate
col.sampling / row sampling


Chrome File Edit View History Bookmarks Profiles Tab Window Help
11:07
LightGBM
(MS)
→ faster GBDT
→
75 / 75

Scikit-Learn API

Scikit-Learn Wrapper interface for XGBoost.

class **xgboost.XGBRegressor**(**, objective='reg:squarederror', **kwargs*)

Bases: `xgboost.sklearn.XGBModel`, `sklearn.base.RegressorMixin`

Implementation of the scikit-learn API for XGBoost regression.

Parameters:

- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds.
- **max_depth** (*Optional[int]*) – Maximum tree depth for base learners.
- **max_leaves** – Maximum number of leaves; 0 indicates no limit.
- **max_bin** – If using histogram-based algorithm, maximum number of bins per feature
- **grow_policy** – Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow depth-wise. 1: favor splitting at nodes with highest loss change.
- **learning_rate** (*Optional[float]*) – Boosting learning rate (xgb's "eta")
- **verbosity** (*Optional[int]*) – The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
- **objective** (*Union[str, Callable[[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], NoneType]*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

to use: gbtree, gblinear or

Standalone Random Forest With XGBoost API

The following parameters must be set to enable random forest training.

- `booster` should be set to `gbtree` as we are training forests. Note that as this is the default, this parameter needn't be set explicitly.
- `subsample` must be set to a value less than 1 to enable random selection of training cases (rows).
- One of `colsample_by*` parameters must be set to a value less than 1 to enable random selection of columns. Normally, `colsample_bynode` would be set to a value less than 1 to randomly sample columns at each tree split.
- `num_parallel_tree` should be set to the size of the forest being trained.
- `num_boost_round` should be set to 1 to prevent XGBoost from boosting multiple random forests. Note that this is a keyword argument to `train()`, and is not part of the parameter dictionary.
- `eta` (alias: `learning_rate`) must be set to 1 when training random forest regression.
- `random_state` can be used to seed the random number generator.

Other parameters should be set in a similar way they are set for gradient boosting. For instance, `objective` will typically be `reg:squarederror` for regression and `binary:logistic` for classification, `lambda` should be set according to a desired regularization weight, etc.

If both `num_parallel_tree` and `num_boost_round` are greater than 1, training will use a combination of random forest and gradient boosting strategy. It will perform `num_boost_round` rounds, boosting a random[illegible] ping is not enabled, the final model w[illegible]

77 / 78

- One of `colsample_by*` parameters must be set to a value less than 1 to enable random selection of columns. Normally, `colsample_bynode` would be set to a value less than 1 to randomly sample columns at each tree split.
- `num_parallel_tree` should be set to the size of the forest being trained.
- `num_boost_round` should be set to 1 to prevent XGBoost from boosting multiple random forests. Note that this is a keyword argument to `train()`, and is not part of the parameter dictionary.
- `eta` (alias: `learning_rate`) must be set to 1 when training random forest regression.
- `random_state` can be used to seed the random number generator.

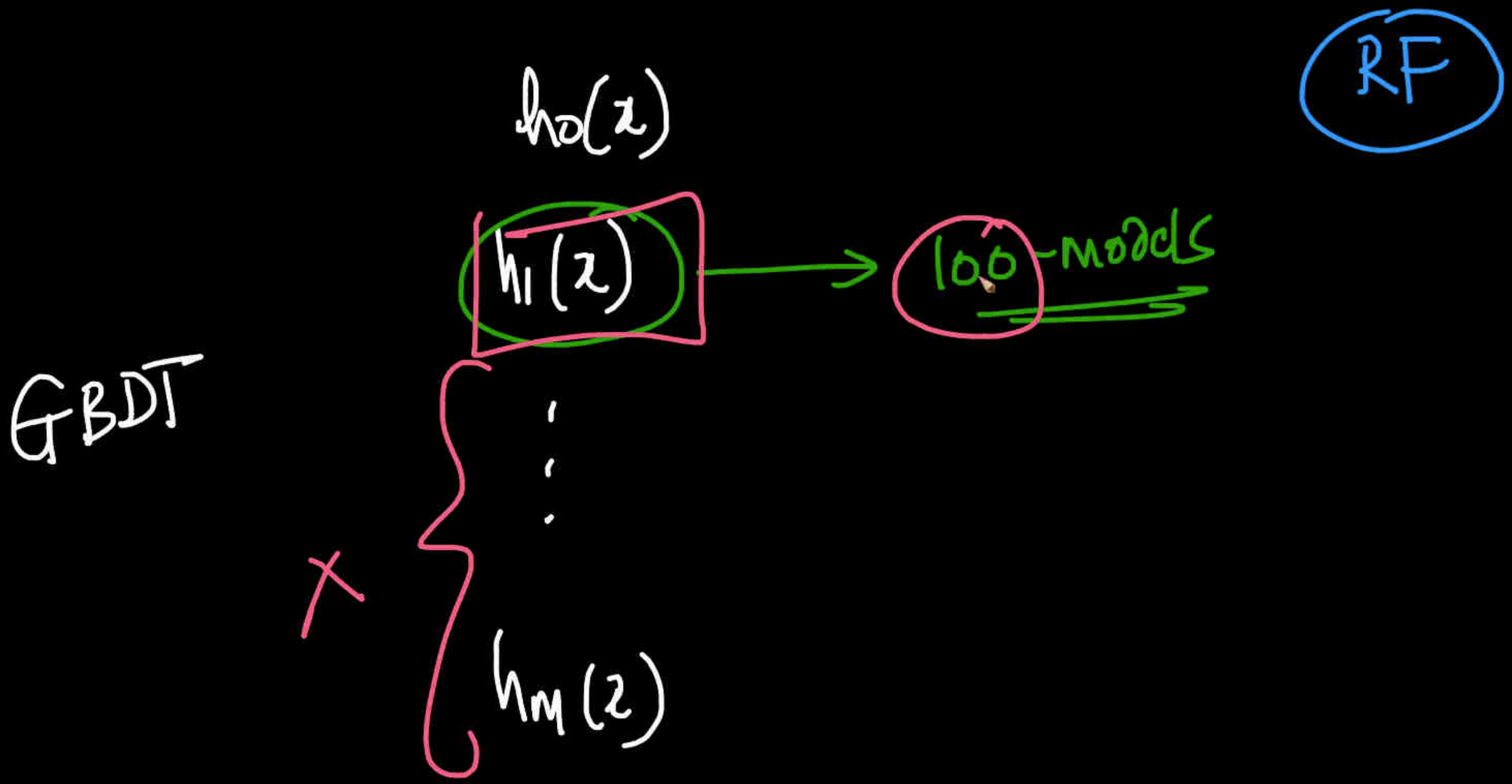
Other parameters should be set in a similar way they are set for gradient boosting. For instance, `objective` will typically be `reg:squarederror` for regression and `binary:logistic` for classification, `lambda` should be set according to a desired regularization weight, etc.

If both `num_parallel_tree` and `num_boost_round` are greater than 1, training will use a combination of random forest and gradient boosting strategy. It will perform `num_boost_round` rounds, boosting a random forest of `num_parallel_tree` trees at each round. If early stopping is not enabled, the final model will consist of `num_parallel_tree` * `num_boost_round` trees.

Here is a sample parameter dictionary for training a random forest on a GPU using xgboost:

```
params = {
  'colsample_bynode': 0.8,
  'learning_rate': 1,
  'max_depth': 5,
  'num_parallel_tree': 100,
  'objective': 'binary:logistic',
  'sub
  'tre
```

78 / 79



`lambda` should be set according to a desired regularization weight, etc.

If both `num_parallel_tree` and `num_boost_round` are greater than 1, training will use a combination of random forest and gradient boosting strategy. It will perform `num_boost_round` rounds, boosting a random forest of `num_parallel_tree` trees at each round. If early stopping is not enabled, the final model will consist of `num_parallel_tree` * `num_boost_round` trees.

Here is a sample parameter dictionary for training a random forest on a GPU using xgboost:

```
params = {
  'colsample_bynode': 0.8,
  'learning_rate': 1,
  'max_depth': 5,
  'num_parallel_tree': 100,
  'objective': 'binary:logistic',
  'subsample': 0.8
  'tree_method': 'gpu_hist'
}
```

RF with 100 learn

A random forest model can then be trained as follows:

```
bst = train(params, dmatrix, num_boost_round=1
```

Standalone Random Forest With Scikit-Learn-Like API

`XGBRFCl` ... random forest functionality. They are basically versions of ... that train random forest instead of

80 / 80

`lambda` should be set according to a desired regularization weight, etc.

If both `num_parallel_tree` and `num_boost_round` are greater than 1, training will use a combination of random forest and gradient boosting strategy. It will perform `num_boost_round` rounds, boosting a random forest of `num_parallel_tree` trees at each round. If early stopping is not enabled, the final model will consist of `num_parallel_tree` * `num_boost_round` trees.

Here is a sample parameter dictionary for training a random forest on a GPU using xgboost:

```
params = {
  'colsample_bynode': 0.8,
  'learning_rate': 1,
  'max_depth': 5,
  'num_parallel_tree': 100,
  'objective': 'binary:logistic',
  'subsample': 0.8,
  'tree_method': 'gpu_hist'
}
```

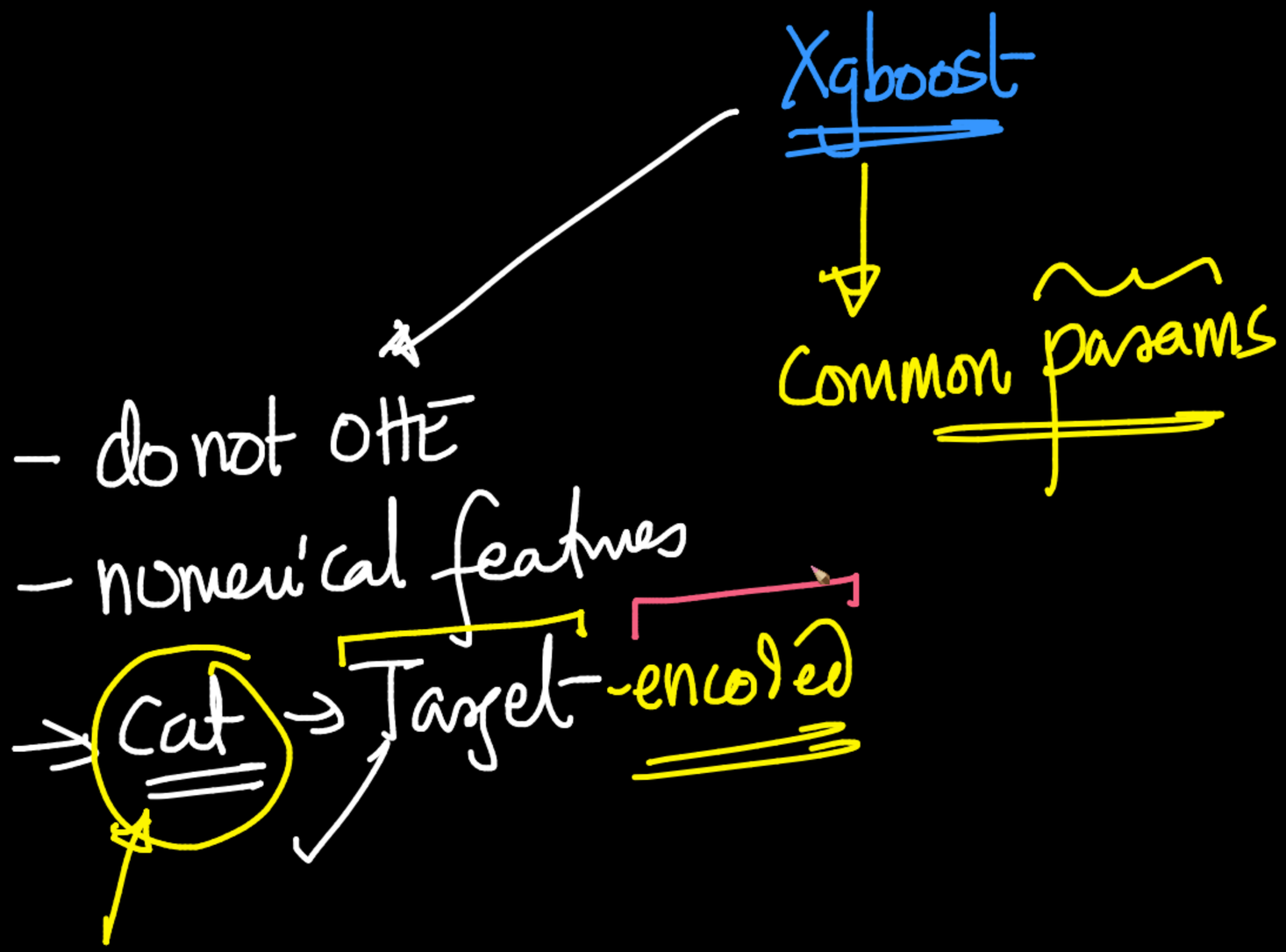
A random forest model can then be trained as follows:

```
bst = train(params, dmatrix, num_boost_round=1)
```

Standalone Random Forest With Scikit-Learn-Like API

`XGBRFCl` ... random forest functionality. They are basically versions of ... that train random forest instead of

81 / 81



LightGBM (2017)

→ faster than XgBoost

→ GOSS: Gradient based one side sampling

m^{th} model: $\{(x_i, r_{im})\}_{i=1}^{n}$

↳ drop where it is small

↳ sampling ✓

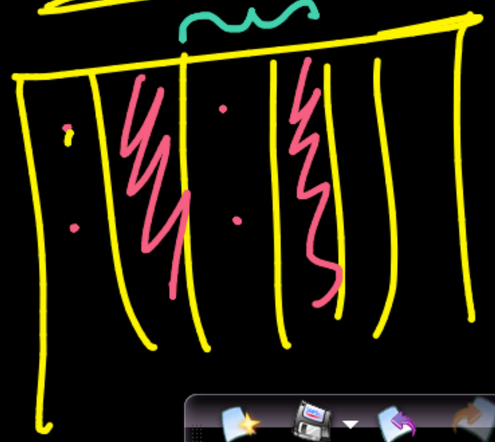
#datapoints ↓

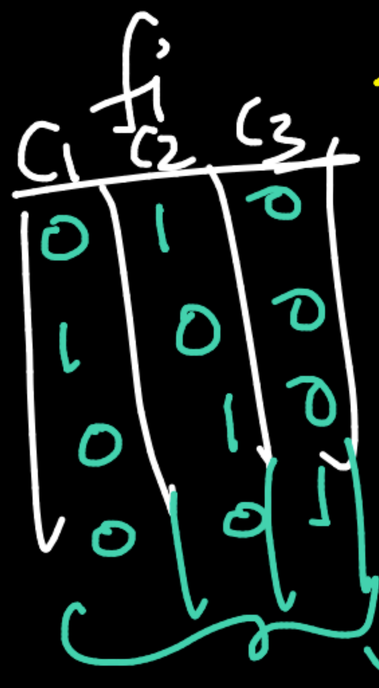
→ Exclusive feature Bundling: → reduce #features

graph-based

Categorical-feature :- OHE

d-dim →





group

Target encoding

GBDT > AdaBoost

[flexibility of choosing L]

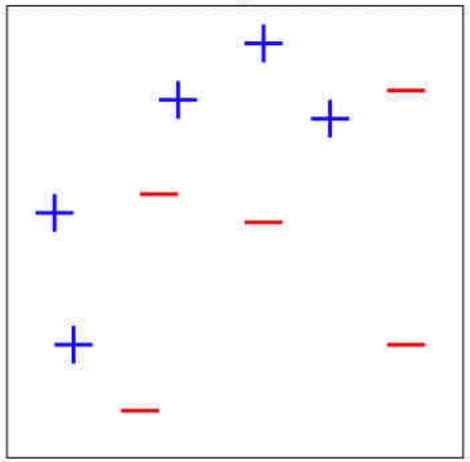
adaptive

Toy Example

"Adaboost"

: base learner

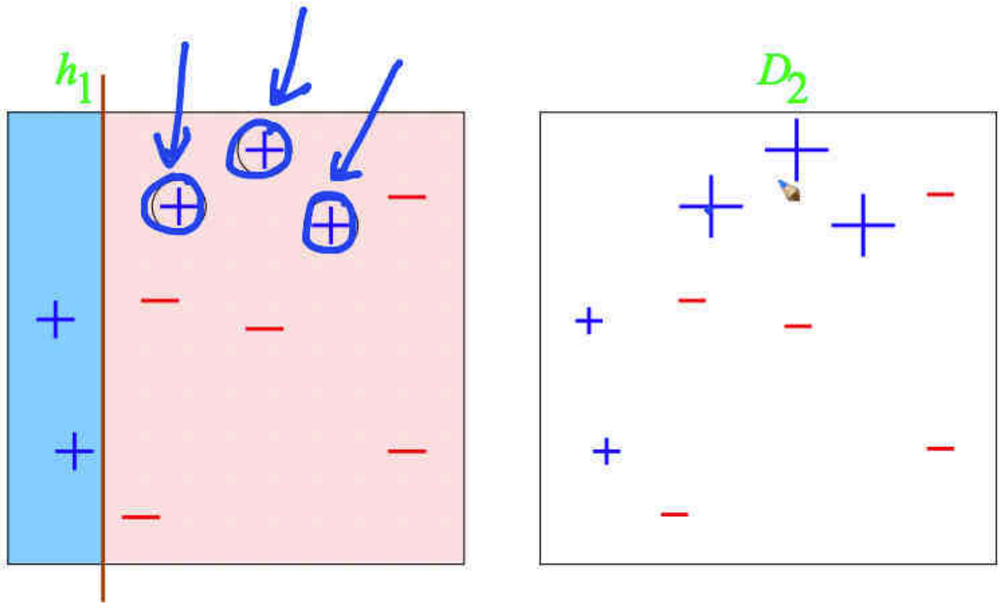
D_1



weak classifier

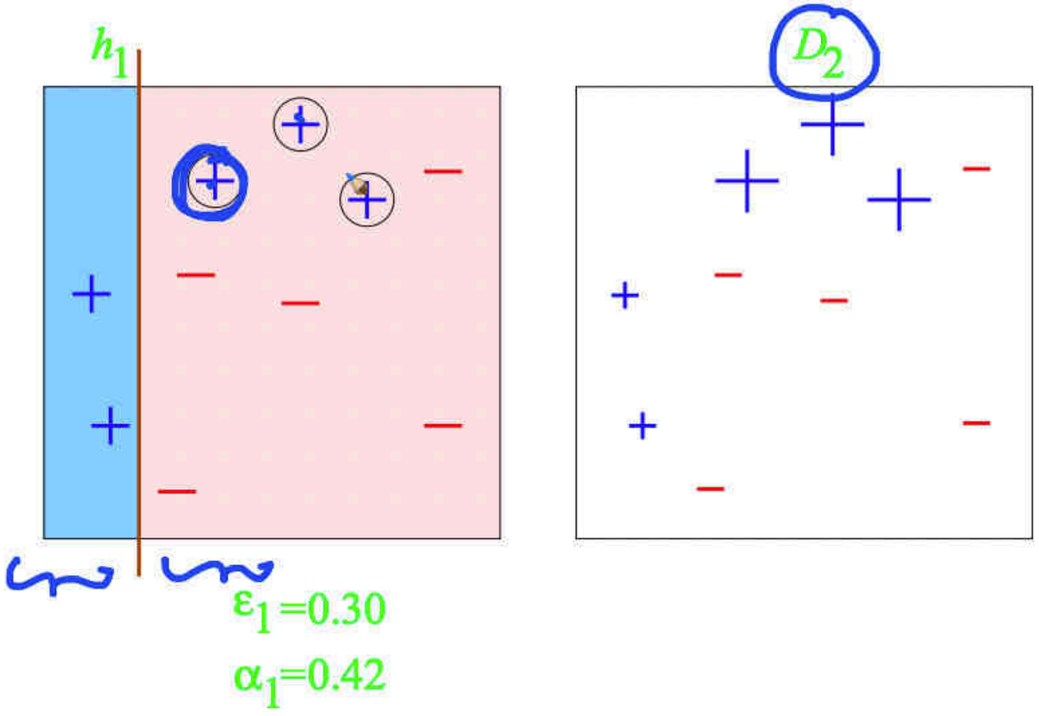
Round 1

$h_0(x) + h_1(x)$

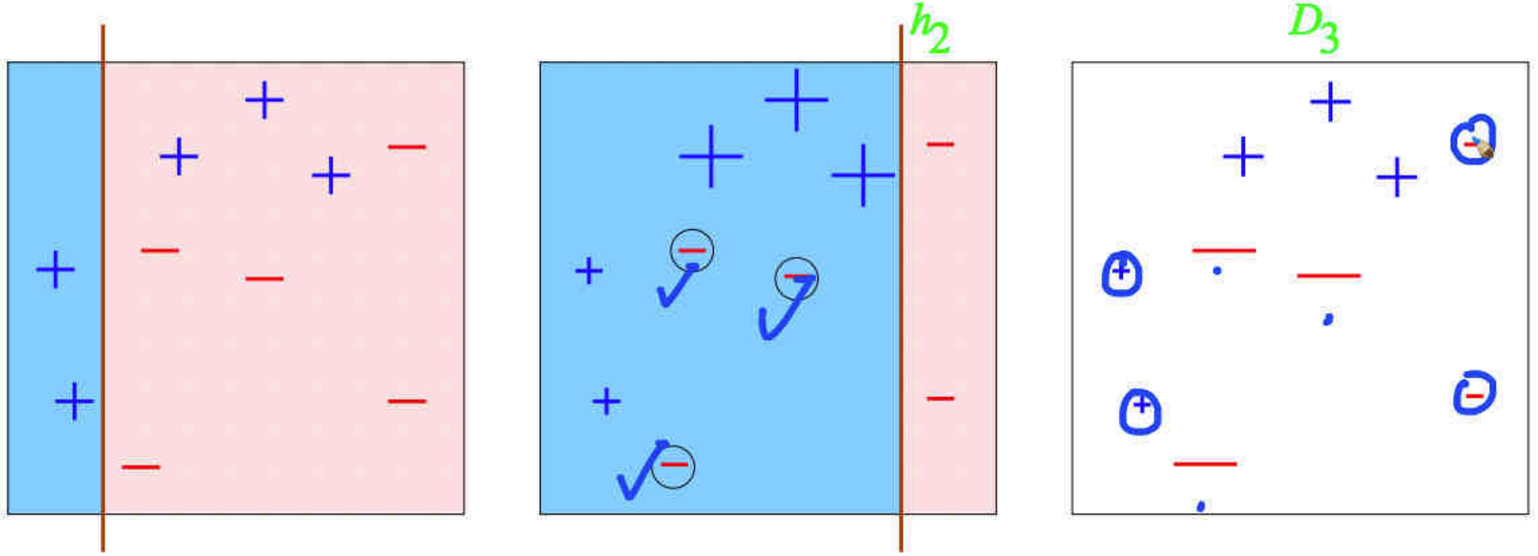


$\varepsilon_1 = 0.30$

$\alpha_1 = 0.42$



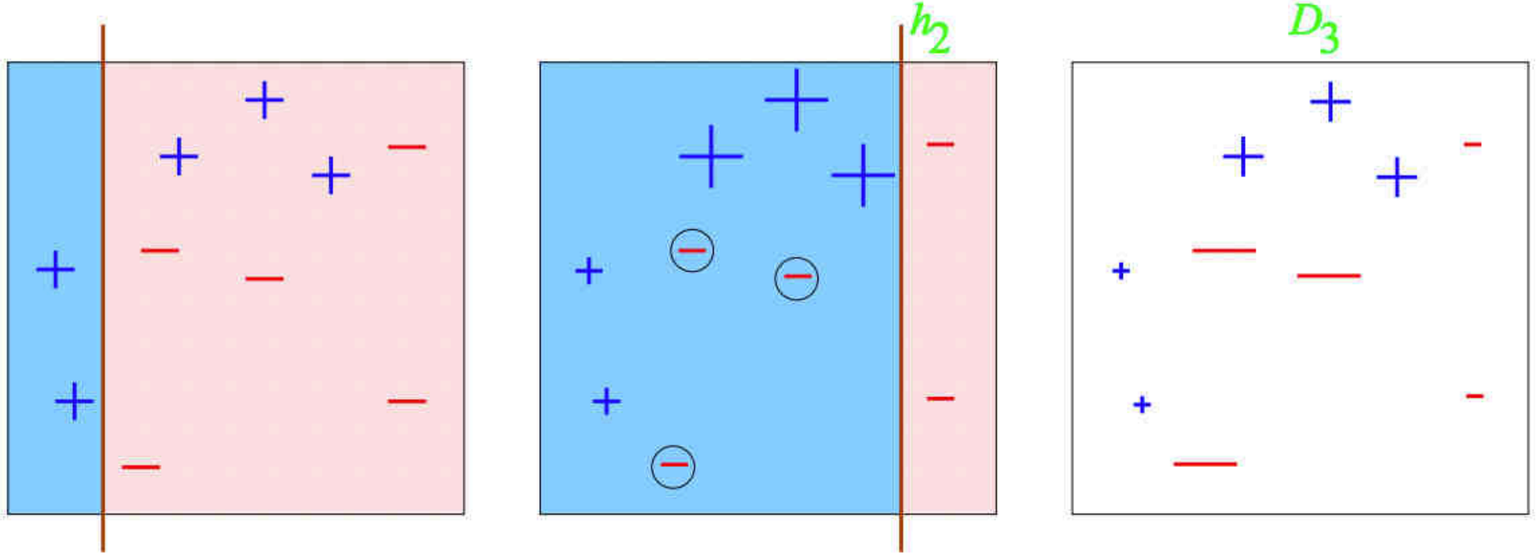
Round 2



$\varepsilon_2=0.21$

$\alpha_2=0.65$

Round 2

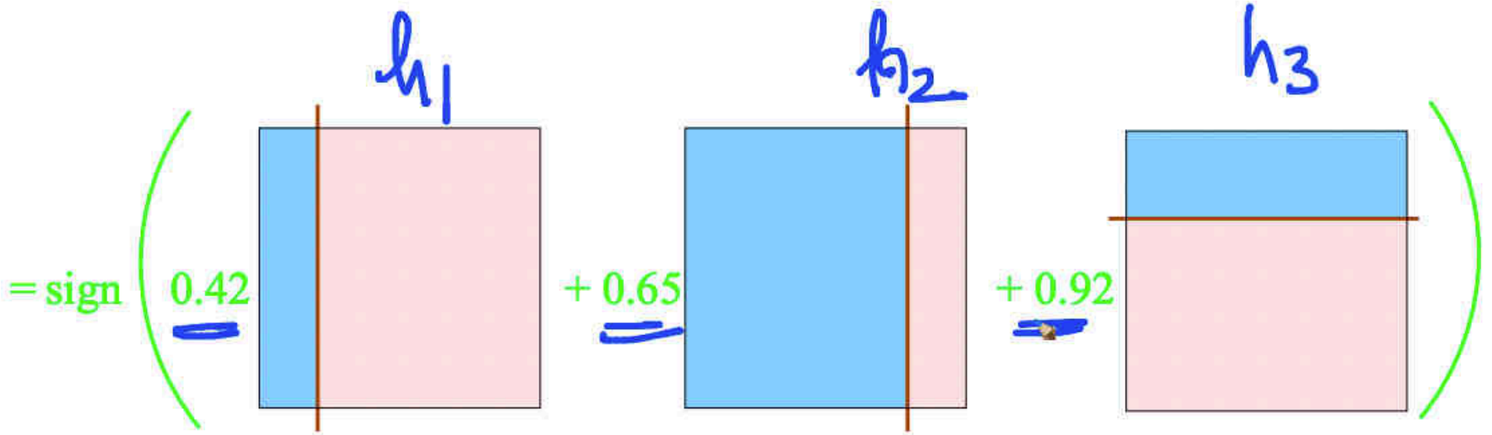


$\varepsilon_2 = 0.21$

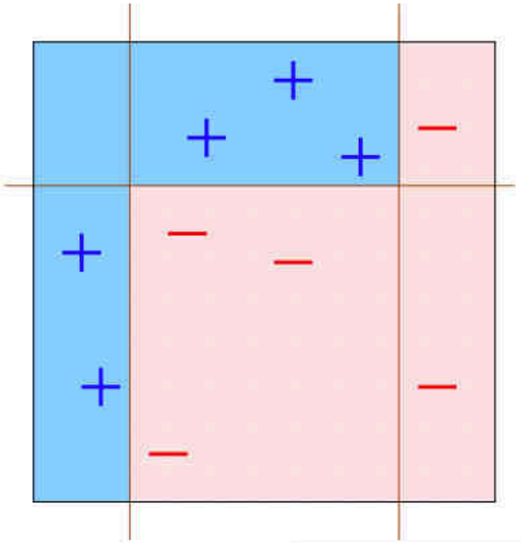
$\alpha_2 = 0.65$

Final Classifier

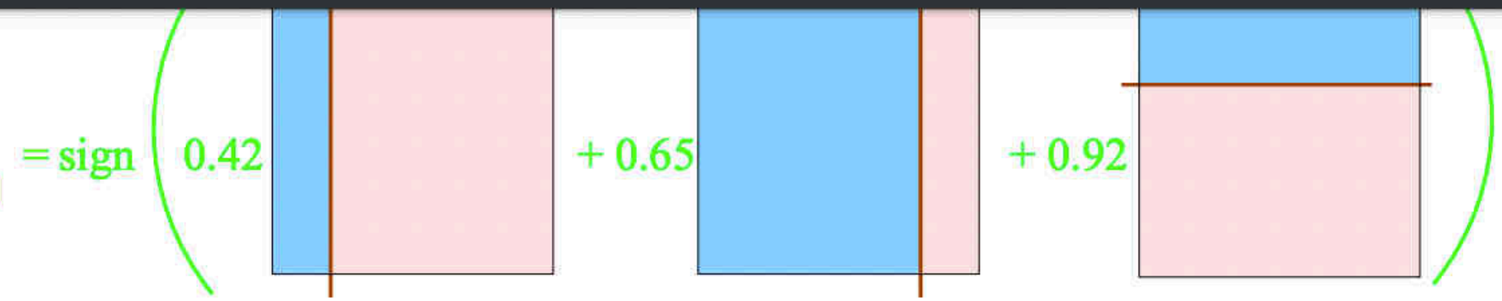
$$H_{\text{final}} = \text{sign}\left(0.42\, h_1 + 0.65\, h_2 + 0.92\, h_3\right)$$



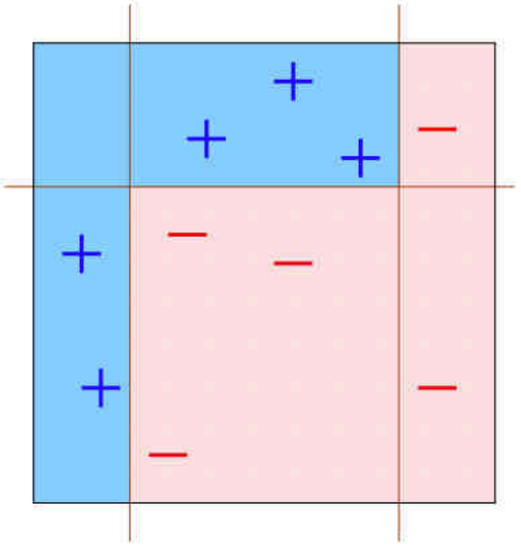
=



$$H_{\text{final}} = \text{sign}\left(0.42 + 0.65 + 0.92\right)$$

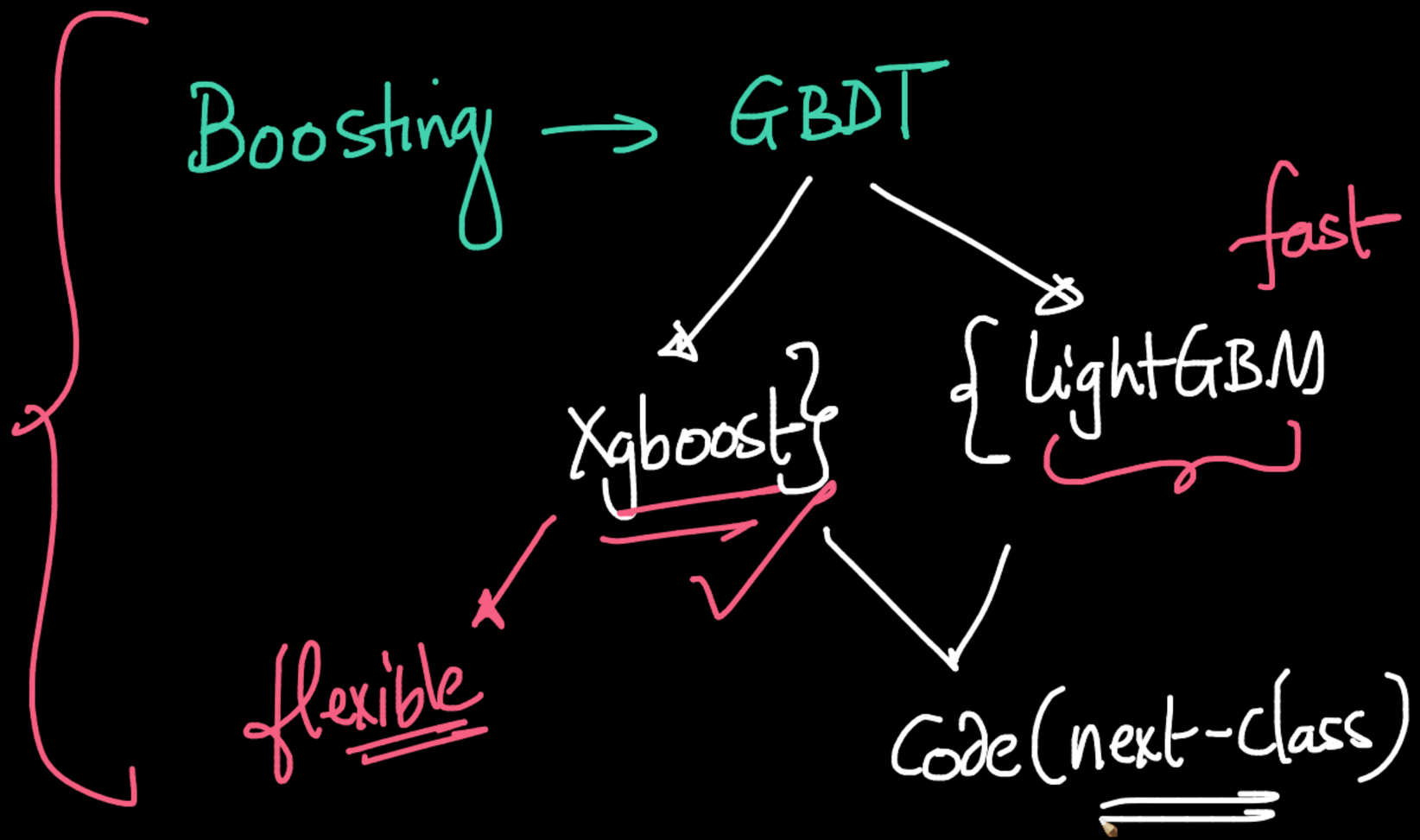


$=$

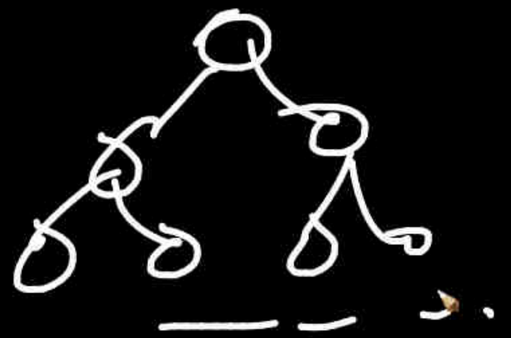


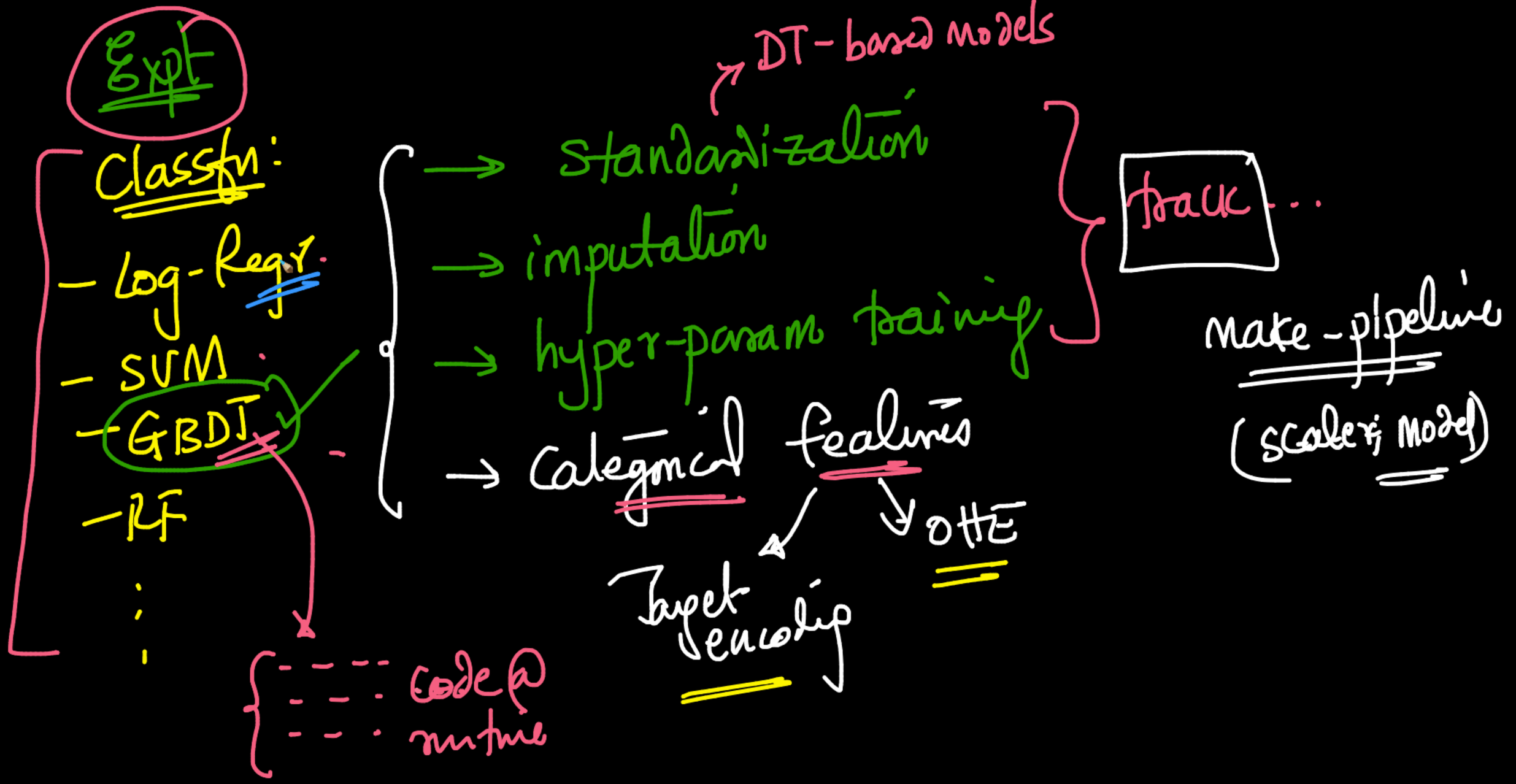
2009

Analyzing the

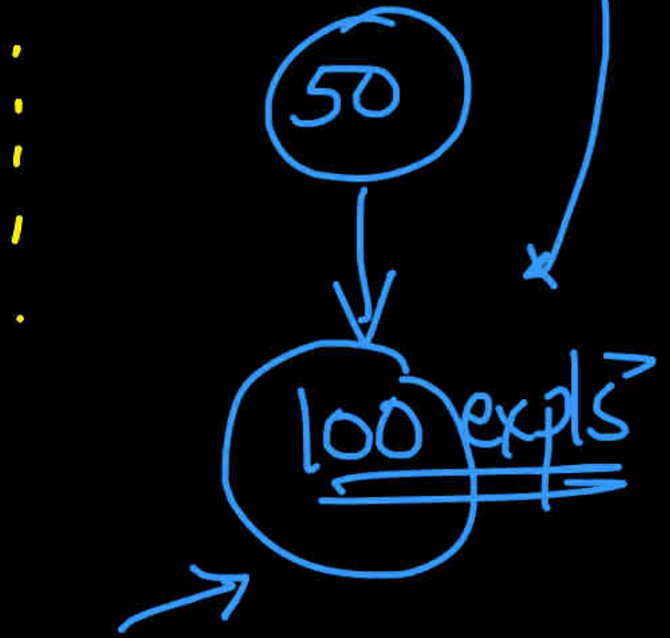

Boosting → GBDT
fast
Xgboost
LightGBM
flexible
Code (next-class)

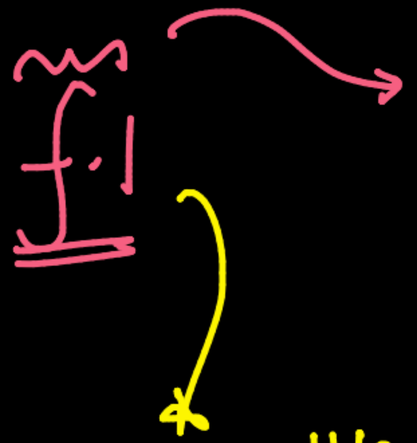
See the trees




Expt
Classfn:
- Log-Regr
- SVM
- GBDT
- RF
DT-based models
Standardization
imputation
hyper-param training
Categorical features
OHE
Target encoding
Code & runtime
track...
make-pipeline
(scaler; model)

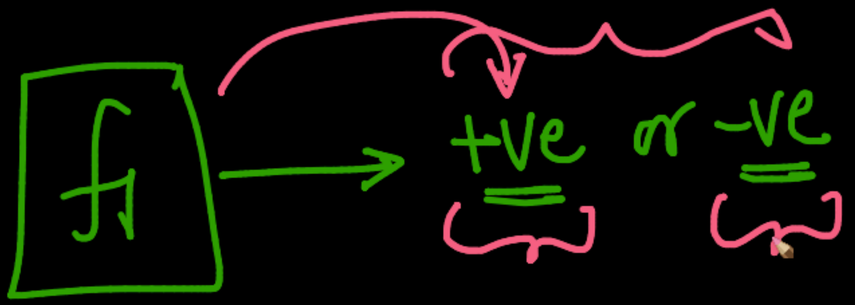
Git + internal-wiki





+ve → 1 Cancer

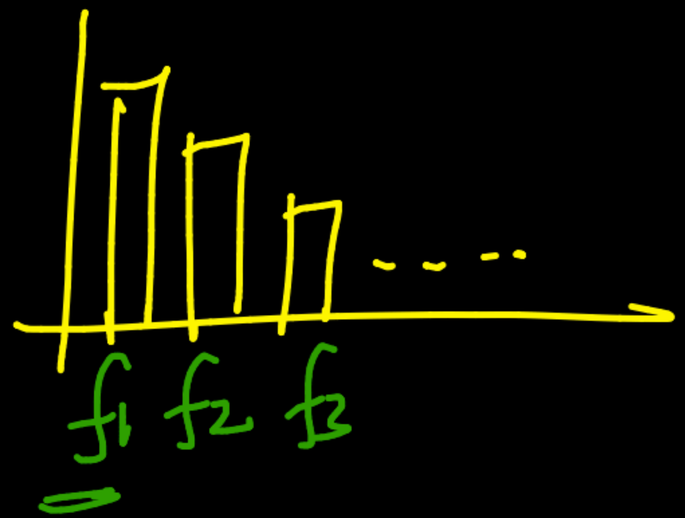
-ve → 0 not-cancer



{ IsSmoking }

+ve weight

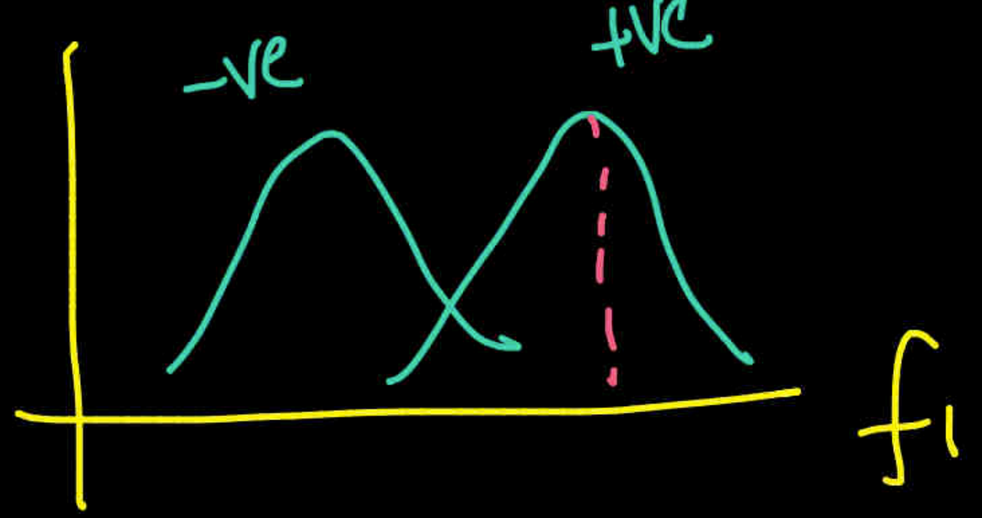
+ve-values

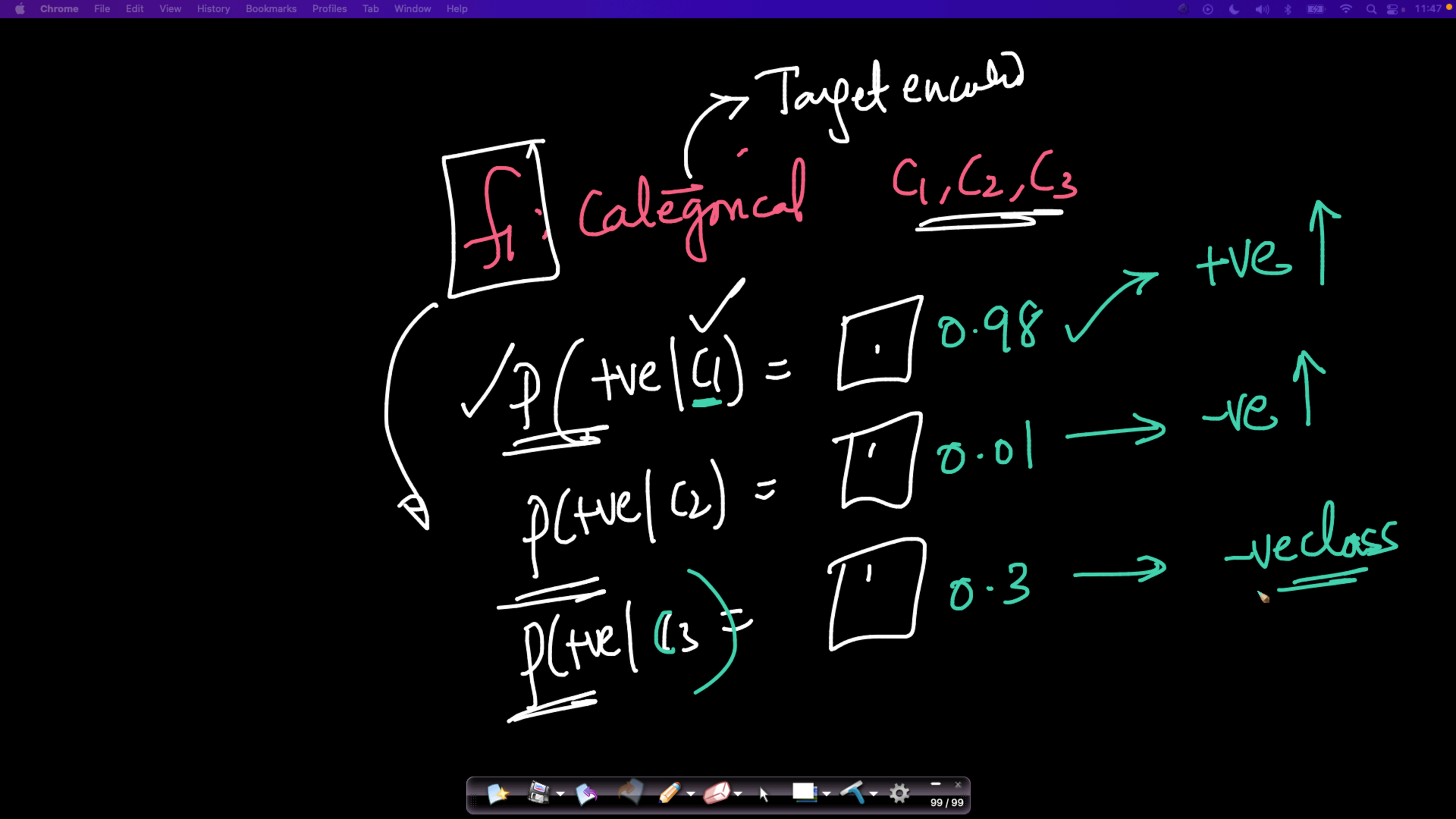


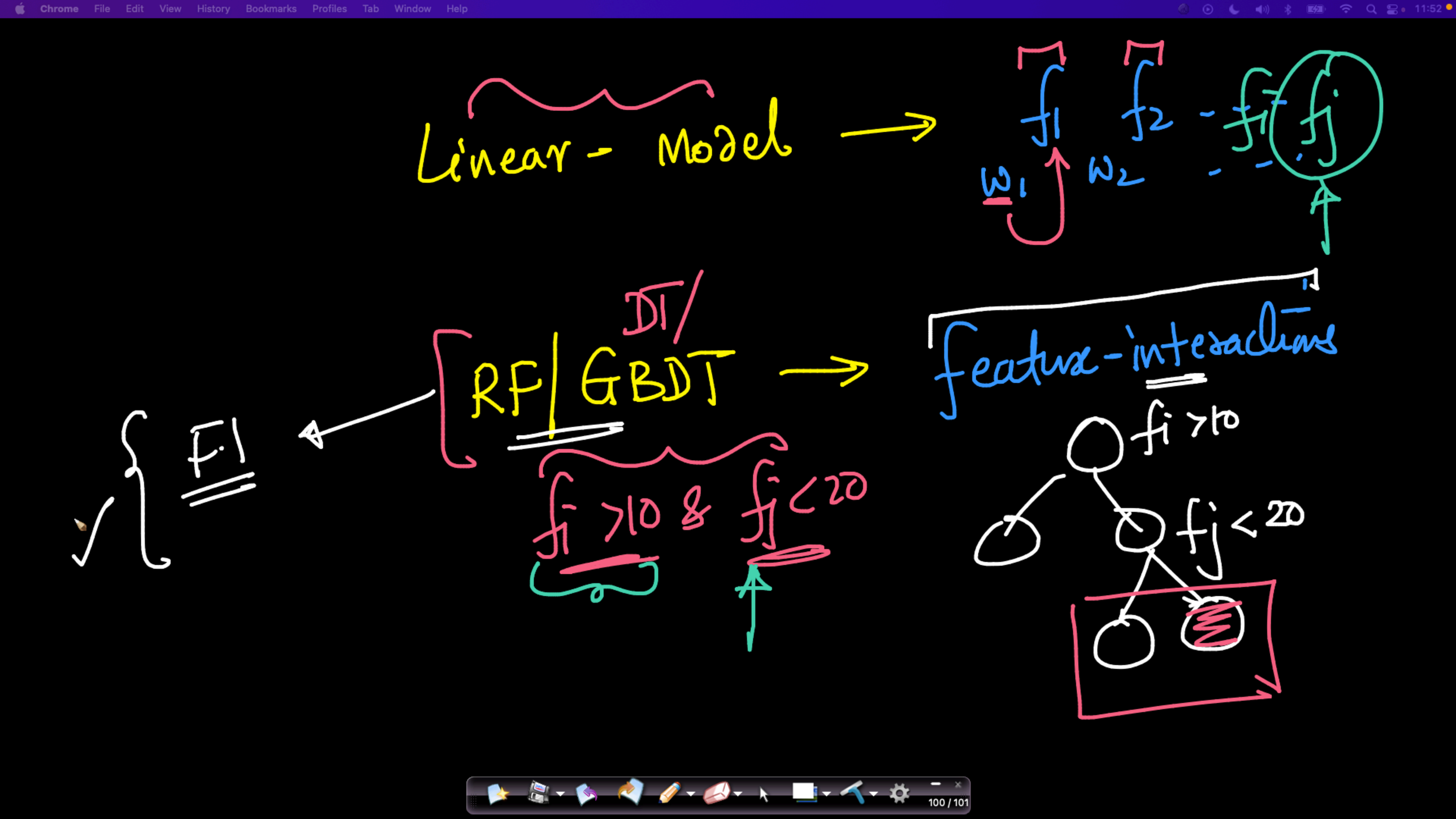
$f_i \uparrow$

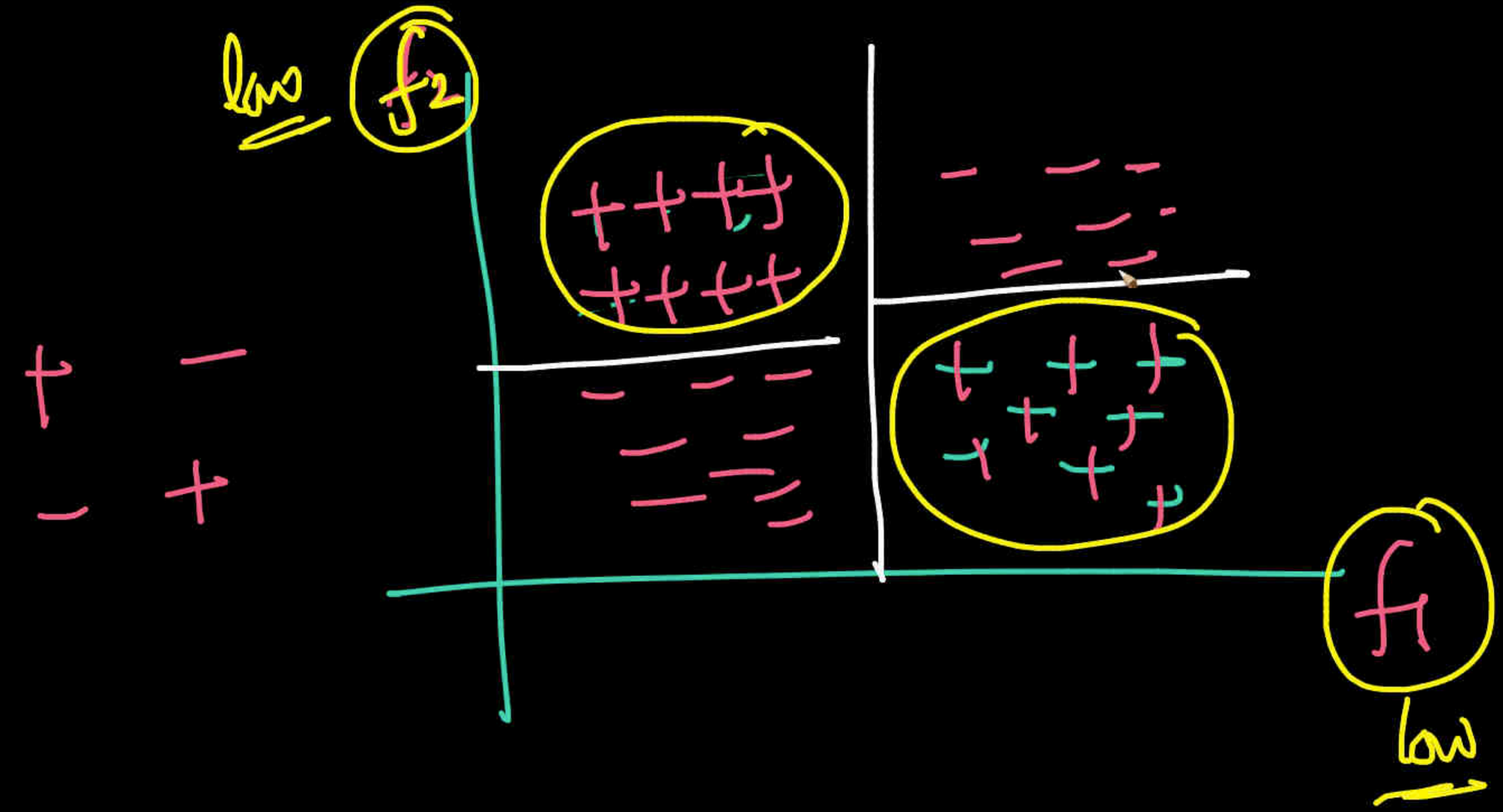
numerical

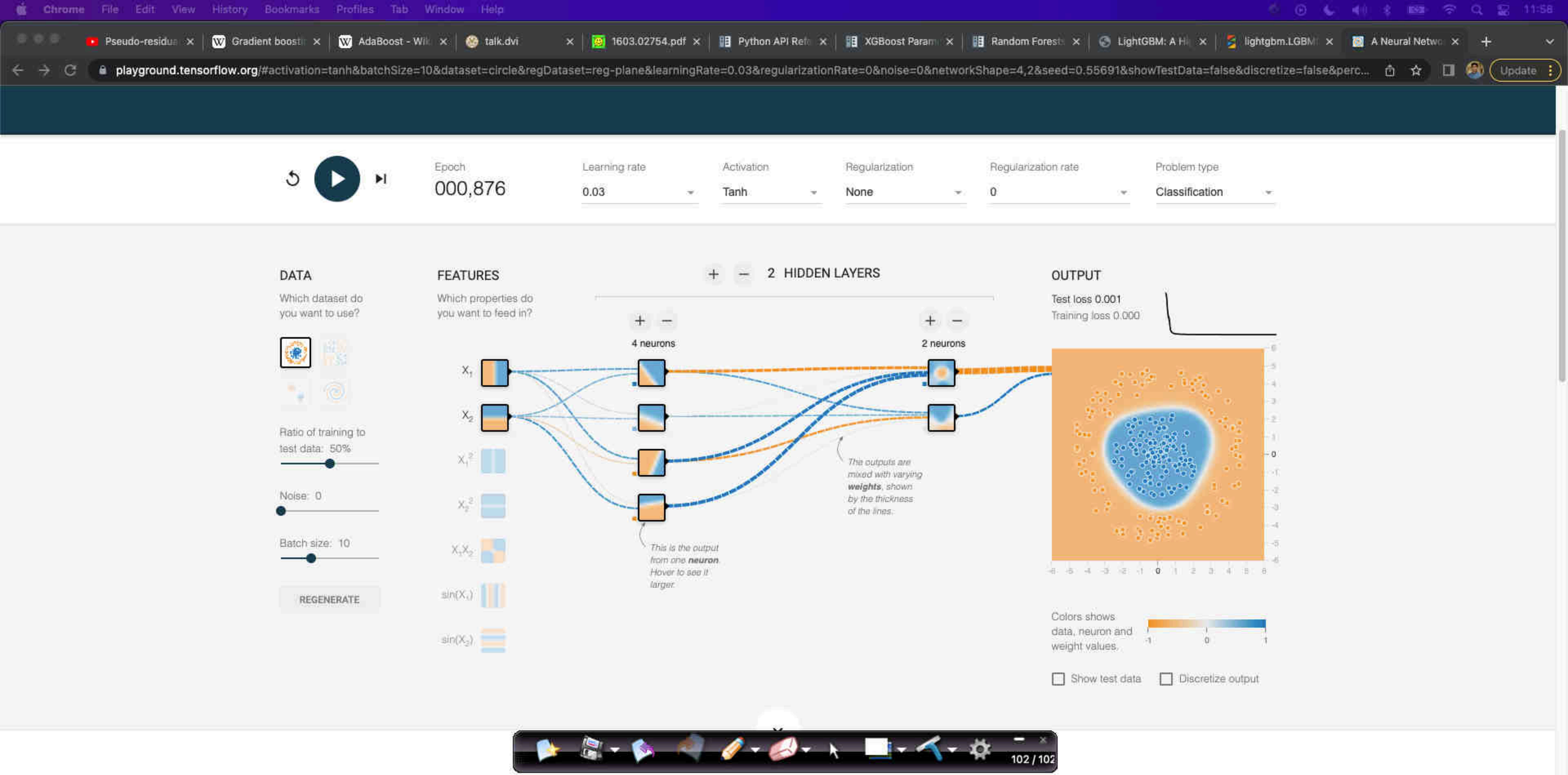
+ve or -ve




Target encoding
f1 : Categorical
C1, C2, C3
P(+ve | C1) = 0.98
+ve
P(+ve | C2) = 0.01
-ve
P(+ve | C3) = 0.3
-ve class
99 / 99


Linear - Model
f1
f2
fi
fj
w1
w2
DT/
RF/ GBDT
Feature-intesactions
F1
fi >10 & fj<20
fi >10
fj<20


low
f2
+ −
− +
f1
low



Um, What Is a Neural Network?

It's a technique for building a computer program that learns from data. It is based very loosely on how we think the human brain works. First, a