Boosting – Gradient Descent

Zhe Yang

Contents

- Gradient Boosting Machine (GBM)
- Regularization methods
- XGB & LightGBM frameworks

Gradient Boosting Machine (GBM)

Parameter space, function space, and margin space

Predictive learning (supervised)

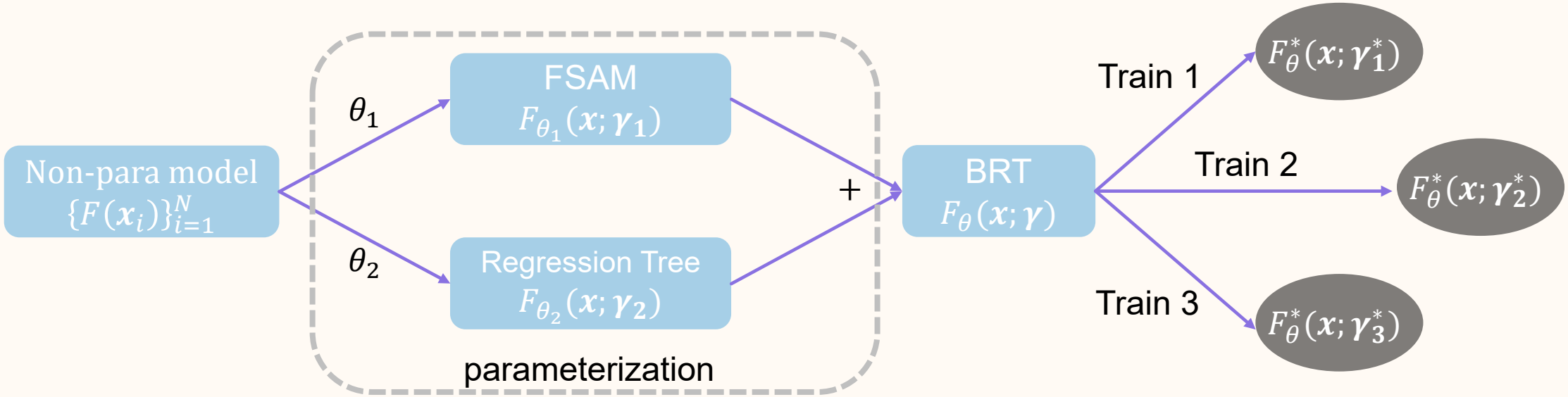
- The data
  - $\boldsymbol{x} = \{x_1, x_2, \cdots, x_p\}, y$ → one "training" sample is then $\{y_i, \boldsymbol{x_i}\}$
  - "training" sample set $\{y_i, \boldsymbol{x_i}\}_1^N$

- The goal
  - Obtain an estimate/approximation $\hat{F}$ of the truth function F^* mapping $\boldsymbol{x}$ to y
  - The $\hat{F}$ minimize some specific loss function $L(y, F(\boldsymbol{x}))$ over the joint distribution of all $(y, \boldsymbol{x})$ values.

$$F^* = \arg\min_F E_{y,\boldsymbol{x}}[L(y, F(\boldsymbol{x}))] = \arg\min_F E_{\boldsymbol{x}}\left[E_y\left(L\left(y, F(\boldsymbol{x})\right)\right)|\boldsymbol{x}\right]$$

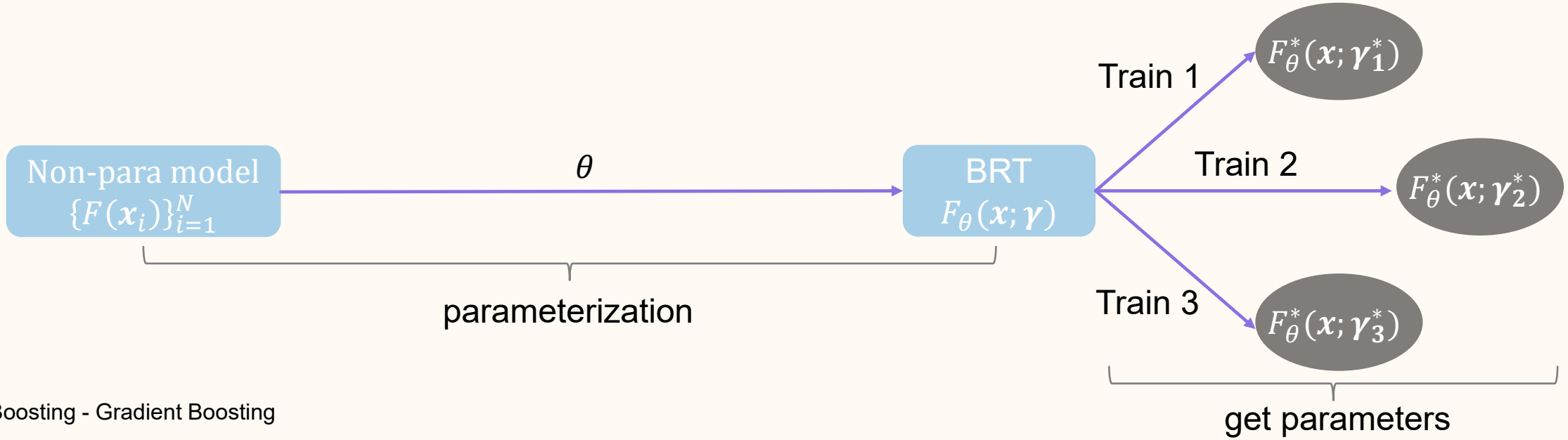
Select the parameterized model “class”

- In order to get the parameterized function model “object” that can be used to calculate the concrete predictions, we need to restrict $\hat{F}$ to be of a member of a parameterized model “class” of functions $F_\theta(\boldsymbol{x};\boldsymbol{\gamma}) \rightarrow \hat{F}$
  - here θ is the hyperparameter, and $\boldsymbol{\gamma}$ is the parameter
- For a Boosted Regression Tree (BRT) model class, we use the model class derived from two “parent class”
  - class 1: Forward Stagewise Additive Modeling (referred to as FSAM)
  - class 2: Classification and Regression Tree (CART)



Select the parameterized model “class”

- In order to get the parameterized function model “object” that can be used to calculate the concrete predictions, we need to restrict $\hat{F}$ to be of a member of a parameterized model “class” of functions $F_\theta(\boldsymbol{x};\boldsymbol{\gamma}) \rightarrow \hat{F}$
  - here θ is the hyperparameter, and $\boldsymbol{\gamma}$ is the parameter
- For a Boosted Regression Tree (BRT) model class, we use the model class derived from two “parent class”
  - class 1: Forward Stagewise Additive Modeling (referred to as FSAM)
  - class 2: Classification and Regression Tree (CART)



Select the parameterized model “class” (continued)

- Consequently, the parameterized GBM model has the form

$$F_\theta(\boldsymbol{x};\boldsymbol{\gamma}) = F_\theta(\boldsymbol{x};\{\beta_m,\boldsymbol{\gamma_m}\}_1^M) = \sum_{m=1}^{M}\beta_m b(\boldsymbol{x};\boldsymbol{\gamma_m})$$

  - where M is one of the hyperparameter θ representing the number of rounds
  - $b(\boldsymbol{x};\boldsymbol{\gamma_m})$ is the basis function or weak learner at round m, chosen to be decision tree
  - β_m is the coefficient for each weak learner

  - Note: the process of the parameterization is imposing a constraint condition

- The optimization process will lead to the calculable model “object”

Optimize loss in parameter (γ) space

- Minimize loss

$$F_\theta^*(\boldsymbol{x};\boldsymbol{\gamma}^*) = \arg\min_F E_{y,\boldsymbol{x}}[L(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma}))]$$

  - define $\Phi_\theta(\boldsymbol{\gamma}) = E_{y,\boldsymbol{x}}[L(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma}))]$

$$\rightarrow \boldsymbol{\gamma}^* = \arg\min_{\boldsymbol{\gamma}^*} \Phi_\theta(\boldsymbol{\gamma})$$

- Then numerical minimization method is then applied to get $\boldsymbol{\gamma}^*$
- One of the simplest but popular choice is the steepest-descent
  - in the FSAM framework, at stage m

$$\boldsymbol{g}_m = \left\{\frac{\partial \Phi_\theta(\boldsymbol{\gamma})}{\partial \gamma_j}\right\}_{j=1}^{\#(\boldsymbol{\gamma})}\Bigg|_{\boldsymbol{\gamma}=\boldsymbol{\gamma}_{m-1}}$$

  - make an update along the direction of g_m ("line search")

$$\boldsymbol{\gamma}_m = \boldsymbol{\gamma}_{m-1} - \rho_m \boldsymbol{g}_m$$

  - where ρ_m is chosen to minimize the $\Phi(\boldsymbol{\gamma})$
- However, $\boldsymbol{g}_m$ is not available for parameters $\boldsymbol{\gamma}$ of the decision tree
  - the choice of split variable, value of split & output prediction in each leaf

Optimize loss in function ($F(x)$) space

- Instead of directly seeking the γ values to minimize $\Phi_\theta(\boldsymbol{\gamma})$, we can first find the addition term in function space $\boldsymbol{F}(X_N) \overset{\text{def}}{=} \{F(\boldsymbol{x_i})\}_{i=1}^N$ to minimize $\Phi(\boldsymbol{F}(X_N))$

$$\Phi_\theta(\boldsymbol{\gamma}) = E_{y,\boldsymbol{x}}\left[L\left(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma})\right)\right] \rightarrow \Phi\left(\boldsymbol{F}(X_N)\right) = E_{y,\boldsymbol{x}}\left[L\left(y, \boldsymbol{F}(X_N)\right)\right]$$

- We consider the response of the model $\boldsymbol{F}(X_N)$ as a "parameter" and try to minimize $\Phi\left(\boldsymbol{F}(X_N)\right)$
  - in the provided dataset with size N, we have N of such "parameters" $\boldsymbol{F}(X_N) = \{F(\boldsymbol{x_i})\}_1^N$
  - in the full function space, there are infinite such "parameters"
- Utilizing the idea of steepest-descent on these function space "parameters"
  - in the FSAM framework, at stage m

$$\boldsymbol{g}_m(X_N) = \left\{\frac{\partial\Phi\left(\boldsymbol{F}(X_N)\right)}{\partial F(\boldsymbol{x_i})}\right\}_{i=1}^N \Bigg|_{\boldsymbol{F}(X_N)=\boldsymbol{F_{m-1}}(X_N)}$$

  - make an update along the direction of g_m ("line search")

$$\boldsymbol{F}_m(X_N) = \boldsymbol{F}_{m-1}(X_N) - \rho_m \boldsymbol{g}_m(X_N)$$

  - where ρ_m is used to scale the update along the direction $\boldsymbol{g}_m(X_N)$

Optimize loss in function ($F(x)$) space (continued)

- Once we know the optimal update in function space, we perform the parameterization and training procedures to get calculable $F_\theta^*(\boldsymbol{x};\boldsymbol{\gamma}^*)$
  - for GBM, we now essentially fitting the tree $T_m(\boldsymbol{x};\boldsymbol{\gamma})$ to $\boldsymbol{g}_m(\boldsymbol{X}_N)$
    - $\boldsymbol{g}_m(\boldsymbol{X}_N)$ is called "pseudo residual"
  - usually a top-down greedy procedures is used to construct the tree at stage m

$$T_m(\boldsymbol{x};\boldsymbol{\gamma}) = \sum_{j=1}^{J} \gamma_j I(x \in R_j),$$

    - with parameters $\gamma = \{R_j, \gamma_j\}_1^J$, where R_j is the corresponding region to each leaf and γ_j is the output prediction value of the region.
    - J is the number of terminal nodes (leaves) which is part of hyperparameter θ
    - the construction of the tree involves another level loss function (e.g., square error)
- Then $\boldsymbol{g_m}(\boldsymbol{X_N})$ fitted by $T_m(x;\gamma)$ is used to make the update

$$\boldsymbol{F}_m(X_N) = \boldsymbol{F}_{m-1}(X_N) - \rho_m \boldsymbol{g}_m(X_N) \rightarrow \boldsymbol{F}_{m-1}(x) - \rho_m \boldsymbol{T}_m(X_N;\boldsymbol{\gamma})$$

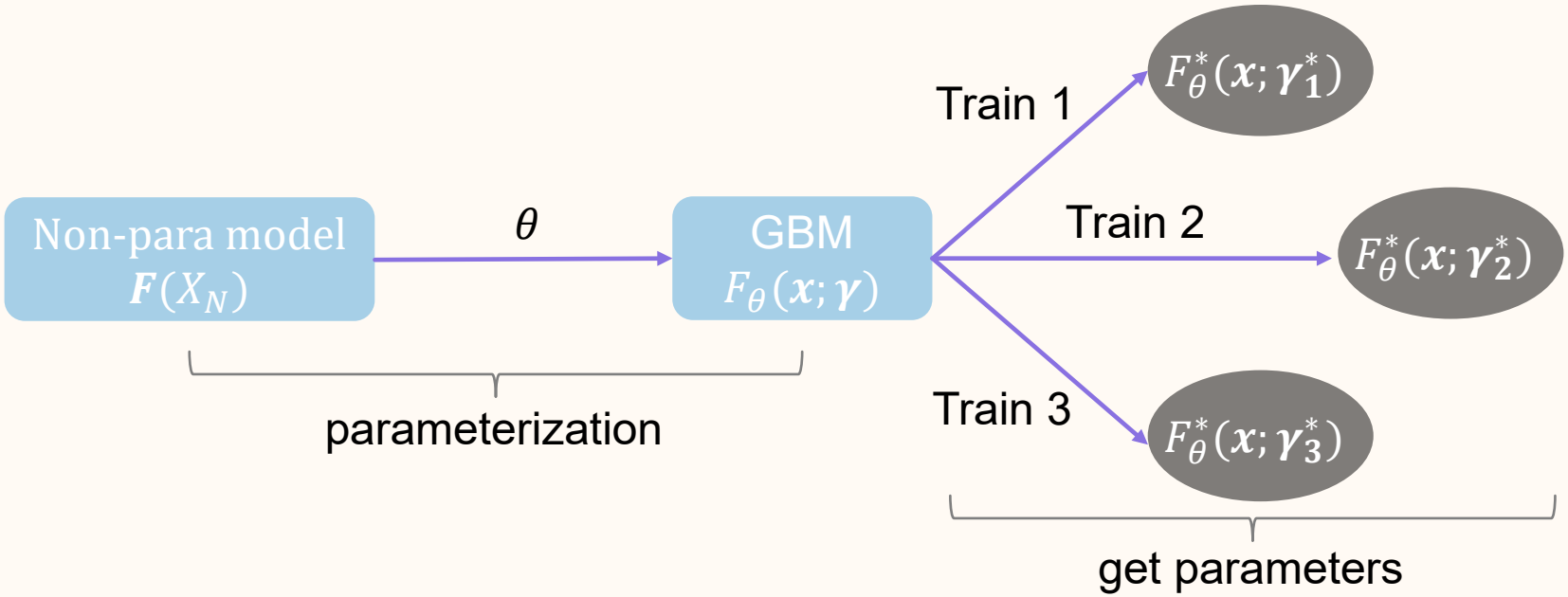
  - where ρ_m is chosen to minimize the $\Phi(\boldsymbol{F}(X_N))$

Commonly used loss functions

TABLE 10.2. *Gradients for commonly used loss functions.*

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Gradient Boost Machine (GBM) summary



- Gradient boosting can be considered to perform the gradient descent in function space $\boldsymbol{F}(X_N)$ first, and then perform the parameterization $F_\theta(\boldsymbol{x};\boldsymbol{\gamma})$ and training model objects $F_\theta^*(\boldsymbol{x};\boldsymbol{\gamma}^*)$

Gradient tree boosting

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

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

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

The "line fit" parameter is merged into terminal region value γ_{jm} optimization

AdaBoost is a special gradient boosting method

- Now, we illustrate that the AdaBoost method can be considered as a special Gradient Boosting method which optimizing in the "margin" $yF(x)$ space
- We are now trying to find the addition term in the margin space $\boldsymbol{H}(X_N) \overset{\text{def}}{=} \boldsymbol{yF}(X_N) = \{y_i F(\boldsymbol{x_i})\}_{i=1}^N$ to minimize $\Phi\big(\boldsymbol{H}(X_N)\big)$
  - recall that the exponential form loss function used in AdaBoost $e^{-yF(x)}$, consequently we can make the equivalent transformation of the loss

$$L\big(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma})\big) = L\big(yF_\theta(\boldsymbol{x};\boldsymbol{\gamma})\big) = L\big(H_\theta(\boldsymbol{x};\boldsymbol{\gamma})\big)$$

  - accordingly

$$\Phi_\theta(\boldsymbol{\gamma}) = E_{y,\boldsymbol{x}}\big[L\big(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma})\big)\big] = E_{y,\boldsymbol{x}}\big[L\big(H_\theta(\boldsymbol{x};\boldsymbol{\gamma})\big)\big]$$

  - in the margin space

$$\Phi\big(\boldsymbol{H}(X_N)\big) = E_{y,\boldsymbol{x}}\big[L\big(\boldsymbol{H}(X_N)\big)\big]$$

AdaBoost is a special gradient boosting method (cont.)

- Utilizing the idea of steepest-descent on the margin space "parameters"
  - in the FSAM framework, at stage m

$$\boldsymbol{h}_m(X_N) = \left\{ \frac{\partial \Phi\left(\boldsymbol{H}(X_N)\right)}{\partial H(\boldsymbol{x_i})} \right\}_{i=1}^{N} \Bigg|_{\boldsymbol{H}(X_N)=\boldsymbol{H_{m-1}}(X_N)}$$

  - Using the exponential form of loss

$$\Phi\left(\boldsymbol{H}(X_N)\right) = E_{y,\boldsymbol{x}}\left[L\left(\boldsymbol{H}(X_N)\right)\right] = E_{y,\boldsymbol{x}}\left[e^{-\boldsymbol{H}(X_N)}\right]$$

  - Then

$$\boldsymbol{h}_m(X_N) = \left\{-e^{-H_{m-1}(\boldsymbol{x}_i)}\right\}_{i=1}^{N}$$

We see that positive margin sample will be "less" boosted, while negative margin sample will be "more" boosted

  - make an update along the direction of g_m ("line search")

$$\boldsymbol{H}_m(X_N) = \boldsymbol{H}_{m-1}(X_N) - \rho_m \boldsymbol{h}_m(X_N) = \boldsymbol{H}_{m-1}(X_N) + \rho_m \left\{e^{-H_{m-1}(\boldsymbol{x}_i)}\right\}_{i=1}^{N}$$

  - where ρ_m is used scale the update along the direction $\boldsymbol{h}_m(X_N)$ minimizing $\Phi\left(\boldsymbol{H}(X_N)\right)$

AdaBoost is a special gradient boosting method (cont.)

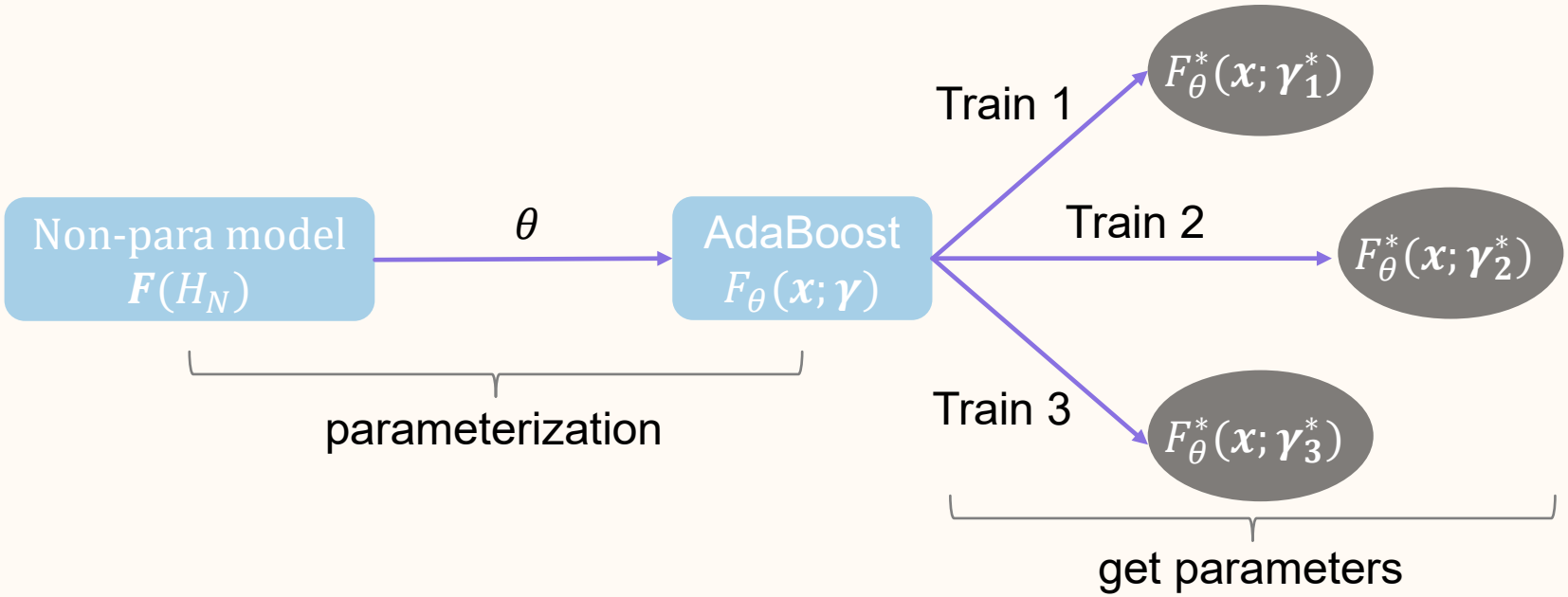
- Once we know the optimal update in margin space, we perform the parameterization and training procedures to get calculable $H_\theta^*(\boldsymbol{x};\boldsymbol{\gamma}^*)$
  - for BDT, we now essentially fitting the $yT_m(\boldsymbol{x};\boldsymbol{\gamma})$ to $\boldsymbol{h}_m(\boldsymbol{X}_N)$

- The construction of the tree is then a process to minimize a special weighted loss function as shown in the following expression

$$L' = \boldsymbol{L}\cdot\left(-\boldsymbol{h}_m(X_N)\right) = \sum_{i=1}^{N} L(\boldsymbol{x_i})\left(-h_m(\boldsymbol{x_i})\right) = \sum_{i=1}^{N} L(\boldsymbol{x_i})e^{-H_{m-1}(\boldsymbol{x_i})}$$

  - where L is the loss function chosen to construct the tree
  - and $e^{-H_{m-1}(x_i)}$ is the w_i^m we see in AdaBoost

AdaBoost is a special gradient boosting method (cont.)



- AdaBoost can be approximately considered to perform the gradient descent in margin space $F(H_N)$ first, and then perform the parameterization $F_\theta(x;\gamma)$ and training model objects $F_\theta^*(x;\gamma^*)$

Regularization

GBDT can be prone to overfitting if not properly regularized, unlike AdaBoost…

Right-sized trees

- No constraint on complexity
  - “Dynamic” choice of optimal tree size for each tree
    - first fit an oversized tree and then perform pruning
  - trees tend to be much too large (especially at the early iterations)
- Method 1: Fix number of terminal nodes
  - ESL
- Method 2: Limit number of terminal nodes or tree depth
  - XGBoost, LightGBM

Shrinkage - Lasso

- Add a shrinkage factor in the stage update of GBM (Algorithm 10.3)

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \nu \sum_{j=1}^{J} \gamma_{jm} I(\boldsymbol{x} \in R_{jm}), 0 < \nu < 1$$

- Empirical evidence shows that smaller values of ν favor better test error, together with larger numbers of total iterations M
  - in practice, usually set $\nu < 0.1$
  - price paid: computation cost
- With small ν, it fits a monotone version of the lasso
  - infinitesimal forward stagewise algorithm (FS_0)
- With $\nu = 1$, it performs the subset selection (L_0 penalty)

Penalized regression

- Now, we show that GBM with shrinkage can be obtained by drawing analogies with penalized linear regression with a large basis expansion.
- Consider the span of all possible J-terminal trees $\mathcal{T} = \{T_k\}$ that could be realized on the training data as basis functions, the linear model can then be set as

$$f(\boldsymbol{x}) = \sum_{k=1}^{K} \alpha_k T_k(\boldsymbol{x}),$$

  - where K is the "size" of the set $\mathcal{T}$
- Utilizing a square error loss for the coefficient α_k estimation with penalty

$$\min_{\alpha} \left\{ \sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{K} \alpha_k T_k(x_i) \right)^2 + \lambda \cdot J(\alpha) \right\},$$

  - where $J(\alpha)$ penalized large α values

$$J(\alpha) = \begin{cases} \sum_{k=1}^{K} |\alpha_k|^2, & \text{ridge} \\ \sum_{k=1}^{K} |\alpha_k|, & \text{lasso} \end{cases}$$

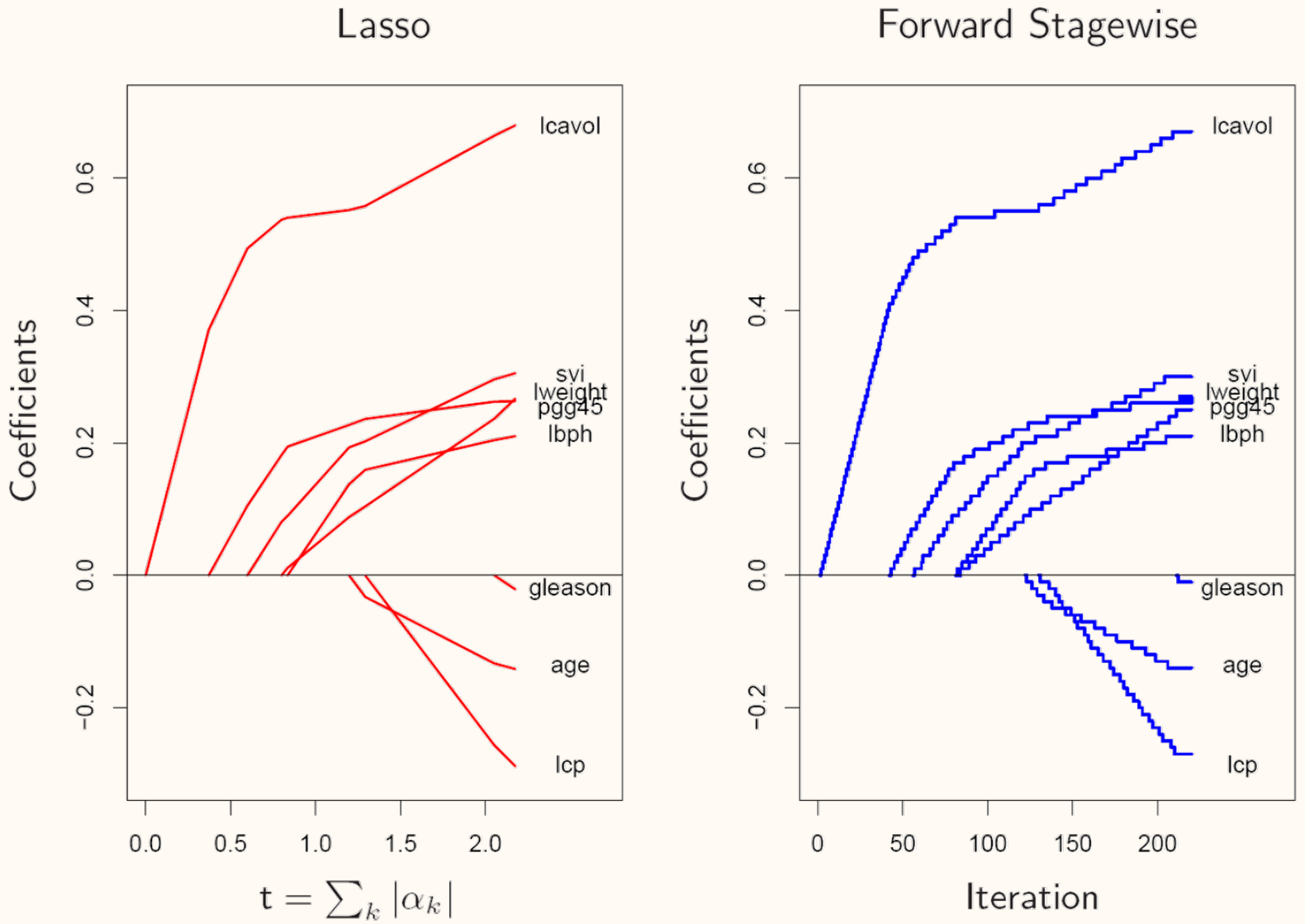
Penalized regression (continued)

- K is usually very large, directly optimizing α_k is not possible
- A feasible approximation solution utilize the forward stagewise framework

Algorithm 16.1 *Forward Stagewise Linear Regression.*

1. Initialize $\check{\alpha}_k = 0,\ k = 1, \ldots, K$. Set $\varepsilon > 0$ to some small constant, and M large.
2. For $m = 1$ to M:
   (a) $(\beta^*, k^*) = \arg\min_{\beta,k} \sum_{i=1}^{N} \left(y_i - \sum_{l=1}^{K} \check{\alpha}_l T_l(x_i) - \beta T_k(x_i) \right)^2$.
   (b) $\check{\alpha}_{k^*} \leftarrow \check{\alpha}_{k^*} + \varepsilon \cdot \text{sign}(\beta^*)$.
3. Output $f_M(x) = \sum_{k=1}^{K} \check{\alpha}_k T_k(x)$.

Penalized regression (continued)



Penalized regression (continued)

- Tree boosting with shrinkage closely resembles the Forward Stagewise Linear Regression (Algorithm 16.1) discussed above
  - learning rate $\nu \rightarrow$ infinitesimal step ϵ
- Similar results observed in AdaBoost (Rosset et al. 2004 a)

- One can view tree boosting with shrinkage as a form of monotone ill-posed regression on all possible (J-terminal) trees $\mathcal{T} = \{T_k\}$ with the lasso penalty as a regularizer

FS tend to be monotone and smoother

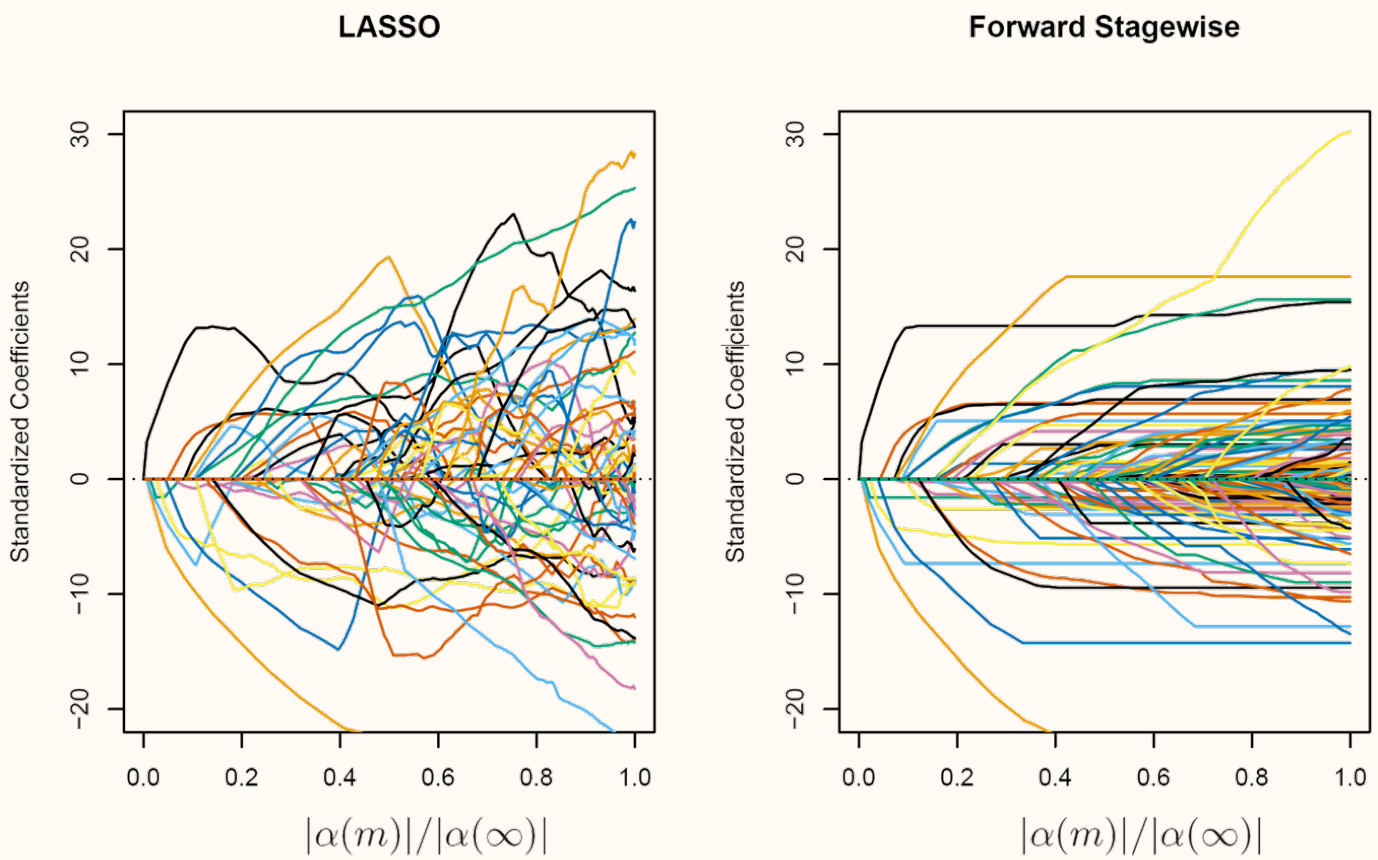


FIGURE 16.3. *Comparison of lasso and infinitesimal forward stagewise paths on simulated regression data. The number of samples is* 60 *and the number of variables is* 1000. *The forward-stagewise paths fluctuate less than those of lasso in the final stages of the algorithms.*

Subsampling

- Training set subsampling
  - select a fraction of training set to perform update (LightGBM: bagging_fraction)
  - frequency to reperform the subsampling (LightGBM: bagging_freq)
- Feature subsampling
  - randomly select a subset of features on each iteration (tree) (LightGBM: feature_fraction)
- Gradient subsampling
  - Gradient-based One-Side Sampling

XGBoost & LightGBM

Open-source ML frameworks utilizing GBDT

XGBoost

- XGBoost (eXtreme Gradient Boosting) 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 solves many data science problems in a fast and accurate way
- The major contributions (from paper)
  - We design and build a highly scalable end-to-end tree boosting system
  - We propose a theoretically justified weighted quantile sketch for efficient proposal calculation
  - We introduce a novel sparsity-aware algorithm for parallel tree learning
  - We propose an effective cache-aware block structure for out-of-core tree learning

XGBoost: Loss function

- A regularized loss is used

$$\Phi' = \Phi + \sum_m \Omega(f_m)$$

  - where

$$\Phi = E_{y,x}\left[L\left(y, F_\theta(\boldsymbol{x};\boldsymbol{\gamma})\right)\right], \qquad \Omega(f_m) = \beta\, J + \frac{1}{2}\lambda \left|\left|\gamma\right|\right|^2$$

- Second-order approximation is used

$$\boldsymbol{g}_m(X_N) = \left\{\frac{\partial \Phi\left(\boldsymbol{F}(X_N)\right)}{\partial F(\boldsymbol{x_i})}\right\}_{i=1}^{N} \Bigg|_{\boldsymbol{F}(X_N)=\boldsymbol{F_{m-1}}(X_N)}$$

$$\boldsymbol{h}_m(X_N) = \left\{\frac{\partial^2 \Phi\left(\boldsymbol{F}(X_N)\right)}{\partial F(\boldsymbol{x_i})^2}\right\}_{i=1}^{N} \Bigg|_{\boldsymbol{F}(X_N)=\boldsymbol{F_{m-1}}(X_N)}$$

- Then

$$\Phi'\left(y, \boldsymbol{F}_{m-1}(X_N) + f_m(X_N)\right) \approx \Phi\left(y, \boldsymbol{F}_{m-1}(X_N)\right) + \boldsymbol{g}_m(X_N)\boldsymbol{f}_m(X_N) + \frac{1}{2}\boldsymbol{h}_m(X_N)\boldsymbol{f}_m^2(X_N) + \sum_m \Omega(f_m)$$

- The corresponding optimal estimation for node j prediction γ_j is

$$\gamma_j^* = -\frac{\sum_{i\in R_j} g_i}{\sum_{i\in R_j} h_i + \lambda}$$

XGBoost: Quantile sketch

- The exact greedy algorithm is very powerful since it enumerates over all possible splitting points greedily, but it's not feasible to do so and an approximation method is needed.

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted*$(I,$ *by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

Algorithm 2: Approximate Algorithm for Split Finding

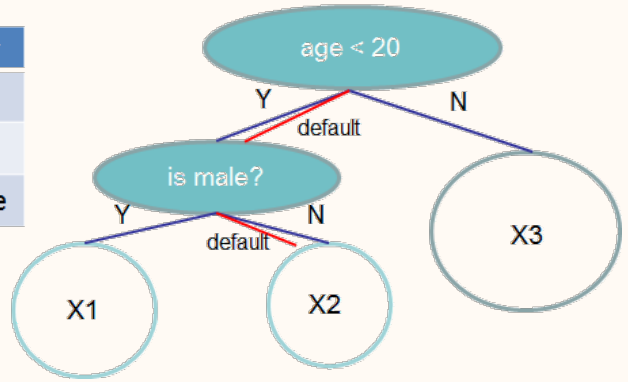
for $k = 1$ ***to*** m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.
  Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ ***to*** m **do**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv}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

- splitting points according to percentiles of feature distribution

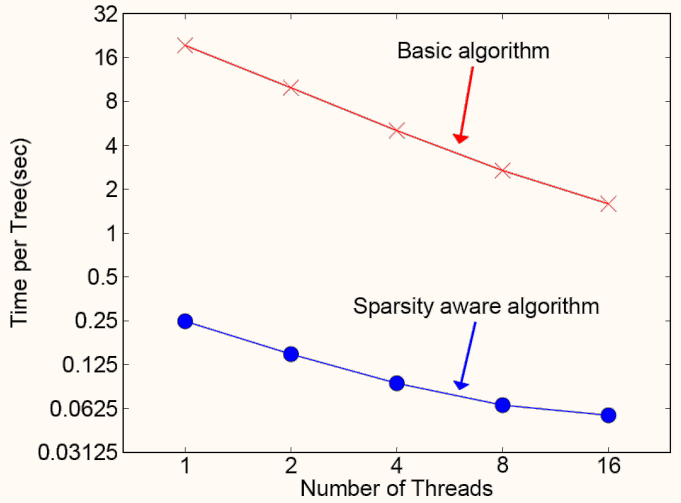
XGBoost: Sparsity-aware

Data

Example	Age	Gender
X1	?	male
X2	15	?
X3	25	female



- It is quite common for the input x to be sparse.
- There are multiple possible causes for sparsity
  - presence of missing values in the data
  - frequent zero entries in the statistics
  - artifacts of feature engineering such as one-hot encoding
- Solution: add a default direction in each tree node
  - when a value is missing in the sparse matrix x, the instance is classified into the default direction
  - there are two choices, choose the optimal one from the trial

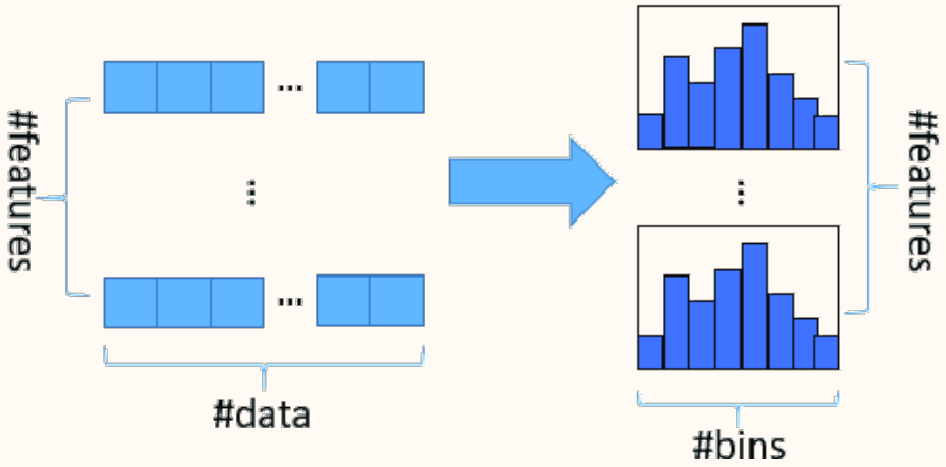


LightGBM

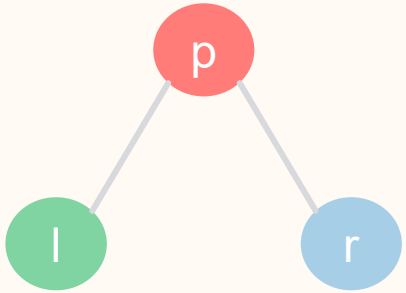
- LightGBM is a gradient boosting framework that uses tree-based learning algorithms
  - it is designed to be distributed and efficient with faster training speed and higher efficiency
  - LightGBM has lower memory usage and better accuracy
  - it supports parallel, distributed, and GPU learning and is capable of handling large-scale data
- Major contribution (compared to XGBoost)
  - histogram-based algorithms
  - Gradient-based One-Side Sampling (GOSS)
  - Exclusive Feature Bundling (EFB)
  - leaf-wise tree grow, categorical features support

LightGBM: Histogram-based algorithm

- Significantly lower memory usage



- Faster speed
  - For example, $hist_p - hist_l = hist_r$



LightGBM: Gradient-based One-Side Sampling (GOSS)

- The weight in AdaBoost is a good indicator for samples to pay attention
  - the gradient for each data instance in GBDT provides good analogy
  - if an instance is associated with a small gradient, the training error for this instance is small and it is already well-trained
- A straightforward idea is to discard those data instances with small gradients
  - however, the data distribution will be changed by doing so, which will hurt the accuracy of the learned model.
  - to avoid this problem the Gradient-based One-Side Sampling (GOSS) is proposed

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 $\times$ len(I) , randN $\leftarrow$ b $\times$ len(I)
for $i = 1$ ***to*** d **do**
  preds $\leftarrow$ models.predict(I)
  g $\leftarrow$ $loss$(I, preds), w $\leftarrow$ {1,1,...}
  sorted $\leftarrow$ GetSortedIndices(abs(g))
  topSet $\leftarrow$ sorted[1:topN]
  randSet $\leftarrow$ RandomPick(sorted[topN:len(I)], randN)
  usedSet $\leftarrow$ topSet + randSet
  w[randSet] $\times =$ fact ▷ Assign weight $fact$ to the small gradient data.
  newModel $\leftarrow$ L(I[usedSet], $-$ g[usedSet], w[usedSet])
  models.append(newModel)

LightGBM: Exclusive Feature Bundling

- High-dimensional data are usually very sparse.
  - the sparsity of the feature space provides a possibility of designing a nearly lossless approach to reduce the number of features.
- Q1: Which features should be bundled?
  - the graph coloring problem
  - 1. we construct a graph with weighted edges, whose weights correspond to the total conflicts between features
  - 2. we sort the features by their degrees in the graph in the descending order
  - 3. we check each feature in the ordered list, and either assign it to an existing bundle with a small conflict or create a new bundle

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder $\leftarrow$ G.sortByDegree()
bundles $\leftarrow$ {}, bundlesConflict $\leftarrow$ {}
for i ***in*** *searchOrder* **do**
    needNew $\leftarrow$ True
    for $j = 1$ ***to*** *len(bundles)* **do**
        cnt $\leftarrow$ ConflictCnt(bundles[j],F[i])
        if *cnt* + *bundlesConflict[i]* $\leq K$ **then**
            bundles[j].add(F[i]), needNew $\leftarrow$ False
            break
    if *needNew* **then**
        Add $F[i]$ as a new bundle to *bundles*
Output: *bundles*

LightGBM: Exclusive Feature Bundling

- Q2: How should we merge the bundle?
  - need a good way of merging the features in the same bundle in order to reduce the corresponding training complexity

- The key is to ensure that the values of the original features can be identified from the feature bundles.
  - construct a feature bundle by letting exclusive features reside in different bins
  - this can be done by adding offsets to the original values of the features

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: $newBin, binRanges$

Hyperparameters tuning

Control Overfitting

When you observe high training accuracy, but low test accuracy, it is likely that you encountered overfitting problem.

There are in general two ways that you can control overfitting in XGBoost:

- The first way is to directly control model complexity.
  - This includes `max_depth`, `min_child_weight` and `gamma`.
- The second way is to add randomness to make training robust to noise.
  - This includes `subsample` and `colsample_bytree`.
  - You can also reduce stepsize `eta`. Remember to increase `num_round` when you do so.

Faster training performance

There's a parameter called `tree_method`, set it to `hist` or `gpu_hist` for faster computation.

Handle Imbalanced Dataset

For common cases such as ads clickthrough log, the dataset is extremely imbalanced. This can affect the training of XGBoost model, and there are two ways to improve it.

- If you care only about the overall performance metric (AUC) of your prediction
  - Balance the positive and negative weights via `scale_pos_weight`
  - Use AUC for evaluation
- If you care about predicting the right probability
  - In such a case, you cannot re-balance the dataset
  - Set parameter `max_delta_step` to a finite number (say 1) to help convergence

Notes on Parameter Tuning — xgboost 1.7.5 documentation

For Better Accuracy

- Use large `max_bin` (may be slower)
- Use small `learning_rate` with large `num_iterations`
- Use large `num_leaves` (may cause over-fitting)
- Use bigger training data
- Try `dart`

Deal with Over-fitting

- Use small `max_bin`
- Use small `num_leaves`
- Use `min_data_in_leaf` and `min_sum_hessian_in_leaf`
- Use bagging by set `bagging_fraction` and `bagging_freq`
- Use feature sub-sampling by set `feature_fraction`
- Use bigger training data
- Try `lambda_l1`, `lambda_l2` and `min_gain_to_split` for regularization
- Try `max_depth` to avoid growing deep tree
- Try `extra_trees`
- Try increasing `path_smooth`

Parameters Tuning — LightGBM 3.3.2 documentation

Summary

- Boosting algorithms can be viewed as a gradient descent process on a provided loss function with certain constraint conditions
- To avoid overfitting, we can constraint the training by limiting model complexity, shrinkage update, subsampling etc.

- XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. LightGBM is another successful GBDT framework which more focus on performance and scalability

References

- Textbook
  - ESL Chapter 10, 16
- Paper
  - Friedman, Jerome H. “Greedy Function Approximation: A Gradient Boosting Machine.” *The Annals of Statistics* 29, no. 5 (October 2001): 1189–1232. https://doi.org/10.1214/aos/1013203451.
- Documents & Webpages
  - XGBoost Documentation — xgboost 1.7.5 documentation
  - Welcome to LightGBM’s documentation! — LightGBM 3.3.2 documentation

Backup

AdaBoost Recap

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Forward Stagewise Additive Modeling

Effectively optimizing the exponential loss with the Newton-like method

Focus on mistakes by reweighting

Boosting Essentials

- The target of a boosting algorithm is defined by the loss function
  - AdaBoost: exponential loss
  - Gradient: square, absolute, Huber, Deviance, …
- The framework that make the process practical is the FSAM
  - Forward, Stagewise: new iterations don't affect the past results
  - Additive: ensemble multiple weak learners
- The choice of the weak learner
  - E.g., decision tree/stump

Boosting Essentials (Continued)

- The stage update to minimize the loss
  - AdaBoost - Newton
    - Constraint: specify the form of weak learner $h(x)$ (algorithm + hyperparameters)
    - Update: optimize the addition $h(x)$ to **minimize** $\boldsymbol{L(y, f(x) + ch(x))}$
      - approximately minimizing the weighted square choice of $f(x) \in \{-1,1\}$
      - 2nd order
    - Input twist: reweighting
  - Gradient - Cauthy
    - Constraint: directly investigating the **steepest direction** to decrease the loss
    - Update: find the optimal weak learner f that minimize the loss **along the "line"**
      - 1st order
    - Input twist: pseudo residual

Boosting Trees

- A tree can be formally expressed as

$$T(x;\Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j),$$

  - with parameters $\Theta = \{R_j, \gamma_j\}_1^J$. J is the number of terminal nodes (leaves) which is a hyperparameter for tuning. R_j is the corresponding region to each leaf and γ_j is the output prediction value of the region.
- In each stage, we need to find R_j and γ_j
  - Finding γ_j: typically trivial and often $\widehat{\gamma_j} = \overline{y_j}$
  - Finding R_j: the difficult part, need an approximating solution like the “greedy top-down recursive partitioning”

Gradient Boosting Tree

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

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

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Forward Stagewise Additive Modeling

Focus on mistakes by compensating pseudo residual

Effectively optimizing the loss with the Cauthy-like method