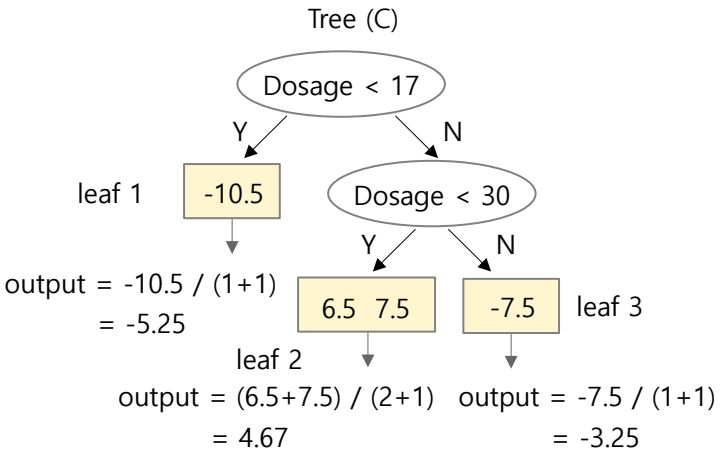
Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
12	-10	0.5	-10.5	-1.08	-8.92
22	7	0.5	6.5	1.90	5.10
28	8	0.5	7.5	1.90	6.10
32	-7	0.5	-7.5	-0.48	-6.52



11. Extreme Gradient Boosting

(XGBoost: Regression)

Part 1: Training and Prediction process

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

MX-AI

▪ Extreme Gradient Boosting (XGBoost) : Overview

- XGBoost was proposed by Tianqi Chen and Carlos Guestrin in 2016 to improve existing GBM to effectively process large amounts of data. Regularization and pruning were considered when branching the decision trees, and approximation approaches were proposed when determining the optimal split point. Additionally, several techniques were introduced, such as missing value handling, parallel processing to quickly find the optimal split points, and cache-aware access, etc.

XGBoost: A Scalable Tree Boosting System

Tianqi Chen
University of Washington
tqchen@cs.washington.edu

Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

ABSTRACT

Tree boosting is a highly effective and widely used machine learning method. In this paper, we describe a scalable end-to-end tree boosting system called XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. We propose a novel sparsity-aware algorithm for sparse data and weighted quantile sketch for approximate tree learning. More importantly, we provide insights on cache access patterns, data compression and sharding to build a scalable tree boosting system. By combining these insights, XGBoost scales beyond billions of examples using far fewer resources than existing systems.

- Main contents

1. Exact Greedy Algorithm for Split Finding
2. Approximate Algorithm for Split Finding
3. Weighted quantile sketch
4. Sparsity-aware algorithm
5. Column Block for Parallel Learning
6. Cache patterns
7. Data compression and sharding

MX-AI

▪ Exact Greedy Algorithm for Split Finding : Training process

- Split the node with all split point candidates and select the most optimal split point. It is the most accurate but takes a long time.
- This example is from the YouTube "StatQuest with Josh Starmer", XGBppst Part 1.

	feature (x)	target (y)	initial prediction	residual (1)
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$
i=1	12	-10	0.5	-10.5
i=2	22	7	0.5	6.5
i=3	28	8	0.5	7.5
i=4	32	-7	0.5	-7.5

1) Initialize the prediction of target y and compute the residual (1).

$\hat{y}_i^{(0)}=0.5$ ← average value of y is good initial value.

$r_1^{(1)}=y-\hat{y}_1^{(0)}=-10-0.5=-10.5$

$r_2^{(1)}=7-0.5=6.5$

- If a leaf node has many residuals with different signs, the similarity score will be small because they cancel each other out.
- That is, the more identical signs are gathered in a leaf node, the more the score increases.
- The less data a leaf node has, the smaller its score will be due to the influence of λ.

2) Calculate the similarity score of the root node containing the residual (1).

$$score = \frac{\left(\sum_{i\in root} r_i^{(1)}\right)^2}{|residual|_{root}+\lambda} = \frac{(-10.5+6.5+7.5-7.5)^2}{4+1}=3.2$$

← λ : regularization constant
|residual| : the number of residuals

3) For every split point of feature x, we split the root node and calculate the score of each leaf node. The larger the sum of the scores of child nodes, the more likely it is to be a better split. In the example below, tree (A) is better than (B)

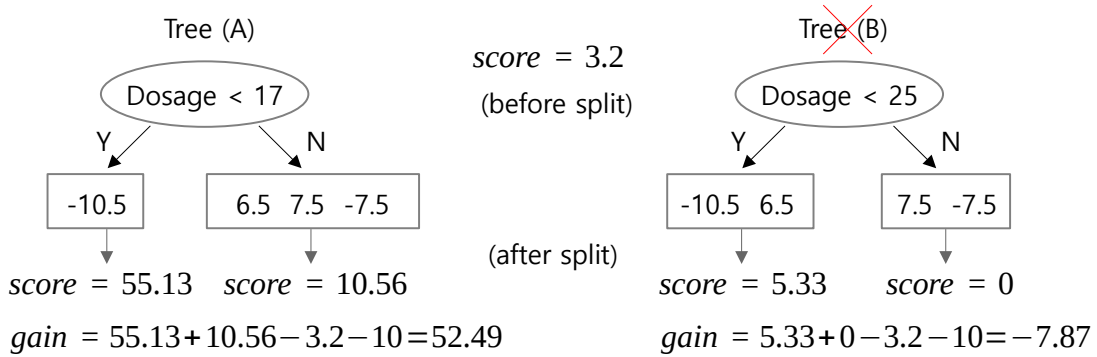
$$score = \frac{\left(\sum_{i\in right} r_i^{(1)}\right)^2}{|residual|_{right}+\lambda} = \frac{(6.5+7.5-7.5)^2}{3+1}=10.56$$

$$score = \frac{\left(\sum_{i\in left} r_i^{(1)}\right)^2}{|residual|_{left}+\lambda} = \frac{(-10.5)^2}{1+1}=55.13$$

MX-AI

▪ Exact Greedy Algorithm for Split Finding : Training process

	feature (x)	target (y)	initial prediction	residual (1)
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$
i=1	12	-10	0.5	-10.5
i=2	22	7	0.5	6.5
i=3	28	8	0.5	7.5
i=4	32	-7	0.5	-7.5



$gain = 55.13 + 10.56 - 3.2 - 10 = 52.49$

$gain = 5.33 + 0 - 3.2 - 10 = -7.87$

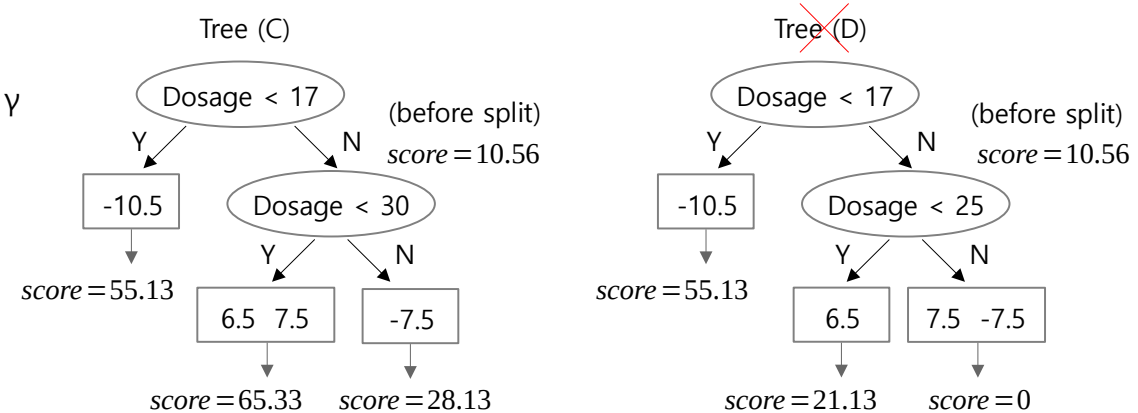
4) Calculate the gain using the similarity score before and after the split.

Gain = sum of the scores after split – the score before split – γ

(γ : pruning constant)

When the gamma, γ = 10, the gain of Tree(A) is greater than that of Tree(B). Among all candidate splits, the one with the highest gain is selected (Tree A). However, if gain ≤ 0, the node will not be split. This is because there is no benefit to split it. The larger the gamma γ, the less likely the node will be split, and vice versa. the γ is pruning constant.

5) Split the child nodes to a predefined depth in the same way.
In the example below, tree (C) is better than (D)



$gain = 65.33 + 28.13 - 10.56 - 10 = 72.9$

$gain = 21.13 + 0 - 10.56 - 10 = 0.57$

MX-AI

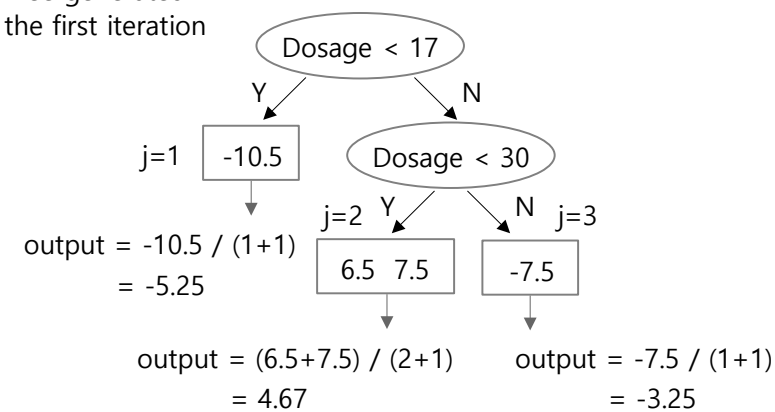
▪ Exact Greedy Algorithm for Split Finding : Training process

	feature (x)	target (y)	initial prediction	residual (1)		
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
i=1	12	-10	0.5	-10.5	-1.08	-8.92
i=2	22	7	0.5	6.5	1.90	5.10
i=3	28	8	0.5	7.5	1.90	6.10
i=4	32	-7	0.5	-7.5	-0.48	-6.52

* Residual has decreased.

* Tree generated in the first iteration

Tree (C)



6) Calculate the output values for the leaf nodes of the final tree.

$$\text{output value } (w_j) = \frac{\sum_{i \in \text{leaf}} r_i^{(1)}}{|residual|_{\text{leaf}} + \lambda}$$ (j: leaf node number)

7) Calculate new predictions using the output values (w).

$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + \eta\, w_j$$

(η: learning rate, w: output value, j: Leaf node number to which data point i belongs.)

$$\hat{y}_1^{(1)} = \hat{y}_1^{(0)} + \eta\, w_1 = 0.5 + 0.3 \times (-5.25) = -1.08$$
$$\hat{y}_2^{(1)} = \hat{y}_2^{(0)} + \eta\, w_2 = 0.5 + 0.3 \times 4.67 = 1.90$$
$$\hat{y}_3^{(1)} = \hat{y}_3^{(0)} + \eta\, w_2 = 0.5 + 0.3 \times 4.67 = 1.90$$
$$\hat{y}_4^{(1)} = \hat{y}_4^{(0)} + \eta\, w_3 = 0.5 + 0.3 \times (-3.25) = -0.48$$

8) Calculate new residuals using the new predictions calculated in step 7.

$$r_1^{(2)} = y_1 - \hat{y}_1^{(1)} = -10 - (-1.08) = -8.92$$
$$r_2^{(2)} = y_2 - \hat{y}_2^{(1)} = 7 - 1.9 = 5.1$$
$$r_3^{(2)} = y_3 - \hat{y}_3^{(1)} = 8 - 1.9 = 6.1$$
$$r_4^{(2)} = y_4 - \hat{y}_4^{(1)} = -7 - (-0.48) = -6.52$$

9) Repeat steps (2) through (8).

- The residuals continue to decrease and the predicted y gets closer to the actual y.

MX-AI

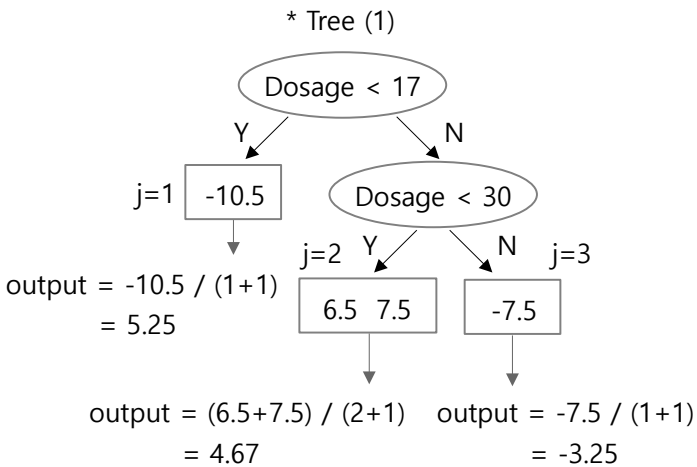
- Exact Greedy Algorithm for Split Finding : Prediction process
  - The target value of the test data can be predicted using the trees saved during the training process. (the same as the GBM.)

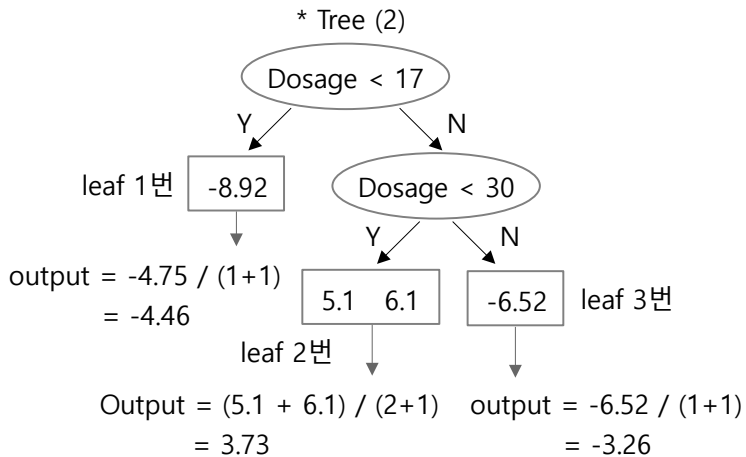
Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
12	-10	0.5	-10.5	-1.08	-8.92
22	7	0.5	6.5	1.90	5.10
28	8	0.5	7.5	1.90	6.10
32	-7	0.5	-7.5	-0.48	-6.52

feature (x_test) → Drug dosage; target (y_pred) → Drug effect

Drug dosage	Drug effect
18	?

test data





10) Predict the target value of the test data using Tree (1) and Tree (2).

$$\hat{y}^{test} = \hat{y}^{(0)} + \eta * \text{tree(1).predict(x\_test)} + \eta * \text{tree(2).predict(x\_test)}$$

$$= 0.5 + 0.3 * 4.67 + 0.3 * 3.73 = 3.02$$

$\eta = 0.3$, $\hat{y}^{(0)}$: initial prediction $\hat{y}^{test}$: predicted target value of the test data.

- For the sake of a simple example, we only created two trees.
- In the actual training, multiple trees are generated by increasing the number of iterations (m) until the residuals become sufficiently small.
- The actual split points of Tree (1) and (2) will be different.

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0

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

for k = 1 **to** d **do**:

  $G_L=0,\ \ H_L=0$

  for j in sorted(I, by $x_{i,k}$) **do**:

    $G_L=G_L+g_i,\ \ H_L=H_L+h_i$

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

    $gain=max\left(gain,\ \frac{G_L^2}{H_L+\gamma}+\frac{G_R^2}{H_R+\gamma}-\frac{G^2}{H+\gamma}\right)$

  end

end

Output: Split with max gain

11. Extreme Gradient Boosting

(XGBoost: Regression)

Part 2: Exact Greedy Algorithm analysis

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Regularized Learning Objective (Chapter 2.1 in the paper)

- Create an objective function for optimizing the t-th tree. Optimize the output value (w) of this tree.
- Objective : regularized loss function

The output value of the node to which x_i belongs in Tree(t).

The output value of the j-th node in Tree (t)

$$L^{(t)} = \underbrace{\sum_{i=1}^{n} \frac{1}{2}\left[y_i - \left(\hat{y}_i^{(t-1)} + w_{x_i}^{(t)}\right)\right]^2}_{\text{mean squared error}} + \underbrace{\gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^{(t)\,2}}_{\text{regularization term}}$$

n: the number of data points

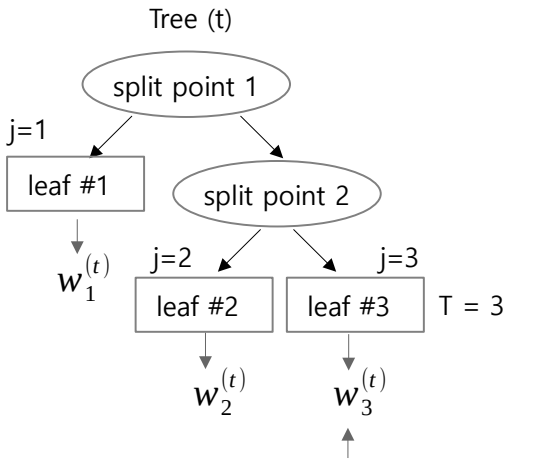
T: the number of leaf nodes

$$\hat{y}_i^{(t-1)} = \hat{y}_i^{(0)} + w_{x_i}^{(1)} + w_{x_i}^{(2)} + \ldots + w_{x_i}^{(t-1)}$$

$$\hat{y}_i^{(t-1)} = \hat{y}_i^{(0)} + \eta\, w_{x_i}^{(1)} + \eta\, w_{x_i}^{(2)} + \ldots + \eta\, w_{x_i}^{(t-1)}$$

* shrinkage factor η (learning rate)

The larger the T and the larger the w, the larger the penalty.



- Objective : $\min_{w_j^{(t)}} L^{(t)}$ ← Find the output value $w^{(t)}_j$ that minimizes the loss $L^{(t)}$ of Tree (t).

$$\frac{\partial L^{(t)}}{\partial w_j^{(t)}} = 0$$

← Find $w^{(t)}_j$ from this equation. Once $w^{(t)}_j$ is determined, the optimal split point is also determined.

$$L^{(t)} = \sum_{i=1}^{n} \frac{1}{2}\left[y_i - \left(\hat{y}_i^{(t-1)} + w_{x_i}^{(t)}\right)\right]^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^{(t)2}$$

To differentiate $L^{(t)}$ with respect to $w^{(t)}_j$, we need to convert it to an expression for $w^{(t)}_j$.

- Use Taylor series

MX-AI

- Regularized Learning Objective (Chapter 2.1 in the paper)

data i-oriented expression. (i=1,2,...n) leaf node j-oriented expression. (j=1,2,...T)

$$L^{(t)} = \sum_{i=1}^{n} l\left(y_i,\ \hat{y}_i^{(t-1)} + w_{x_i}^{(t)}\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^{(t)2}$$

we need to convert it to an expression for $w^{(t)}_j$., leaf node j-oriented expression.

(Taylor series)

$$f(x+a, y+b) = f(x,y) + \frac{\partial f}{\partial x}a + \frac{\partial f}{\partial y}b + \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}a^2 + \frac{\partial^2 f}{\partial y^2}b^2 + 2\frac{\partial^2 f}{\partial x \partial y}ab\right) + \ldots$$

$$l\left(y_i, \hat{y}_i^{(t-1)} + w_{x_i}^{(t)}\right) = l\left(y_i, \hat{y}_i^{(t-1)}\right) + \frac{\partial l}{\partial \hat{y}_i^{(t-1)}} w_{x_i}^{(t)} + \frac{1}{2}\frac{\partial^2 l}{\partial \hat{y}_i^{(t-1)2}} w_{x_i}^{(t)2} + \ldots$$

$$L^{(t)} \simeq \sum_{i=1}^{n} \frac{1}{2}\left[\left(y_i - \hat{y}^{(t-1)}\right)^2 + g_i w_{x_i} + \frac{1}{2} h_i w_{x_i}^2\right] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

This term is independent of w_j from the Tree(t). The superscript (t) of w is omitted.

$$g_i = \frac{\partial \frac{1}{2}\left(y_i - \hat{y}^{(t-1)}\right)^2}{\partial \hat{y}^{(t-1)}} = -\left(y_i - \hat{y}^{(t-1)}\right) = -residual$$

$$h_i = \frac{\partial^2 \frac{1}{2}\left(y_i - \hat{y}^{(t-1)}\right)^2}{\partial \hat{y}^{(t-1)2}} = 1$$

← This value is always 1 for regression with MSE, but not for classification with CE.

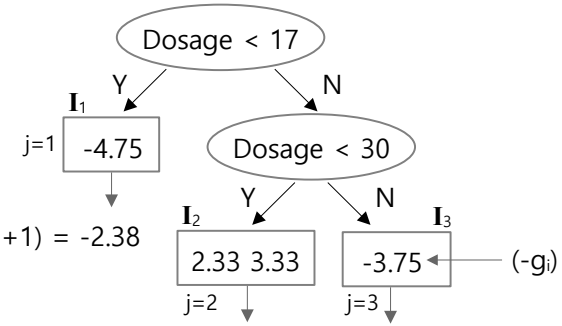
$$\sum_{i=1}^{n} g_i w_{x_i} = 4.75 * (-2.38) + (-2.33) * 1.89 + (-3.33) * 1.89 + 3.75 * (-1.88)$$

$$\sum_{j=1}^{T} \left(\sum_{i \in I_j} g_i\right) w_j = 4.75 * (-2.38) + (-2.33 - 3.33) * 1.89 + 3.75 * (-1.88)$$

$$\tilde{L}^{(t)} \simeq \sum_{i=1}^{n} \left(g_i w_{x_i} + \frac{1}{2} h_i w_{x_i}^2\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

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

← Differentiable with respect to w_j.



w_1 = -4.75 / (1+1) = -2.38

w_2 = (2.33+3.33) / (2+1) = 1.89

w_3 = -3.75 / (1+1) = -1.88

$$\sum_{i \in I_1} g_i = -(-4.75) \qquad \sum_{i \in I_2} g_i = -(2.33 + 3.33)$$

← - (sum of residuals)

$$\sum_{i \in I_1} h_i = 1 \qquad \sum_{i \in I_2} h_i = 2$$

← the number of residuals

MX-AI

- Regularized Learning Objective (Chapter 2.1 in the paper)

$$\tilde{L}^{(t)} \simeq \sum_{j=1}^{T}\left[\left(\sum_{i\in I_j} g_i\right)w_j + \frac{1}{2}\left(\sum_{i\in I_j} h_i+\lambda\right)w_j^2\right] + \gamma T$$ ← equation (4)

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

Replace w in equation (4) with equation (5).

equation (6) → $$\tilde{L}^{(t)} \simeq -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i+\lambda} + \gamma T$$ ←

Equation 6 can be used as an impurity score to evaluate a tree. The smaller this score, the better the performance of a tree.

* Differentiate L with respect to w_j

$$\frac{\partial \tilde{L}^{(t)}}{\partial w_j} = \sum_{j=1}^{T}\left[\sum_{i\in I_j} g_i + \left(\sum_{i\in I_j} h_i+\lambda\right)w_j\right] = 0$$

Since it is a greedy algorithm, it optimizes each individual leaf node.
Derivative of the jth leaf node. = 0.

$$\sum_{i\in I_j} g_i + \left(\sum_{i\in I_j} h_i+\lambda\right)w_j = 0$$

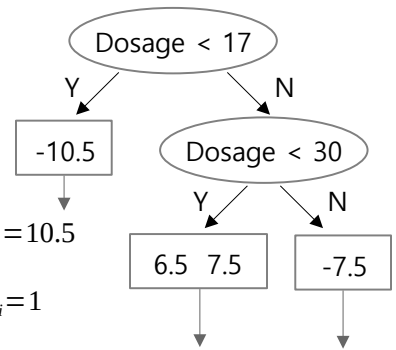
- Optimal output value of jth leaf node

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i+\lambda} = \frac{\text{sum of residuals}}{\text{the number of residuals} + \lambda}$$

equation (5)

$$g_i = -\left(y_i - \hat{y}^{(t-1)}\right) = -residual$$

Tree (C)



$G_1 = \sum_{i\in I_1} g_i = 10.5$

$H_1 = \sum_{i\in I_1} h_i = 1$

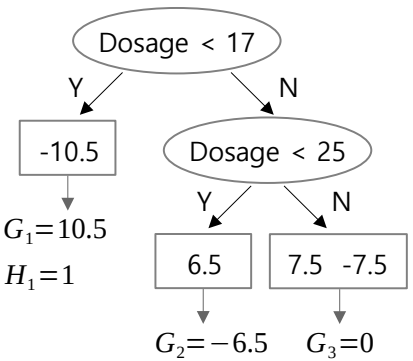
$G_2 = \sum_{i\in I_2} g_i = -6.5 - 7.5 = -14$

$H_2 = \sum_{i\in I_2} h_i = 1+1 = 2$

$G_3 = 7.5$

$H_3 = \sum_{i\in I_3} h_i = 1$

Tree (D)



$G_1 = 10.5$

$H_1 = 1$

$G_2 = -6.5$

$H_2 = 1$

$G_3 = 0$

$H_3 = 2$

$\lambda = 1,\ \gamma = 10,\ T = 3$

$$\tilde{L}^{(t)} = -\frac{1}{2}\frac{10.5^2+(-14)^2+7.5^2}{(1+1)+(2+1)+(1+1)} + 10\times 3 = 11.82 \quad < \quad \tilde{L}^{(t)} = -\frac{1}{2}\frac{10.5^2+(-6.5)^2+0^2}{(1+1)+(1+1)+(2+1)} + 10\times 3 = 19.11$$

MX-AI

▪ Regularized Learning Objective (Chapter 2.1 in the paper)

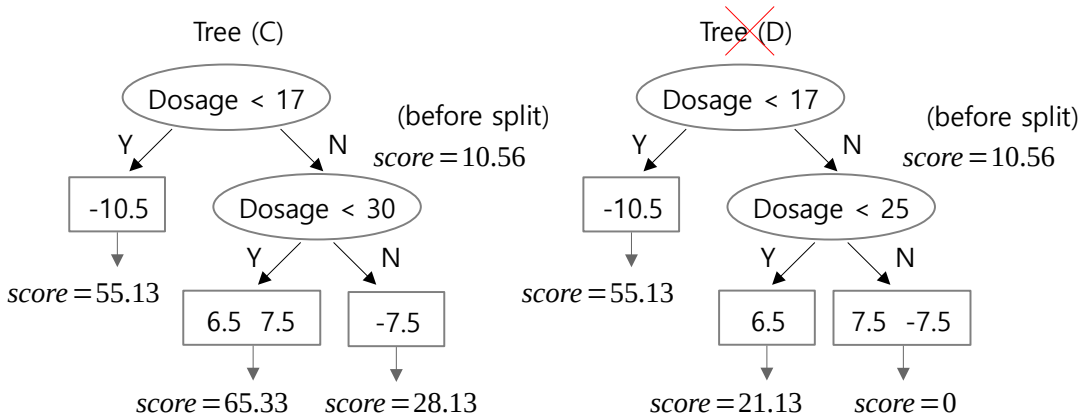
Equation (6) in the paper is a measurement calculated using the completed tree. But T is not known while splitting the tree. This is because the tree is not yet complete and the number of leaf node, T is unknown.

$$\widetilde{L}^{(t)} \simeq -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i+\lambda} + \gamma T \quad \leftarrow \text{equation (6)}$$

When splitting a node using the greedy algorithm, use the following equation (7). Compare the the score before splitting and the sum of the left and right scores after splitting. If the sum of scores after splitting is greater, a gain has occurred and the node is split. Equation (6) is the loss to be minimized, and Equation (7) is the gain to be maximized. So the sign changed to plus.

$$\underbrace{L_{split}}_{\text{gain}} = \frac{1}{2}\Big[\underbrace{\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda}}_{\text{the score of left node}} + \underbrace{\frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i+\lambda}}_{\text{the score of right node}} - \underbrace{\frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i+\lambda}}_{\text{parent's score after splitting}}\Big] - \gamma \quad \leftarrow \text{equation (7)}$$

The sum of scores after splitting



$gain = 65.33+28.13-10.56-10=72.9 > gain = 21.13+0-10.56-10=0.57$

$\lambda=1,\ \gamma=10$

The score and gain used in the previous video were calculated using equation (7). Since only the size will be compared, 1/2 was not multiplied.

MX-AI

- Regularized Learning Objective (Chapter 2.1 in the paper)

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0

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

for k = 1 **to** d **do**:

$G_L=0,\quad H_L=0$

split point list

for j in sorted(I, by $x_{i,k}$) **do**:

$G_L=G_L+g_i,\quad H_L=H_L+h_i$

$G_R=G-G_L,\quad H_R=H-H_L$

$$gain=max\left(gain,\ \frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G^2}{H+\lambda}\right)$$

end

end

Output: Split with max gain ←

```
if gain > γ:
    split
else
    no split
```

← The γ of Equation (7) is considered here.

$$G_L=\sum_{i\in I_L} g_i,\quad H_L=\sum_{i\in I_L} h_i$$
$$G_R=\sum_{i\in I_R} g_i,\quad H_R=\sum_{i\in I_R} h_i$$

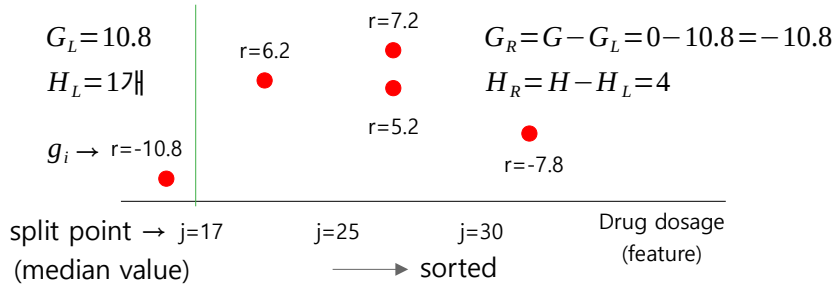
Source : Algorithm 1 in the paper (Some terms have been corrected.)

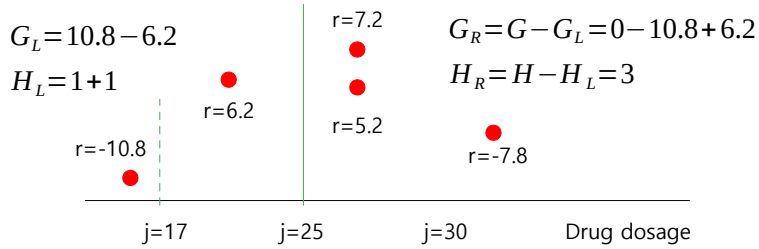
feature (x) → target (y) → residual (1) →

Drug dosage	Drug effect	$r_{i,1}$
12	-10	-10.8
22	7	6.2
28	6	5.2
28	8	7.2
32	-7	-7.8

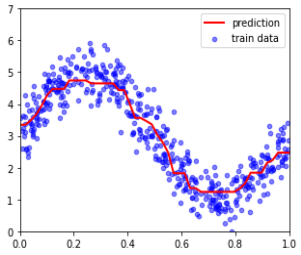
$G=10.8-6.2-5.2-7.2+7.8=0$

$H=5$





```python
# [MXML-11-03] MyXGBoostRegressor.py
# Upgraded version of CART in [MXML-02-07] [MXML-02-11] for XGBoost
import numpy as np
from collections import Counter
import copy

# Implement "Exact Greedy Algorithm for Split Finding" presented
# in XGBoost paper. [1] Tianqi Chen et, al., 2016, XGBoost: A
# Scalable Tree Boosting
class MyXGBRegressionTree:
    def __init__(self, max_depth, reg_lambda, prune_gamma):
        self.max_depth = max_depth      # depth of the tree
        self.reg_lambda = reg_lambda    # regularization constant
        self.prune_gamma = prune_gamma  # pruning constant
        self.estimator1 = None          # tree result-1
        self.estimator2 = None          # tree result-2
        self.feature = None             # feature x.
        self.residual = None            # residuals
        self.base_score = None          # initial prediction

    # [1] 2.1 Regularized Learning Objec
    # Algorithm 1: Exact Greedy Algorith
    # Split a node into left and right.
    # with highest gain and split the no
    def node_split(self, did):
        r = self.reg_lambda
        max_gain = -np.inf
        d = self.feature.shape[1]      #
        G = -self.residual[did].sum() #
        H = did.shape[0]               #
        p_score = (G ** 2)/(H + r)     #
```



11. Extreme Gradient Boosting

(XGBoost: Regression)

Part 3: Implementation of XGBoost Regression from scratch

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

- Implementation of XGBoost regression from scratch

```python
# [MXML-11-03] MyXGBoostRegressor.py
# Upgraded version of CART in [MXML-02-07] [MXML-02-11] for XGBoost
# Source: www.youtube.com/@meanxai
import numpy as np
from collections import Counter
import copy

# Implement "Exact Greedy Algorithm for Split Finding" presented
# in XGBoost paper. [1] Tianqi Chen et, al., 2016, XGBoost: A
# Scalable Tree Boosting
class MyXGBRegressionTree:
    def __init__(self, max_depth, reg_lambda, prune_gamma):
        self.max_depth = max_depth      # depth of the tree
        self.reg_lambda = reg_lambda    # regularization constant
        self.prune_gamma = prune_gamma  # pruning constant
        self.estimator1 = None          # tree result-1
        self.estimator2 = None          # tree result-2
        self.feature = None             # feature x.
        self.residual = None            # residuals
        self.base_score = None          # initial prediction

    # [1] 2.1 Regularized Learning Objective
    # Algorithm 1: Exact Greedy Algorithm for Split Finding
    # Split a node into left and right. Find the best split point
    # with highest gain and split the node with the point.
    def node_split(self, did):
        r = self.reg_lambda
        max_gain = -np.inf
        d = self.feature.shape[1]      # feature dimension
        G = -self.residual[did].sum() # G before split
        H = did.shape[0]               # the number of residuals
        p_score = (G ** 2)/(H + r)     # score before the split
```

```python
        for k in range(d):
            GL = HL = 0.0

            # split x_feat using the best feature and the best
            # split point. The code below is inefficient because
            # it sorts x_feat every time it is split.
            # Future improvements are needed.
            x_feat = self.feature[did, k]

            # remove duplicates of x_feat and sort in ascending order
            x_uniq = np.unique(x_feat)
            s_point = [np.mean([x_uniq[i-1], x_uniq[i]]) \
                       for i in range(1, len(x_uniq))]
            l_bound = -np.inf   # lower left bound

            for j in s_point:
                # split x_feat into the left and the right node.
                left = did[np.where(np.logical_and(x_feat > \
                            l_bound, x_feat <= j))[0]]
                right = did[np.where(x_feat > j)[0]]

                # Calculate the scores after splitting
                GL -= self.residual[left].sum()
                HL += left.shape[0]
                GR = G - GL
                HR = H - HL

                # Calculate gain for this split
                gain = (GL**2)/(HL+r) + (GR**2)/(HR+r) - p_score

                # find the point where the gain is greatest.
                if gain > max_gain:
```

MX-AI

- Implementation of XGBoost regression from scratch

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0
$G=\sum_{i\in I} g_i,\ \ H=\sum_{i\in I} h_i,$
for k = 1 to d do:
  $G_L=0,\ \ H_L=0$
  for j in sorted(I, by xi,k) do:
    $G_L=G_L+g_i,\ \ H_L=H_L+h_i$
    $G_R=G-G_L,\ \ H_R=H-H_L$
    $gain=max\left(gain,\ \frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G^2}{H+\lambda}\right)$
  end
end
Output: Split with max gain ←

```
if gain > γ:
    split
else
    no split
```

$$g_i = -\left(y_i - \hat{y}^{(t-1)}\right)$$

$$G = \sum_i g_i$$

$$score = \frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda}$$

```
# [1] 2.1 Regularized Learning Objective
# Algorithm 1: Exact Greedy Algorithm for Split Finding
def node_split(self, did):
    r = self.reg_lambda
    max_gain = -np.inf
    d = self.feature.shape[1]      # feature dimension
    G = -self.residual[did].sum() # G before split
    H = did.shape[0]               # the number of residuals
    p_score = (G ** 2)/(H + r)     # score before splitting
```

```
    for k in range(d):
        GL = HL = 0.0

        # split x_feat using the best feature and the best
        # split point. The code below is inefficient because
        # it sorts x_feat every time it is split.
        # Future improvements are needed.
        x_feat = self.feature[did, k]

        # remove duplicates of x_feat and sort in ascending order
        x_uniq = np.unique(x_feat)
        s_point = [np.mean([x_uniq[i-1], x_uniq[i]]) \
                   for i in range(1, len(x_uniq))]
        l_bound = -np.inf   # lower left bound

        for j in s_point:
            # split x_feat into the left and the right node.
            left = did[np.where(np.logical_and(x_feat > \
                        l_bound, x_feat <= j))[0]]
            right = did[np.where(x_feat > j)[0]]

            # Calculate the scores after splitting
            GL -= self.residual[left].sum()
            HL += left.shape[0]
            GR = G - GL
            HR = H - HL

            # Calculate gain for this split
            gain = (GL**2)/(HL+r) + (GR**2)/(HR+r) - p_score

            # find the point where the gain is greatest.
            if gain > max_gain:
```

MX-AI

- Implementation of XGBoost regression from scratch

```
                max_gain = gain
                b_fid = k        # best feature id
                b_point = j      # best split point
            l_bound = j

        if max_gain >= self.prune_gamma:
            # split the node using the best split point
            x_feat = self.feature[did, b_fid]
            b_left = did[np.where(x_feat <= b_point)[0]]
            b_right = did[np.where(x_feat > b_point)[0]]
            return {'fid':b_fid, 'split_point':b_point,
                    'left':b_left, 'right':b_right}
        else:
            return  np.nan    # no split

    # Create a binary tree using recursion
    def recursive_split(self, node, curr_depth):
        left = node['left']
        right = node['right']

        # exit recursion
        if curr_depth >= self.max_depth:
            return

        # process recursion
        s = self.node_split(left)
        if isinstance(s, dict):   # split
            node['left'] = s
            self.recursive_split(node['left'], curr_depth+1)
        s = self.node_split(right)
```

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

```
        if isinstance(s, dict):   # split
            node['right'] = s
            self.recursive_split(node['right'], curr_depth+1)

    # Calculate the output value of a leaf node
    def output_value(self, did):
        r = self.residual[did]
        return np.sum(r) / (did.shape[0] + self.reg_lambda)

    # Calculate output values for every leaf node in a tree
    def output_leaf(self, d):
        if isinstance(d, dict):
            for key, value in d.items():
                if key == 'left' or key == 'right':
                    rtn = self.output_leaf(value)
                    if rtn[0] == 1:      # leaf node
                        d[key] = rtn[1]
            return 0, 0  # first 0 = non-leaf node
        else:            # leaf node
            return 1, self.output_value(d) # first 1 = leaf node

    # It creates a tree using the training data, and returns the
    # result of the tree. (x : feature data, y: residuals)
    def fit(self, x, y):
        self.feature = x
        self.residual = y
        self.base_score = y.mean()  # initial prediction

        root = self.node_split(np.arange(x.shape[0]))
```

MX-AI

- Implementation of XGBoost regression from scratch

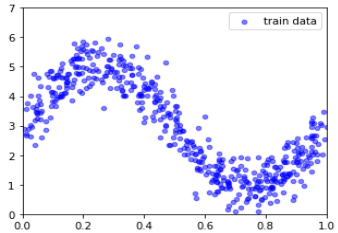
```
        if isinstance(root, dict):
            self.recursive_split(root, curr_depth=1)

        # tree result-1. Every leaf node has data indices.
        self.estimator1 = root

        # tree result-2. Every leaf node has its output values.
        if isinstance(self.estimator1, dict):
            self.estimator2 = copy.deepcopy(self.estimator1)
            self.output_leaf(self.estimator2)   # tree result-2
        return self.estimator2

    # Estimate the target value of a test data point.
    def x_predict(self, p, x):
        if x[p['fid']] <= p['split_point']:
            if isinstance(p['left'], dict): # recursion if not leaf.
                return self.x_predict(p['left'], x)
            else:                           # leaf
                return p['left']
        else:
            if isinstance(p['right'], dict):# not a leaf. recursion
                return self.x_predict(p['right'], x)
            else:         # return the value in the leaf, if leaf.
                return p['right']

    # Estimate the target values for all x_test points.
    def predict(self, x_test):
        p = self.estimator2    # predictor

        if isinstance(p, dict):
            y_pred = [self.x_predict(p, x) for x in x_test]
            return np.array(y_pred)
        else:
            return np.array([self.base_score] * x_test.shape[0])

# Build XGBoost regression tree
class MyXGBRegressor:
    def __init__(self, n_estimators=10, max_depth=3,
                 learning_rate=0.3, prune_gamma=0.0,
                 reg_lambda=0.0, base_score=0.5):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.eta = learning_rate        # learning rate
        self.prune_gamma = prune_gamma  # pruning constant
        self.reg_lambda = reg_lambda    # regularization constant
        self.base_score = base_score    # initial prediction

        self.estimator1 = dict()        # tree result-1
        self.estimator2 = dict()        # tree result-2
        self.models = []
        self.loss = []

    # The same as GBM algorithm. In XGBoost, only the node
    # splitting method changes.
    def fit(self, x, y):
        # step-1: Initialize model with a constant value.
        Fm = self.base_score
        self.models = []
        self.loss = []
        for m in range(self.n_estimators):
            # step-2 (A): Compute so-called pseudo-residuals
```

MX-AI

- Implementation of XGBoost regression from scratch

```
            residual = y - Fm

            # step-2 (B): Fit a regression tree to the residual
            model=MyXGBRegressionTree(max_depth=self.max_depth,
                                      reg_lambda=self.reg_lambda
                                    prune_gamma=self.prune_gamma)
            model.fit(x, residual)

            # step-2 (C): compute gamma (prediction)
            gamma = model.predict(x)

            # step-2 (D): Update the model
            Fm = Fm + self.eta * gamma

            # save tree models
            self.models.append(model)

            # Calculate the loss = mean squared error.
            self.loss.append(((y - Fm) ** 2).sum())

        return self.loss

    def predict(self, x_test):
        y_pred = np.zeros(shape=(x_test.shape[0],)) +\
                 self.base_score
        for model in self.models:
            y_pred += self.eta * model.predict(x_test)

        return y_pred
```

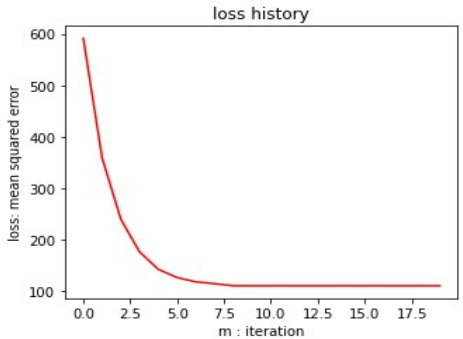
```
# [MXML-11-03] 1.XGBoost(regression).py
# Source: www.youtube.com/@meanxai
import numpy as np
from MyXGBoostRegressor import MyXGBRegressor
import matplotlib.pyplot as plt

# Generate the training data
def nonlinear_data(n, s):
   rtn_x, rtn_y = [], []
   for i in range(n):
       x = np.random.random()
       y = 2.0 * np.sin(2.0 * np.pi * x) + \
           np.random.normal(0.0, s) + 3.0
       rtn_x.append(x)
       rtn_y.append(y)
   return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)
x, y = nonlinear_data(n=500, s=0.5)

# Plot the training data and estimated curve
def plot_prediction(x, y, x_test, y_pred):
    plt.figure(figsize=(5,4))
    plt.scatter(x,y,c='blue',s=20,alpha=0.5,label='train data')
    plt.plot(x_test, y_pred, c='red', lw=2.0, label='prediction')
    plt.xlim(0, 1)
    plt.ylim(0, 7)
    plt.legend()
    plt.show()

y_mean = y.mean()      # initial prediction
n_depth = 3            # tree depth
n_tree = 20            # the number of trees
eta = 0.3              # learning rate
```



MX-AI

- Implementation of XGBoost regression from scratch

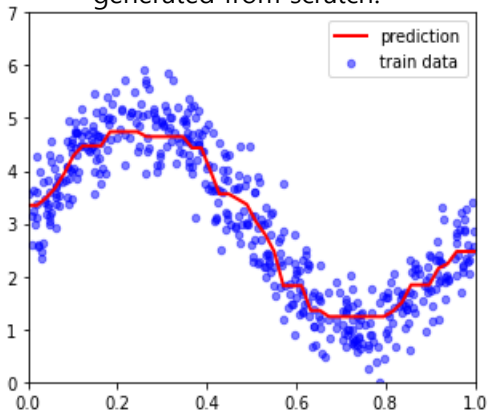
```
reg_lambda = 1.0       # regularization constant
prune_gamma = 2.0      # pruning constant
my_model = MyXGBRegressor(n_estimators=n_tree,
                          max_depth=n_depth,
                          learning_rate=eta,
                          prune_gamma = prune_gamma,
                          reg_lambda=reg_lambda,
                          base_score = y_mean)
loss = my_model.fit(x, y)
```



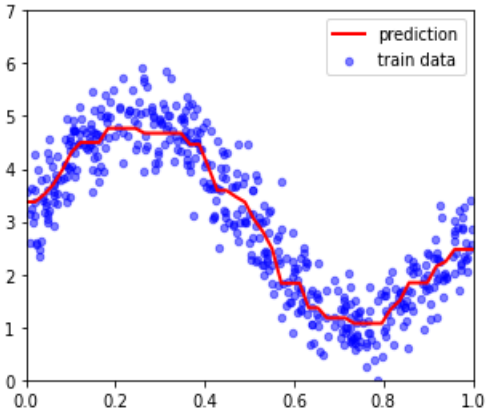

```
# Check the loss history
plt.figure(figsize=(5,4))
plt.plot(loss, c='red')
plt.xlabel('m : iteration')
plt.ylabel('loss: mean squared error')
plt.title('loss history')
plt.show()

x_test = np.linspace(0, 1, 50).reshape(-1, 1)
y_pred = my_model.predict(x_test)

# Plot the training data and estimated curve
plot_prediction(x, y, x_test, y_pred)
```

```
# Compare with the results from XGBRegressor library.
from xgboost import XGBRegressor

xg_model = XGBRegressor(n_estimators=n_tree,
                        max_depth=n_depth,
                        learning_rate=eta,
                        gamma=prune_gamma,
                        reg_lambda=reg_lambda,
                        base_score = y_mean)
xg_model.fit(x, y)
y_pred = xg_model.predict(x_test)  # predict the test data

# Plot the training data and estimated curve
plot_prediction(x, y, x_test, y_pred)
```

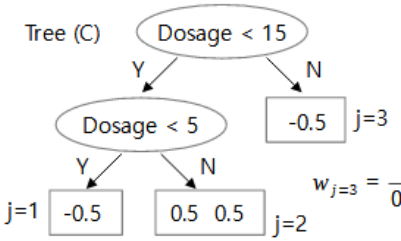
The result from our code generated from scratch.



The result from XGBoost library



	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
i=1	3	0	0.5	-0.5	0.42	-0.42
i=2	8	1	0.5	0.5	0.63	0.37
i=3	12	1	0.5	0.5	0.63	0.37
i=4	17	0	0.5	-0.5	0.42	-0.42



$$w_{j=3} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4$$

$$w_{j=1} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4 \qquad w_{j=2} = \frac{0.5+0.5}{2\times 0.5(1-0.5)+1} = 0.67$$

11. Extreme Gradient Boosting (XGBoost: Classification)

Part 4: Training and Prediction process

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Classification: Exact Greedy Algorithm for Split Finding - Training process

- Split the node with all split point candidates and select the most optimal split point.
- The source of this example is the YouTube "StatQuest with Josh Starmer", XGBppst Part 2.

regression: $\hat{y}_0 \rightarrow r_1 \rightarrow \hat{y}_1 \rightarrow r_2 \rightarrow \hat{y}_2 \rightarrow \ldots$

classification: $\hat{y}_0 \rightarrow F_0 \rightarrow r_1 \rightarrow F_1 \rightarrow \hat{y}_1 \rightarrow r_2 \rightarrow F_2 \rightarrow \hat{y}_2 \rightarrow \ldots$

	feature (x)	target (y)	initial prediction	residual (1)	
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	
i=1	3	0	0.5	-0.5	
i=2	8	1	0.5	0.5	
i=3	12	1	0.5	0.5	← (B)
i=4	17	0	0.5	-0.5	← (A)

1) Initialize the prediction of target y, and compute the logodds and the residual (1).

$$\hat{y}_i^{(0)} = 0.5 \qquad F_i^{(0)} = \log\left(\frac{\hat{y}_i^{(0)}}{1-\hat{y}_i^{(0)}}\right) = \log\left(\frac{0.5}{1-0.5}\right) = 0$$

$$r_1^{(1)} = y_1 - \hat{y}_1^{(0)} = 0-0.5 = -0.5$$

$$r_2^{(1)} = y_2 - \hat{y}_2^{(0)} = 1-0.5 = 0.5$$

$$r_3^{(1)} = 1-0.5 = 0.5 \qquad r_4^{(1)} = 0-0.5 = -0.5$$

2) Calculate the similarity score of the root node containing the residual (1).

$$score = \frac{\left(\sum_{i \in I_L} residual_i\right)^2}{\sum_{i \in I_L} h_i + \lambda}$$

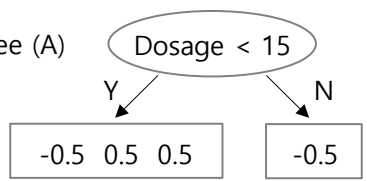
$$h_i = \hat{y}_i^{(0)}\left(1-\hat{y}_i^{(0)}\right) \qquad \lambda = 1$$

$$score = \frac{(-0.5+0.5+0.5-0.5)^2}{0.5(1-0.5)+0.5(1-0.5)+0.5(1-0.5)+0.5(1-0.5)+1}$$

$$= \frac{(-0.5+0.5+0.5-0.5)^2}{4\times 0.5(1-0.5)+1} = 0$$

3) For every split point of feature x, we split the root node and calculate the score of each leaf node. The larger the sum of the scores of the child nodes, the more likely it is to be a better split. In the example below, tree (A) is better than (B)

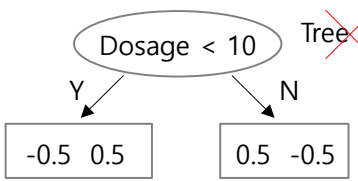
Tree (A)



$$score_{left} = \frac{(-0.5+0.5+0.5)^2}{3\times 0.5(1-0.5)+1} = 0.14$$

$$score_{right} = \frac{(-0.5)^2}{1\times 0.5(1-0.5)+1} = 0.2$$

~~Tree (B)~~



$$score_{left} = \frac{(-0.5+0.5)^2}{2\times 0.5(1-0.5)+1} = 0$$

$$score_{right} = \frac{(0.5-0.5)^2}{2\times 0.5(1-0.5)+1} = 0$$

MX-AI

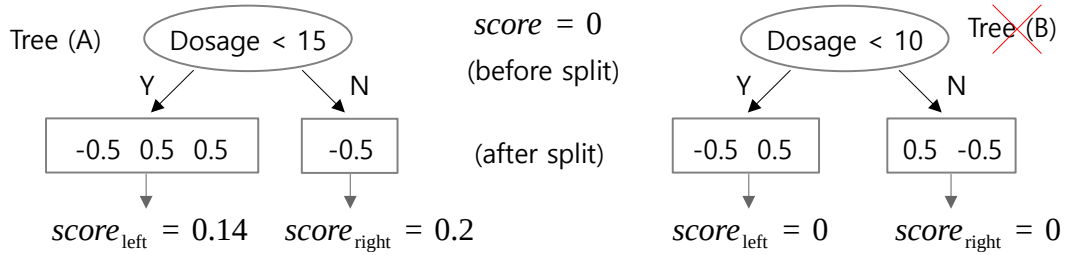
▪ Classification: Exact Greedy Algorithm for Split Finding - Training process

	feature (x)	target (y)	initial prediction	residual (1)	
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	
i=1	3	0	0.5	-0.5	← (C)
i=2	8	1	0.5	0.5	← (D)
i=3	12	1	0.5	0.5	
i=4	17	0	0.5	-0.5	

4) Calculate the gain using the similarity score before and after splitting.

Gain = sum of the scores after splitting
– the score before splitting
– γ (γ : pruning constant)

When the gamma, γ = 0.1, the gain of Tree(A) is greater than that of Tree(B). Among all candidate splits, the one with the highest gain is selected (Tree A). However, if gain ≤ 0, the node will not be split. This is because there is no benefit to split it. The larger the gamma γ, the less likely the node will be split, and vice versa. The γ is pruning constant.



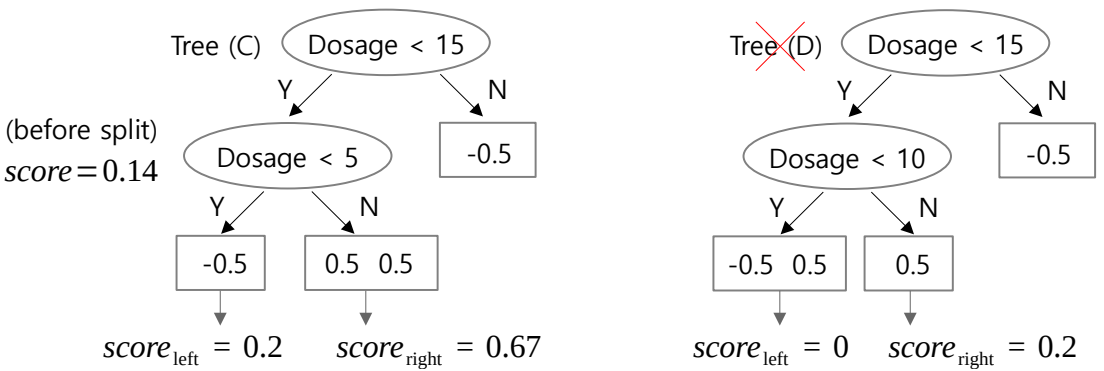
$score_{left} = 0.14$ $score_{right} = 0.2$

$gain = 0.14+0.2-0-0.1=0.24$

$score_{left} = 0$ $score_{right} = 0$

$gain = 0+0-0-0.1=-0.1$

5) Split the child nodes to a predefined depth in the same way. In the example below, tree (C) is better than (D)



$score_{left} = 0.2$ $score_{right} = 0.67$

$gain = 0.2+0.67-0.14-0.1=0.63$

$score_{left} = 0$ $score_{right} = 0.2$

$gain = 0+0.2-0.14-0.1=-0.04$

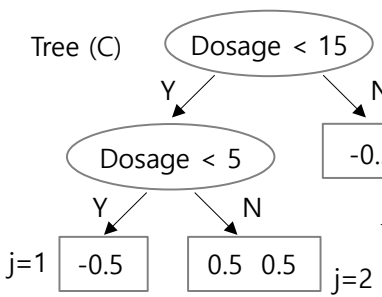
MX-AI

▪ Classification: Exact Greedy Algorithm for Split Finding - Training process

	feature (x)	target (y)	initial prediction	residual (1)		residual (2)
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
i=1	3	0	0.5	-0.5	0.42	-0.42
i=2	8	1	0.5	0.5	0.63	0.37
i=3	12	1	0.5	0.5	0.63	0.37
i=4	17	0	0.5	-0.5	0.42	-0.42

* The residuals are decreasing.

Tree (C)



$$w_{j=3} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4$$

$$w_{j=1} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4 \qquad w_{j=2} = \frac{0.5+0.5}{2\times 0.5(1-0.5)+1} = 0.67$$

6) Calculate the output values for the leaf node in the final tree.

$$output\ value(w_j) = \frac{\sum_{i\in I_j} r_i^{(1)}}{\sum_{i\in I_j} \hat{y}_i^{(t-1)}(1-\hat{y}_i^{(t-1)})+\lambda}$$

(j: leaf node number)

7) Calculate new predictions using the output values (w).

$$F_i^{(1)} = F^{(0)} + \eta\, w_{i\in I_j}$$

(η : learning rate)

$$F_1^{(1)} = 0+0.8(-0.4) = 0-0.32 = -0.32$$

$$\hat{y}_1^{(1)} = \frac{1}{1+\exp(0.32)} = 0.42$$

$$F_1^{(0)} = \log\left(\frac{y^{(0)}}{1-y^{(0)}}\right)$$

$$\hat{y}_i^{(1)} = \frac{1}{1+\exp(-F^{(1)})}$$

* Probability $y^{(0)}$ and log(odds) $F^{(0)}$ are interchangeable each other.

* $y^{(0)} \in \{0, 1\}$, $F^{(0)} \in (-\infty, +\infty)$

8) Calculate new residuals using the new predictions calculated in step 7.

$$r_1^{(2)} = y_1 - \hat{y}_1^{(1)} = 0-0.42 = -0.42 \qquad r_3^{(2)} = y_3 - \hat{y}_3^{(1)} = 1-0.63 = 0.37$$

$$r_2^{(2)} = y_2 - \hat{y}_2^{(1)} = 1-0.63 = 0.37 \qquad r_4^{(2)} = y_4 - \hat{y}_4^{(1)} = 0-0.42 = -0.42$$

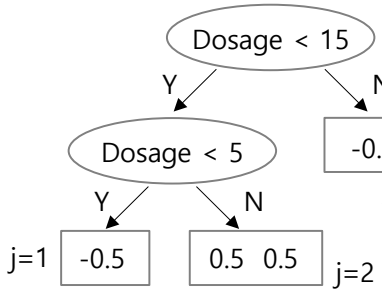
MX-AI

▪ Classification: Exact Greedy Algorithm for Split Finding - Training process

	feature (x)	target (y)	initial prediction	residual (1)		residual (2)
	Drug dosage	Drug effect	$\hat{y}_i^{(0)}$	$r_i^{(1)}$	$\hat{y}_i^{(1)}$	$r_i^{(2)}$
i=1	3	0	0.5	-0.5	0.42	-0.42
i=2	8	1	0.5	0.5	0.63	0.37
i=3	12	1	0.5	0.5	0.63	0.37
i=4	17	0	0.5	-0.5	0.42	-0.42

* The residuals are decreasing.

Tree (t=1)

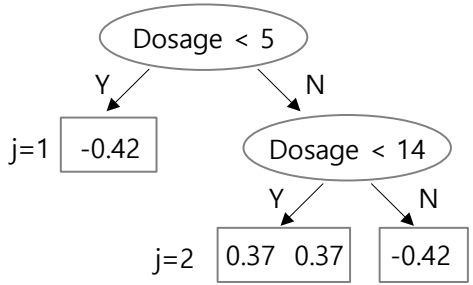


$$w_{j=3} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4$$

$$w_{j=1} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4 \qquad w_{j=2} = \frac{0.5+0.5}{2\times 0.5(1-0.5)+1} = 0.67$$

9) Repeat steps 2 through 5 to create another tree using the residual (2). This is the tree for the iteration round t=2.

Tree (t=2)



$$w_{j=3} = -0.338$$

$$w_{j=1} = \frac{-0.42}{0.42(1-0.42)+1} = -0.338 \qquad w_{j=2} = \frac{0.37+0.37}{2\times 0.63(1-0.63)+1} = 0.505$$

10) Calculate new predictions using the output values (w), and calculate new residuals using the new predictions.

- The residuals continue to decrease and the predicted y gets closer to the actual y.

11) Repeat this process until the residuals are sufficiently small.

MX-AI

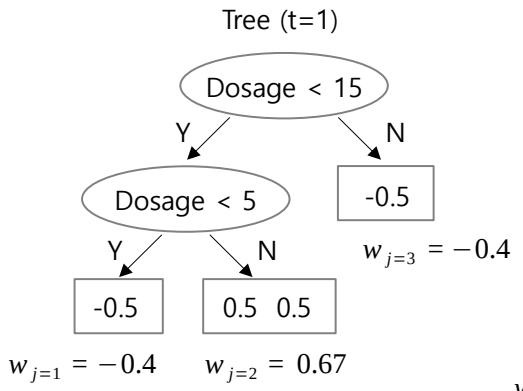
▪ Classification: Exact Greedy Algorithm for Split Finding - Prediction process

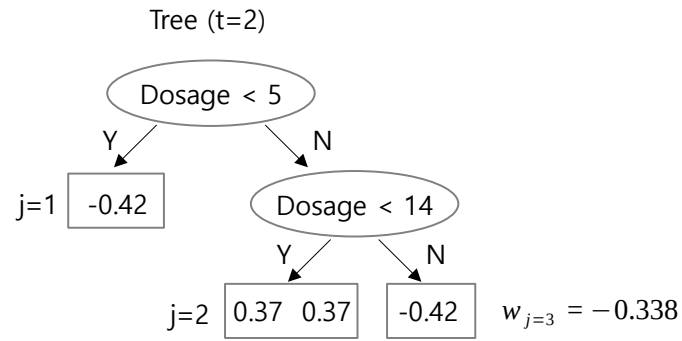
▪ We can predict the target class of test data using the tree models saved during the training process.

feature (x_test) target (y_pred)

Drug dosage	Drug effect
10	?

test data





$$w_{j=1} = \frac{-0.42}{0.42(1-0.42)+1} = -0.338 \qquad w_{j=2} = \frac{0.37+0.37}{2\times 0.63(1-0.63)+1} = 0.505$$

12) Predict the target class of the test data using the trees.

$F^* = F_0 + \eta * \text{tree(1).predict(x\_test)} + \eta * \text{tree(2).predict(x\_test)}$

$= 0 + 0.8 * 0.67 + 0.8 * 0.505 = 0.94$

▪ As a simple example, here we only calculated the residuals two times.

▪ In practice, the training process is repeated until the residuals are sufficiently small.

y_prob = 1 / (1 + exp(-0.94)) = 0.72 ← Since it is greater than 0.5, the target class is predicted as y_pred = 1.

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0

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

for k = 1 **to** d **do**:

  $G_L=0,\ \ H_L=0$

  for j in sorted(I, by $x_{i,k}$) **do**:

    $G_L=G_L+g_i,\ \ H_L=H_L+h_i$

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

    $gain=max\left(gain,\ \frac{G_L^2}{H_L+\gamma}+\frac{G_R^2}{H_R+\gamma}-\frac{G^2}{H+\gamma}\right)$

  end

end

Output: Split with max gain

11. Extreme Gradient Boosting

(XGBoost: Classification)

Part 5: Exact Greedy Algorithm analysis

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Classification: Regularized Learning Objective (Chapter 2.1 in the paper)

▪ For classification, binary cross entropy is used for the loss function instead of MSE.

$\hat{y}$: predict probability $\quad odds = \frac{\hat{y}}{1-\hat{y}} \quad \log(odds) = \log\left(\frac{\hat{y}}{1-\hat{y}}\right) \equiv F \quad \log\left(\frac{\hat{y}^{(t-1)}}{1-\hat{y}^{(t-1)}}\right) \equiv F^{(t-1)} \quad F^{(t-1)} + w_{x_i}^{(t)} = F^{(t)} \quad \frac{1}{1+\exp(-F^{(t)})} = \hat{y}^{(t)}$

$l(y, \hat{y}) = -y \cdot \log(\hat{y}) - (1-y) \cdot \log(1-\hat{y})$ ← Binary cross entropy $\quad l(y, F) = -yF + \log(1+\exp(F))$ ← Binary cross entropy using the logodds F.

▪ loss function of Tree (t) – the logodds, F is used.

$$L^{(t)} = l(y_i, F_i^{(t)}) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

$$L^{(t)} = l(y_i, F_i^{(t-1)} + w_{x_i}^{(t)}) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

$$F_i^{(t-1)} = F_i^{(0)} + w_{x_i}^{(1)} + w_{x_i}^{(2)} + \ldots + w_{x_i}^{(t-1)}$$

$$F_i^{(t-1)} = F_i^{(0)} + \eta w_{x_i}^{(1)} + \eta w_{x_i}^{(2)} + \ldots + \eta w_{x_i}^{(t-1)}$$

* shrinkage factor η (learning rate)

The output value of the node to which x_i belongs in Tree(t).

The output value of the j-th node in Tree (t)

$$L^{(t)} = \sum_{i=1}^{n} \underbrace{\left[-y_i\left(F_i^{(t-1)} + w_{x_i}^{(t)}\right) + \log\left(1+\exp\left(F_i^{(t-1)} + w_{x_i}^{(t)}\right)\right)\right]}_{\text{binary cross entropy}} + \underbrace{\gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^{(t)2}}_{\text{regularization term}}$$

The larger the T and the larger the w, the larger the penalty.

▪ Objective : $\min_{w_j^{(t)}} L^{(t)}$ $\qquad \frac{\partial L^{(t)}}{\partial w_j^{(t)}} = 0$

Find the output value $w^{(t)}_j$ that minimizes the loss $L^{(t)}$ of Tree (t).

Find $w^{(t)}_j$ from this equation. Once $w^{(t)}_j$ is determined, the optimal split point is also determined.

MX-AI

- Classification: Regularized Learning Objective (Chapter 2.1 in the paper)

$$L^{(t)} = \sum_{i=1}^{n}[-y_i(F_i^{(t-1)}+w_{x_i}^{(t)})+\log(1+\exp(F_i^{(t-1)}+w_{x_i}^{(t)}))] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^{(t)2}$$

$$L^{(t)} = l(y_i,\ F_i^{(t-1)}+w_{x_i}^{(t)}) + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$$

(Taylor series)

$$f(x+a, y+b)=f(x,y)+\frac{\partial f}{\partial x}a+\frac{\partial f}{\partial y}b+\frac{1}{2}(\frac{\partial^2 f}{\partial x^2}a^2+\frac{\partial^2 f}{\partial y^2}b^2+2\frac{\partial^2 f}{\partial x \partial y}ab)+\ldots$$

$$l(y_i,\ F_i^{(t-1)}+w_{x_i}^{(t)}) = l(y_i,\ F_i^{(t-1)})+\frac{\partial l}{\partial F_i^{(t-1)}}w_{x_i}^{(t)}+\frac{1}{2}\frac{\partial^2 l}{\partial F_i^{(t-1)2}}w_{x_i}^{(t)2}+\ldots$$

$$L^{(t)} \simeq \sum_{i=1}^{n}[\underline{-y_i F_i^{(t-1)}+\log(1+\exp(F_i^{(t-1)}))}]+g_i w_{x_i} + \frac{1}{2}h_i w_{x_i}^2] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$$

This term is independent of w_j. The superscript (t) of w is omitted.

$$\tilde{L}^{(t)} \simeq \sum_{i=1}^{n}(g_i w_{x_i} + \frac{1}{2}h_i w_{x_i}^2) + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$$

* This is equivalent to regression using MSE.

$$g_i = \frac{\partial l}{\partial F_i^{(t-1)}} = \frac{\partial[-y_i F_i^{(t-1)}+\log(1+\exp(F_i^{(t-1)}))]}{\partial F_i^{(t-1)}}$$

* This is equivalent to regression using MSE.

$$= -y_i + \frac{\exp(F_i^{(t-1)})}{1+\exp(F_i^{(t-1)})} = -y_i + \frac{1}{1+\exp(-F_i^{(t-1)})} = -(y_i - \hat{y}_i^{(t-1)})$$

= - residual

$$h_i = \frac{\partial^2 l}{\partial F_i^{(t-1)2}} = \frac{\partial}{\partial F_i^{(t-1)}}[-y_i + \frac{1}{1+\exp(-F_i^{(t-1)})}] = \frac{\exp(-F_i^{(t-1)})}{(1+\exp(-F_i^{(t-1)}))^2}$$

$$= \frac{\exp(-F_i^{(t-1)})}{1+\exp(-F_i^{(t-1)})}\cdot\frac{1}{1+\exp(-F_i^{(t-1)})}$$

$$= [\frac{1+\exp(-F_i^{(t-1)})}{1+\exp(-F_i^{(t-1)})} - \frac{1}{1+\exp(-F_i^{(t-1)})}]\cdot\frac{1}{1+\exp(-F_i^{(t-1)})}$$

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

When using MSE, it was 1, but when using binary cross entropy, it is this value.

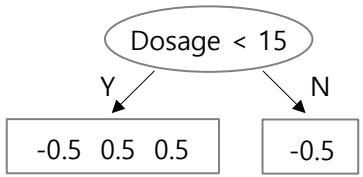
MX-AI

- Classification: Regularized Learning Objective (Chapter 2.1 in the paper)

  - Optimal output value of the leaf node j.

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda} = \frac{\sum_{i\in I_j} residual}{\sum_{i\in I_j} \hat{y}_i^{(t-1)}\left(1-\hat{y}_i^{(t-1)}\right)+\lambda}$$ ← equation (5)

Tree (A)



$$score_{\text{left}} = \frac{(-0.5+0.5+0.5)^2}{3\times 0.5(1-0.5)+1} = 0.14$$

$$score_{\text{right}} = \frac{(-0.5)^2}{0.5(1-0.5)+1} = 0.2$$

$$gain = \frac{1}{2}(0.14+0.2-0)-0.1 = 0.07$$

equation (7)

$$L_{split} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i+\lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i+\lambda}\right] - \gamma$$

gain

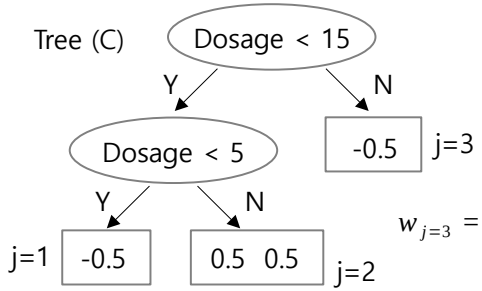
the score of left node

the score of right node

parent's score after splitting

The sum of scores after splitting

Tree (C)



$$w_{j=3} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4$$

$$w_{j=1} = \frac{-0.5}{0.5(1-0.5)+1} = -0.4 \qquad w_{j=2} = \frac{0.5+0.5}{2\times 0.5(1-0.5)+1} = 0.67$$

MX-AI

- Classification: Regularized Learning Objective (Chapter 2.1 in the paper)

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0

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

for k = 1 **to** d **do**:

$G_L=0, \quad H_L=0$

split point list

for j in sorted(I, by $x_{i,k}$) **do**:

$G_L=G_L+g_i, \quad H_L=H_L+h_i$

$G_R=G-G_L, \quad H_R=H-H_L$

$gain=max\left(gain, \frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G^2}{H+\lambda}\right)$

end

end

Output: Split with max gain

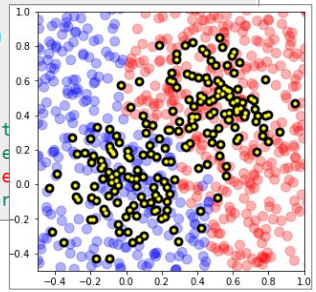
$$G_L=\sum_{i\in I_L} g_i, \quad H_L=\sum_{i\in I_L} h_i$$
$$G_R=\sum_{i\in I_R} g_i, \quad H_R=\sum_{i\in I_R} h_i$$

```
if gain > γ:
   split
else
   no split
```

← The γ of Equation (7) is considered here.

Source : Algorithm 1 in the paper (Some terms have been corrected.)

```python
# [MXML-11-06] MyXGBoostClassifier.py
# Upgraded version of CART in [MXML-02-07] for XGBoost
# Source: www.youtube.com/@meanxai
import numpy as np
from collections import Counter
import copy

# Implement "Exact Greedy Algorithm for Split Finding" presented
# in XGBoost paper. [1] Tianqi Chen et, al., 2016, XGBoost: A
# Scalable Tree Boosting
class MyXGBClassificationTree:
    def __init__(self, max_depth, reg_lambda, prune_gamma):
        self.max_depth = max_depth      # depth of the tree
        self.reg_lambda = reg_lambda    # regularization constant
        self.prune_gamma = prune_gamma  # pruning constant
        self.estimator1 = None          # tree result-1:
        self.estimator2 = None          # tree result-2:
        self.feature = None             # feature x
        self.residual = None            # residuals
        self.prev_yhat = None           # previous y_hat

    # [1] 2.1 Regularized Learning Objective
    #     Algorithm 1: Exact Greedy Algorithm
    def node_split(self, did):
        r = self.reg_lambda
        max_gain = -np.inf
        d = self.feature.shape[1]       # feat
        G = self.residual[did].sum()    # G be
        H = (self.prev_yhat[did]*(1.- self.pre
        p_score = (G ** 2) / (H + r)    # scor
```



11. Extreme Gradient Boosting (XGBoost: Classification)

Part 6: Implementation of XGBoost Classification from scratch

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

- Implementation of XGBoost Classification from scratch

```python
# [MXML-11-06] MyXGBoostClassifier.py
# Upgraded version of CART in [MXML-02-07] for XGBoost
# Source: www.youtube.com/@meanxai
import numpy as np
from collections import Counter
import copy

# Implement "Exact Greedy Algorithm for Split Finding" presented
# in XGBoost paper. [1] Tianqi Chen et, al., 2016, XGBoost: A
# Scalable Tree Boosting
class MyXGBClassificationTree:
    def __init__(self, max_depth, reg_lambda, prune_gamma):
        self.max_depth = max_depth      # depth of the tree
        self.reg_lambda = reg_lambda    # regularization constant
        self.prune_gamma = prune_gamma  # pruning constant
        self.estimator1 = None          # tree result-1
        self.estimator2 = None          # tree result-2
        self.feature = None             # feature x
        self.residual = None            # residuals
        self.prev_yhat = None           # previous y_hat

    # [1] 2.1 Regularized Learning Objective
    #     Algorithm 1: Exact Greedy Algorithm for Split Finding
    def node_split(self, did):
        r = self.reg_lambda
        max_gain = -np.inf
        d = self.feature.shape[1]       # feature dimension
        G = self.residual[did].sum()    # G before split
        H = (self.prev_yhat[did]*(1.- self.prev_yhat[did])).sum()
        p_score = (G ** 2) / (H + r)    # score before the split
```

```python
        for k in range(d):
            GL = HL = 0.0

            # split x_feat using the best feature and the best
            # split point. The code below is inefficient because
            # it sorts x_feat every time it is split.
            # Future improvements are needed.
            x_feat = self.feature[did, k]

            # remove duplicates of x_feat and sort in ascending order
            x_uniq = np.unique(x_feat)
            s_point = [np.mean([x_uniq[i-1], x_uniq[i]]) \
                       for i in range(1, len(x_uniq))]
            l_bound = -np.inf             # lower left bound
            for j in s_point:
                # split x_feat into the left and the right node.
                left = did[np.where(np.logical_and(x_feat >\
                                    l_bound, x_feat <= j))[0]]
                right = did[np.where(x_feat > j)[0]]

                # Calculate the scores after splitting
                GL += self.residual[left].sum()
                HL += (self.prev_yhat[left] * \
                       (1.-self.prev_yhat[left])).sum()
                GR = G - GL
                HR = H - HL

                # Calculate gain for this split
                gain = (GL**2)/(HL+r) + (GR**2)/(HR+r) - p_score

                # find the point where the gain is greatest.
                if gain > max_gain:
                    max_gain = gain
```

MX-AI

- Implementation of XGBoost Classification from scratch

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension

gain ← 0

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

for k = 1 to d do:

$G_L=0,\ \ H_L=0$

for j in sorted(I, by xi,k) do:

$G_L=G_L+g_i,\ \ H_L=H_L+h_i$

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

$gain=max\left(gain,\ \frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G^2}{H+\lambda}\right)$

end

end

Output: Split with max gain ←

```
if gain > γ:
    split
else
    no split
```

$$g_i = -\left(y_i-\hat{y}^{(t-1)}\right)$$

$$h_i = \hat{y}_i^{(t-1)}\left(1-\hat{y}_i^{(t-1)}\right)$$

$$G = \sum_i g_i$$

$$score=\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda}$$

```
# [1] 2.1 Regularized Learning Objective
#     Algorithm 1: Exact Greedy Algorithm for Split Finding
def node_split(self, did):
    r = self.reg_lambda
    max_gain = -np.inf
    d = self.feature.shape[1]        # feature dimension
    G = self.residual[did].sum()     # G before split
    H = (self.prev_yhat[did]*(1.- self.prev_yhat[did])).sum()
    p_score = (G ** 2) / (H + r)     # score before the split
```

```
    for k in range(d):
        GL = HL = 0.0

        # split x_feat using the best feature and the best
        # split point. The code below is inefficient because
        # it sorts x_feat every time it is split.
        # Future improvements are needed.
        x_feat = self.feature[did, k]

        # remove duplicates of x_feat and sort in ascending order
        x_uniq = np.unique(x_feat)
        s_point = [np.mean([x_uniq[i-1], x_uniq[i]]) \
                   for i in range(1, len(x_uniq))]
        l_bound = -np.inf              # lower left bound
        for j in s_point:
            # split x_feat into the left and the right node.
            left = did[np.where(np.logical_and(x_feat >\
                                l_bound, x_feat <= j))[0]]
            right = did[np.where(x_feat > j)[0]]

            # Calculate the scores after splitting
            GL += self.residual[left].sum()
            HL += (self.prev_yhat[left] * \
                   (1.-self.prev_yhat[left])).sum()
            GR = G - GL
            HR = H - HL

            # Calculate gain for this split
            gain = (GL**2)/(HL+r) + (GR**2)/(HR+r) - p_score

            # find the point where the gain is greatest.
            if gain > max_gain:
                max_gain = gain
```

MX-AI

- Implementation of XGBoost Classification from scratch

```
                b_fid = k          # best feature id
                b_point = j        # best split point
            l_bound = j

        if max_gain >= self.prune_gamma:
            # split the node using the best split point
            x_feat = self.feature[did, b_fid]
            b_left = did[np.where(x_feat <= b_point)[0]]
            b_right = did[np.where(x_feat > b_point)[0]]
            return {'fid':b_fid, 'split_point':b_point,\
                    'gain':max_gain, \
                    'left':b_left, 'right':b_right}
        else:
            return  np.nan    # no split

    # Create a binary tree using recursion
    def recursive_split(self, node, curr_depth):
        left = node['left']
        right = node['right']

        # exit recursion
        if curr_depth >= self.max_depth: return

        # process recursion
        s = self.node_split(left)
        if isinstance(s, dict):   # split
            node['left'] = s
            self.recursive_split(node['left'], curr_depth+1)

        s = self.node_split(right)
        if isinstance(s, dict):   # split
```

$$g_i = -\left(y_i - \hat{y}^{(t-1)}\right)$$

$$h_i = \hat{y}_i^{(t-1)}\left(1 - \hat{y}_i^{(t-1)}\right)$$

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

```
            node['right'] = s
            self.recursive_split(node['right'], curr_depth+1)

    # Calculate the output value of a leaf node
    def output_value(self, did):
        r = self.residual[did]
        H = (self.prev_yhat[did]*(1.-self.prev_yhat[did])).sum()
        return np.sum(r) / (H + self.reg_lambda)

    # Calculate output values for every leaf node in a tree
    def output_leaf(self, d):
        if isinstance(d, dict):
            for key, value in d.items():
                if key == 'left' or key == 'right':
                    rtn = self.output_leaf(value)
                    if rtn[0] == 1:      # leaf node
                        d[key] = rtn[1]
            return 0, 0  # first 0 = non-leaf node
        else:            # leaf node
            return 1, self.output_value(d) # first 1 = leaf node

    # It creates a tree using the training data, and returns the
    # result of the tree. (x : feature data, y: residuals)
    def fit(self, x, y, prev_yhat):
        self.feature = x
        self.residual = y
        self.prev_yhat = prev_yhat  # previous y_hat.

        root = self.node_split(np.arange(x.shape[0]))
        if isinstance(root, dict):
            self.recursive_split(root, curr_depth=1)
```

MX-AI

- Implementation of XGBoost Classification from scratch

```
        # tree result-1. leaf node has data indices.
        self.estimator1 = root

        # tree result-2. leaf node has its output values.
        if isinstance(self.estimator1, dict):
            self.estimator2 = copy.deepcopy(self.estimator1)
            self.output_leaf(self.estimator2) # tree result-2
        return self.estimator2

    # Estimate the output value of a test data point.
    def x_predict(self, p, x):
        if x[p['fid']] <= p['split_point']:
            if isinstance(p['left'], dict):# recursion if not leaf
                return self.x_predict(p['left'], x)
            else:                           # leaf
                return p['left']
        else:
            if isinstance(p['right'], dict): # not a leaf
                return self.x_predict(p['right'], x)
            else:        # return the value in the leaf, if leaf.
                return p['right']

    # Estimate the output values for all x_test points.
    def predict(self, x_test):
        p = self.estimator2    # predictor

        if isinstance(p, dict):
            y_pred = [self.x_predict(p, x) for x in x_test]
            return np.array(y_pred)
        else:
            return self.prev_yhat * x_test.shape[0]
```

```
# Build XGBoost regression tree
class MyXGBClassifier:
    def __init__(self,
                 n_estimators=10, max_depth=3, learning_rate=0.3,
                 prune_gamma=0.0, reg_lambda=0.0, base_score=0.5):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.eta = learning_rate        # learning rate
        self.prune_gamma = prune_gamma  # pruning constant
        self.reg_lambda = reg_lambda    # regularization constant
        self.base_score = base_score    # initial prediction

        self.estimator1 = dict()        # tree result-1
        self.estimator2 = dict()        # tree result-2
        self.models = []
        self.loss = []

    # convert the log(odds) into probability
    def F2P(self, x):
        return 1. / (1. + np.exp(-x))

    # The same as GBM algorithm. In XGBoost, only the node
    # splitting method changes.
    def fit(self, x, y):
        # step-1: Initialize model with a constant value.
        F0 = np.log(self.base_score / (1. - self.base_score))
        Fm = np.repeat(F0, x.shape[0])
        y_hat = self.F2P(Fm)
```

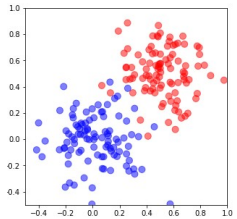
MX-AI

- Implementation of XGBoost Classification from scratch

```
        self.models, self.loss = [], []
        for m in range(self.n_estimators):
            # step-2 (A): Compute so-called pseudo-residuals
            residual = y - y_hat

            # step-2 (B): Fit a classification tree
            model = MyXGBClassificationTree(
                                max_depth = self.max_depth,
                                reg_lambda = self.reg_lambda,
                                prune_gamma = self.prune_gamma)
            model.fit(x, residual, y_hat) # y_hat: previous y_hat

            # step-2 (C): compute gamma (prediction)
            gamma = model.predict(x)

            # step-2 (D): Update the model
            Fm = Fm + self.eta * gamma
            y_hat = self.F2P(Fm)

            # save tree models
            self.models.append(model)

            # Calculate the loss = mean squared error.
            self.loss.append(-(y * np.log(y_hat + 1e-8) + \
                      (1.- y) * np.log(1.- y_hat + 1e-8)).sum())
        return self.loss

    def predict(self, x_test, proba=False):
        Fm = np.zeros(shape=(x_test.shape[0],)) + self.base_score
        for model in self.models:
            Fm += self.eta * model.predict(x_test)
```

```
        y_prob = self.F2P(Fm)

        if proba:
            return y_prob   # return probability
        else:
            y_pred = (y_prob > 0.5).astype('uint8')
            return y_pred   # return label
```

```
# [MXML-11-06] 2.XGBoost(classification).py
import numpy as np
from sklearn.datasets import make_blobs
from MyXGBoostClassifier import MyXGBClassifier
import matplotlib.pyplot as plt

# Generate the training data
x, y = make_blobs(n_samples=200, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.18, center_box=(-1., 1.))

# Plot the training and test data, and the prediction result
def plot_prediction(x, y, x_test, y_pred):
    plt.figure(figsize=(5,5))
    color = ['red' if a == 1 else 'blue' for a in y_pred]
    plt.scatter(x_test[:, 0], x_test[:, 1], s=100, c=color,
                alpha=0.3)
    plt.scatter(x[:, 0], x[:, 1], s=80, c='black')
    plt.scatter(x[:, 0], x[:, 1], s=10, c='yellow')
    plt.xlim(-0.5, 1.0)
    plt.ylim(-0.5, 1.0)
    plt.show()
```



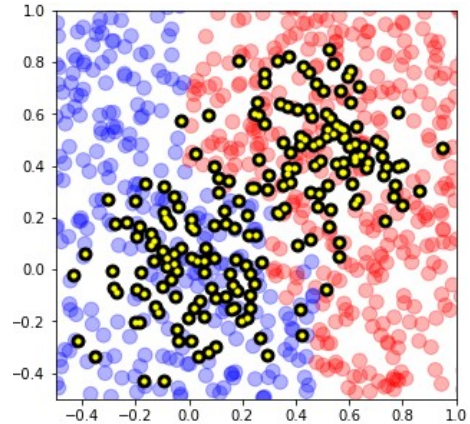
MX-AI

- Implementation of XGBoost Classification from scratch

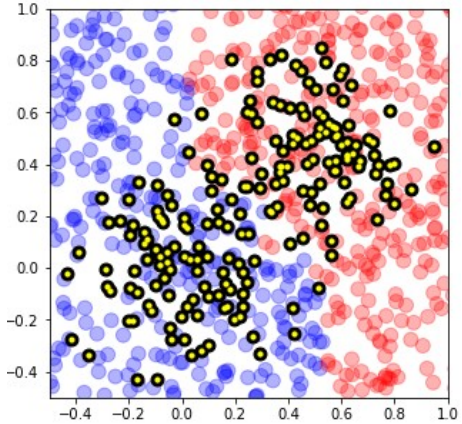
```
y_init = 0.5            # initial prediction
n_depth = 3             # tree depth
n_tree = 20             # the number of trees
eta = 0.3               # learning rate
reg_lambda = 0.1        # regularization constant
prune_gamma = 0.01      # pruning constant

my_model = MyXGBClassifier(
             n_estimators=n_tree,
             max_depth=n_depth,
             learning_rate=eta,
             prune_gamma=prune_gamma,
             reg_lambda=reg_lambda,
             base_score=y_init)
loss = my_model.fit(x, y)

# Check the loss history
plt.figure(figsize=(5,4))
plt.plot(loss, c='red')
plt.xlabel('m : iteration')
plt.ylabel('loss: binary cross entropy')
plt.title('loss history')
plt.show()

# Create the test data, and predict the target class
x_test=np.random.uniform(-0.5,1.5,(1000, 2))
y_pred = my_model.predict(x_test)

# Plot the training and test data, and the prediction result
plot_prediction(x, y, x_test, y_pred)
```


```
# Compare with the results from XGBRegressor library.
from xgboost import XGBClassifier
xg_model = XGBClassifier(objective='binary:logistic',
                         tree_method = 'exact',
                         n_estimators=n_tree,
                         max_depth=n_depth,
                         learning_rate=eta,
                         gamma=prune_gamma,
                         reg_lambda=reg_lambda,
                         base_score=y_init)
xg_model.fit(x, y)

# Predict the target class of the test data and visualize the result
y_pred = xg_model.predict(x_test)
plot_prediction(x, y, x_test, y_pred)
```

Result from our code generated from scratch.



Result from XGBoost library



MX-AI

Multiclass classification

- The multiclass classification of XGBoost is the same as GBM. Only the GBM model needs to be changed to XGBoost.
- Convert multiclass y to one-hot encoding, and perform binary classification for each encoding.
- For example, if you have three types of classes (y = [0, 1, 2]), you can train them using three separate binary classification models.
- At each iteration (m=1, 2, 3…M), as many regression trees are created as there are classes. the number of models=(M * n_classes)
- The prediction step uses the largest softmax probability to predict the class.
- For the detailed process, please refer to the code in the GBM video, [MXML-10-7].

multiclass data | one-hot encoding

no	x1	x2	y	p(y=0)	p(y=1)	p(y=2)
1	0.8	1.2	0	1	0	0
2	9.8	2.0	2	0	0	1
3	3.2	3.5	0	1	0	0
4	4.8	5.2	1	0	1	0
…	…	…	…	…	…	…
N	9.2	2.5	2	0	0	1

x1	x2	p(y=0)
0.8	1.2	1
9.8	2.0	0
3.2	3.5	1
4.8	5.2	0
…	…	…
9.2	2.5	0

XGBoost model-1

$F_m^{(0)},\ \hat{y}_m^{(0)}$

x1	x2	p(y=1)
0.8	1.2	0
9.8	2.0	0
3.2	3.5	0
4.8	5.2	1
…	…	…
9.2	2.5	0

XGBoost model-1

$F_m^{(1)},\ \hat{y}_m^{(1)}$

x1	x2	p(y=2)
0.8	1.2	0
9.8	2.0	1
3.2	3.5	0
4.8	5.2	0
…	…	…
9.2	2.5	1

XGBoost model-1

$F_m^{(2)},\ \hat{y}_m^{(2)}$

At each iteration, the predicted probability of each model is converted to softmax and the residuals are calculated.

Algorithm 2: Approximate Algorithm for Split Finding

for k = 1 **to** d **do**: # d: feature dimension

Propose $S_k = \{ s_{k,1}, s_{k,2}, \dots s_{k,m} \}$ by percentile on feature k.

Propose can be done per terr (global), or per split (local).

end

for k = 1 **to** d **do**:

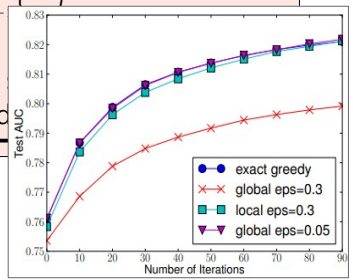
$$G_{kv} = \sum_{j \in \{ j | s_{k,v} \geq x_{j,k} > s_{k,v-1} \}} g_j$$

$$H_{kv} = \sum_{j \in \{ j | s_{k,v} \geq x_{j,k} > s_{k,v-1} \}} h_j$$

end

Follow same step as in previous

max score only among proposed



11. Extreme Gradient Boosting (XGBoost)

Part 7: Approximate Algorithm for Split Finding

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Approximate Algorithm (Chapter 3.2 in the paper)

- Exact Greedy Algorithm (EGA) uses all split point candidates in the feature vector to split nodes, so it can accurately find the optimal split point, but it takes a long time if there is a lot of data. For a large amount of data, Approximate Algorithm (AA) can be used to find approximate optimal split points.

Steps:

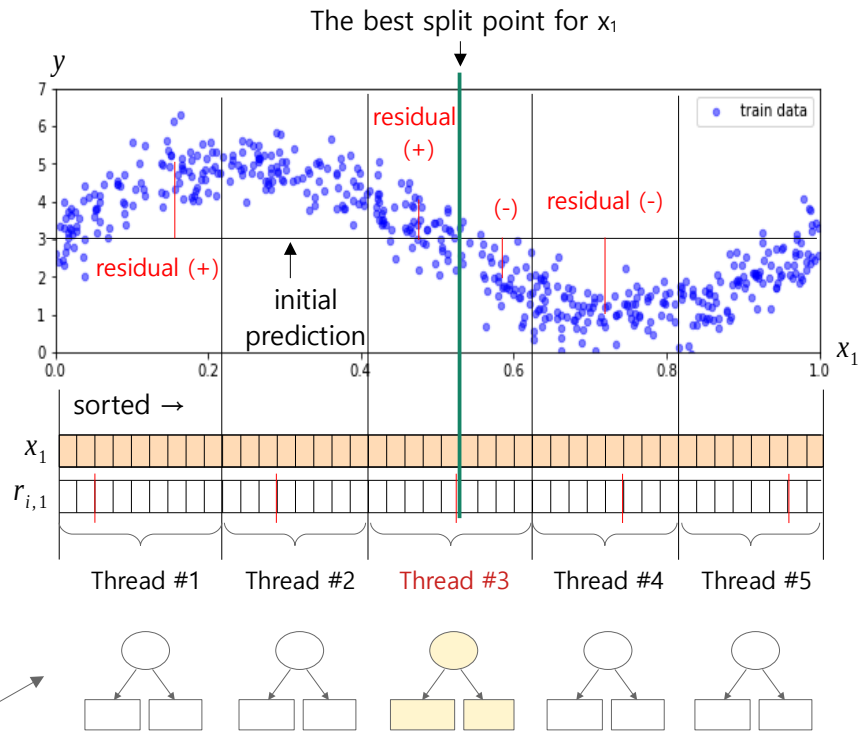
1. Calculate the residual using previous predictions. In the first round, the initial prediction is used for previous predictions.
2. for x in [x_1, x_2, x_3]:
3. sx = Sort x in ascending order.
4. Divide sx by the interval 1/ε. If ε=0.2, divide sx into 5 buckets.
5. Assign sx to each bucket. Make the number of data points in each bucket equal. (percentile).
6. Assign one thread to each bucket. Each thread calculates the gain of its bucket and finds the best split point with the best gain. Each thread runs in parallel. (multi threading).
7. Among the best gains found across the five threads, choose the largest one. This is the optimal split point for the entire sx.
8. Among all the features [x_1, x_2, x_3], select the feature and split point with the largest gain, and split the parent node using the finally selected feature and split point.

* Since the residuals in the Thread #3 bucket cancel each other out, the optimal split point gain for this bucket is the largest. Select the optimal split point found by thread #3 as that of the entire x1.

feature x — target — residual

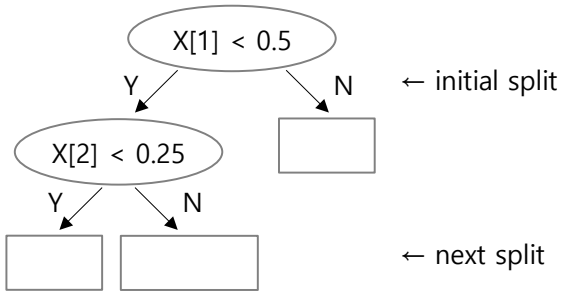
x_1	x_2	x_3	y	$r_i^{(1)}$
0.72	0.67	0.96	2.57	-0.43
0.05	0.78	0.15	3.84	0.84
0.15	0.96	0.55	4.56	1.56
0.38	0.14	0.78	4.06	1.06

⋮



MX-AI

▪ Approximate Algorithm (Chapter 3.2 in the paper) – Global variant vs. local variant



Algorithm 2: Approximate Algorithm for Split Finding

for k = 1 **to** d **do**: # d: feature dimension
  Propose $S_k = \{ s_{k,1}, s_{k,2}, \dots s_{k,m} \}$ by percentile on feature k.
  Propose can be done per terr (global), or per split (local).
end
for k = 1 **to** d **do**:
  $G_{kv} = \sum_{j \in \{ j | s_{k,v} \geq x_{j,k} > s_{k,v-1} \}} g_j$
  $H_{kv} = \sum_{j \in \{ j | s_{k,v} \geq x_{j,k} > s_{k,v-1} \}} h_j$
end

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

Source : Algorithm 2 in the paper (Symbols d, m modified.)

If ε=0.01, the initial split divides the data into 100 parts, and uses 100 threads to find the optimal split point. Dividing the data at the middle bucket, we have 51 (or 50) buckets on the left and 50 (or 51) buckets on the right. (One is a divided bucket.)

▪ Global variant

In the next split, the same process above is performed for the 51 (or 50) buckets on the left and right. As the tree gets deeper, the number of buckets gets smaller, so it is necessary to set ε sufficiently small.

▪ Local variant

In the next split, ε=0.01 is newly applied to each of the left and right sides, and the same process as above is performed for 100 buckets for each side. Even as the tree gets deeper, the number of buckets remains constant at 100, so ε can be set somewhat larger.

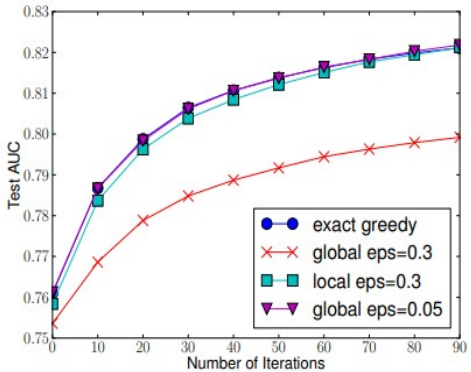


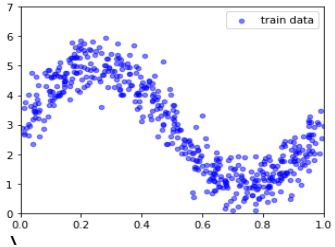
Figure 3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

* Source : Figure 3 of the paper

MX-AI

- Check out the effectiveness of the Approximate Algorithm

```python
# [MXML-11-07] 3.approximation(1).py
import numpy as np
from MyXGBoostRegressor import MyXGBRegressor
import time

# Create training data
def nonlinear_data(n, s):
   rtn_x, rtn_y = [], []
   for i in range(n):
       x = np.random.random()
       y = 2.0 * np.sin(2.0 * np.pi * x) + \
           np.random.normal(0.0, s) + 3.0
       rtn_x.append(x)
       rtn_y.append(y)
   return np.array(rtn_x).reshape(-1,1), np.array(rtn_y)
x, y = nonlinear_data(n=50000, s=0.5)

# 1. Exact Greedy Algorithm (EGA)
start_time = time.time()
my_model = MyXGBRegressor(n_estimators=1, max_depth=1,
                          base_score=y.mean())

my_model.fit(x, y)
e = my_model.models[0].estimator2

print('\nExact greedy algorithm:')
print('split point =', np.round(e['split_point'], 3))
print('gain =', np.round(e['gain'], 3))
print('running time =’ {:.2f} seconds'.\
       format(time.time() - start_time))
```




```python
# 2.Approximate Algorithm (AA).
from multiprocessing.pool import Pool

def find_split_point(x, y):
    my_model = MyXGBRegressor(n_estimators=1,
                              max_depth=1,
                              base_score=y.mean())
    my_model.fit(x, y)
    e = my_model.models[0].estimator2
    return [e['split_point'], e['gain']]

# Divide the data into five parts and allocate 20% of the data to
# each part. (ε=0.2)
c_point = np.percentile(x, [20, 40, 60, 80, 100])

# maps the data into buckets split by c_point
l_bound = -np.inf
x_block, y_block = [], []
for p in c_point:
    idx = np.where(np.logical_and(x > l_bound, x <= p))[0]
    x_block.append(x[idx])
    y_block.append(y[idx])
    l_bound = p

start_time = time.time()
mp = Pool(5)  # 5 processes
args = [[ax, ay] for ax, ay in zip(x_block, y_block)]
ret = mp.starmap_async(find_split_point, args)
mp.close()
mp.join()
```

MX-AI

- Check out the effectiveness of the Approximate Algorithm

```
print('\nApproximate Algorithm:')
print('split_points =', np.array(ret.get())[:, 0].round(3))
print('gain =', np.array(ret.get())[:, 1].round(2))
print('running time = {:.2f} seconds'\
      .format(time.time() - start_time))
print('number of data in blocks =', [len(a) for a in x_block])
```

```
Results:

Exact greedy algorithm:
split point = 0.502
gain = 81767.464
running time = 6.20 seconds

Approximate Algorithm:
split_points = [0.082 0.339 0.506 0.657 0.907]
gain = [2506.2   501.91 3764.17  443.95 2502.24]
running time = 0.27 seconds
number of data in blocks = [10000, 10000, 10000, 10000, 10000]
```

* All five buckets have the same number of data points.

MyXGBRegressor is a class implemented with EGA. To implement this properly, you need to implement the Approximate Algorithm inside the MyXGBRegressor.

```python
# 1. Exact Greedy Algorithm
start_time = time.time()
model = XGBClassifier(n_estimators = TREES,
                max_depth = DEPTH,
                learning_rate = ETA,     # η
                gamma = GAMMA,           # γ for pruning
                reg_lambda = LAMB,       # λ for regularization
                base_score = 0.5,        # initial prediction value
                tree_method = 'exact')  # exact greedy algorithm

model.fit(x_train, y_train)
acc = model.score(x_test, y_test)

print('\nExact greedy algorithm:')
print('Accuracy =', np.round(acc, 3))
print('running time = {:.2f} seconds'.format(time.time() -
start_time))

# 2.Weighted Quantile Sketch
start_time = time.time()
model = XGBClassifier(n_estimators = TREES,
                max_depth = DEPTH,
                learning_rate = ETA,     # η
                gamma = GAMMA,           # γ for pruning
                reg_lambda = LAMB,       # λ for regularization
                base_score = 0.5,        # initial prediction value
                max_bin = int(1/EPS),    # sketch_eps is replaced
                                         # by max_bin
                tree_method = 'approx') # weighted quantile sketch
```

11. Extreme Gradient Boosting (XGBoost)

Part 8: Weighted Quantile Sketch for Split Finding

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Split Finding Algorithm : Weighted Quantile Sketch (Chapter 3.3 in the paper)

- In the Approximate Algorithm, each bucket has the same number of data points. For regression using MSE, h=1, ∑ h=n. If the data is split into 5 buckets, n/5 data points fall into each bucket. Each data point has the same weight of 1. h can be considered a weight. ← quantile sketch
- However, classification using binary cross-entropy has h=p(1-p), so the weight, h, is not constant for all data points. In this case, the data is split into m buckets such that the sum of h in each bucket is equal. ← weighted quantile sketch that can handle weighted data.

- dataset: k-th feature values and h

$$D_k = \{(x_{1,k}, h_1), (x_{2,k}, h_2), \ldots (x_{n,k}, h_n)\}$$

- Our goal is to find candidate split points such that ∑ h is the same in each bucket.

$$s_k = \{s_{k,1}, s_{k,2}, s_{k,3}, \ldots s_{k,m}\}$$

- Define a normalized rank function:

$$r_k(z) = \frac{\sum_{(x,h) \in D_k, x<z} h}{\sum_{(x,h) \in D_k} h}$$ ← equation (8)

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k,1} = \min_i x_{i,k}, \quad s_{k,m} = \max_i x_{i,k}$$

equation (9)

After 3 iterations (t=3), we want to find the optimal split point for the 4th iteration round (t=4).

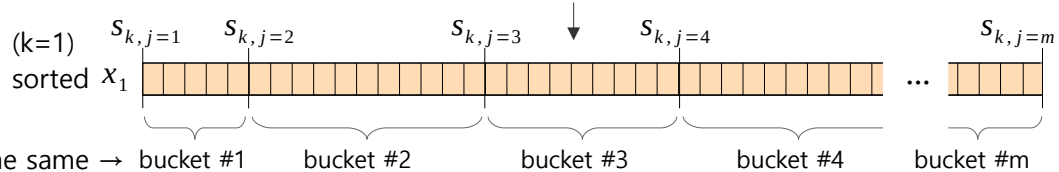
$$h_i^{(4)} = \hat{y}_i^{(3)}(1 - \hat{y}_i^{(3)})$$

x1	x2	y	$\hat{y}_i^{(3)}$	$h_i^{(4)}$	$r_i^{(4)}$
0.28	0.07	0	0.37	0.23	-0.37
-0.08	-0.21	0	0.28	0.20	-0.28
0.17	-0.28	0	0.28	0.20	-0.28
0.44	0.46	1	0.88	0.11	0.12
⋮					

If $\hat{y}_i^{(3)}$ is close to 1 or 0, h is small, and if $\hat{y}_i^{(3)}$ is close to 0.5, h is large.

ex: | 0.6 – 0.8 | < 0.2: Make sure the sum of h does not exceed 20% of the total.

$$|r_k(s_{k,3}) - r_k(s_{k,4})| < \epsilon$$



MX-AI

- Split Finding Algorithm : Weighted Quantile Sketch (Chapter 3.3 in the paper)

  - Weighted squared loss

$$\tilde{L}^{(t)} \simeq \sum_{i=1}^{n}\left(g_i w_{x_i} + \frac{1}{2}h_i w_{x_i}^2\right) + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2 \qquad \leftarrow \text{equation (3)}$$

$$= \frac{1}{2}\sum_{i=1}^{n} h_i\left(w_{x_i}^2 + 2\frac{g_i}{h_i}w_{x_i}\right) + \Omega_w$$

$$= \frac{1}{2}\sum_{i=1}^{n} h_i\left(w_{x_i} + \frac{g_i}{h_i}\right)^2 - \frac{1}{2}\sum_{i=1}^{n} h_i\left(\frac{g_i}{h_i}\right)^2 + \Omega_w \qquad \leftarrow \text{the next equation of equation (9)}$$

$$= \sum_{i=1}^{n}\frac{1}{2}h_i\left(w_{x_i} - \left(-\frac{g_i}{h_i}\right)\right)^2 + \Omega_w + C$$

$$L_{WLR} = \sum_{i=1}^{n}\left[w_i\left(\hat{y}_i - y_i\right)^2\right]$$

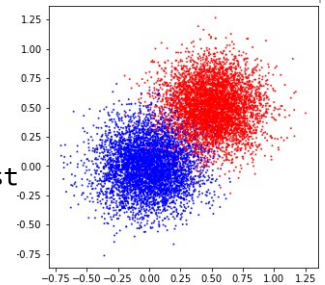
* It is exactly weighted squared loss with labels ($-g_i/h_i$) and weights h_i.

* WLR stands for Weighted Linear Regression.

MX-AI

▪ Implementation of the Weighted Quantile Sketch

```python
# [MXML-11-08] approximation(2).py
# Tianqi Chen et, al., 2016, XGBoost: A Scalable Tree Boosting
# System
# 3. SPLIT FINDING ALGORITHMS
# 3.3 Weighted Quantile Sketch

import numpy as np
from sklearn.datasets import make_blobs
from xgboost import XGBClassifier
from sklearn.model_selection import train_test
import time

# Create a simple training dataset
x, y = make_blobs(n_samples=500000, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.2, center_box=(-1., 1.))

# Generate training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)

TREES = 200      # the number of trees
DEPTH = 5        # the depth of tree
ETA = 0.1        # learning rate, eta
LAMB = 1.0       # regularization constant, λ
GAMMA = 0.1      # pruning constant, γ
EPS = 0.03       # epsilon for approximate algorithm
                 # and weighted quantile sketch
```




```python
# XGBClassifier parameters:
#
# tree_method:
#
# https://xgboost.readthedocs.io/en/stable/parameter.html
# auto: Same as the hist tree method.
# 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.
#
# https://xgboost.readthedocs.io/en/latest/treemethod.html
# approx tree method: An approximation tree method described in
# reference paper. It runs sketching before building each tree using
# all the rows (rows belonging to the root). Hessian is used as
# weights during sketch. The algorithm can be accessed by setting
# tree_method to approx.
```

$$H^{(t)} = \sum_i h_i^{(t)} = \sum_i \hat{y}_i^{(t-1)}(1-\hat{y}_i^{(t-1)})$$

```python
# max_bin:
#
# https://github.com/dmlc/xgboost/issues/8063
# Also, the parameter sketch_eps is replaced by max_bin for aligning
# with hist, the old default for max_bin translated from sketch_eps
# was around 63 while the rewritten one is 256, which means the new
# implementation builds larger histogram.
```

$$L_{split} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i+\lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i+\lambda}\right] - \gamma$$

MX-AI

- Implementation of the Weighted Quantile Sketch

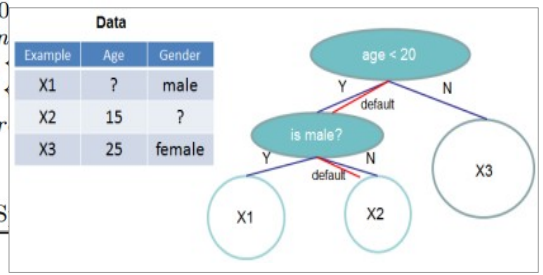
```
# 1. Exact Greedy Algorithm
start_time = time.time()
model = XGBClassifier(n_estimators = TREES,
                max_depth = DEPTH,
                learning_rate = ETA,     # η
                gamma = GAMMA,           # γ for pruning
                reg_lambda = LAMB,       # λ for regularization
                base_score = 0.5,        # initial prediction value
                tree_method = 'exact')   # exact greedy algorithm

model.fit(x_train, y_train)
acc = model.score(x_test, y_test)

print('\nExact greedy algorithm:')
print('Accuracy =', np.round(acc, 3))
print('running time = {:.2f} seconds'.format(time.time() -
start_time))

# 2.Weighted Quantile Sketch
start_time = time.time()
model = XGBClassifier(n_estimators = TREES,
                max_depth = DEPTH,
                learning_rate = ETA,     # η
                gamma = GAMMA,           # γ for pruning
                reg_lambda = LAMB,       # λ for regularization
                base_score = 0.5,        # initial prediction value
                max_bin = int(1/EPS),    # sketch_eps is replaced
                                         # by max_bin
                tree_method = 'approx') # weighted quantile sketch
```

```
model.fit(x_train, y_train)
acc = model.score(x_test, y_test)

print('\nWeighted Quantile Sketch:')
print('Accuracy =', np.round(acc, 3))
print('running time = {:.2f} seconds'.format(time.time() -
start_time))
```

```
Exact greedy algorithm:
Accuracy = 0.961
running time = 9.22 seconds

Weighted Quantile Sketch:
Accuracy = 0.96
running time = 1.99 seconds


Exact greedy algorithm:
Accuracy = 0.962
running time = 8.77 seconds

Weighted Quantile Sketch:
Accuracy = 0.962
running time = 1.80 seconds
```

Algorithm 3: Sparsity-aware Split Finding

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



11. Extreme Gradient Boosting (XGBoost)

- **Part 9: Sparsity-aware Split Finding**

- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

▪ Split Finding Algorithm : Sparsity-aware Split Finding (Chapter 3.4 in the paper)

- In many real-world problems, it is quite common for the input x to be sparse. There are multiple possible causes for sparsity:
  1) presence of missing values in the data;
  2) frequent zero entries in the statistics;
  3) artifacts of feature engineering such as one-hot encoding.

  It is important to make the algorithm aware of the sparsity pattern in the data.

  → Sparsity-aware Split Finding
- Below is part of Santander customer dataset. You can see there are a lot of zeros.

ID	x1	x2	x3	x4	x5	...
1	2	23	0	0	0	0
3	2	34	0	0	0	0
4	2	23	0	0	0	0
8	2	37	0	195	195	0
10	2	39	0	0	0	0
13	2	23	0	0	0	0
23	2	25	0	0	0	0
25	2	42	0	0	0	0
26	2	26	0	0	0	0
29	2	51	0	0	0	0
31	2	43	0	0	0	0
32	2	33	600	1086.48	1952.91	0

Algorithm 3: Sparsity-aware Split Finding

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

- When splitting a node, only non-missing (or non-zeros) values are used to find the optimal split point.
- Calculate the gain when assigning missing values to the left, and calculate the gain when assigning missing values to the right.
- Assign the missing values to the side with the larger gain. This is the default direction. The optimal default directions are learned from the data.
- When predicting test data, missing values are assigned to the default direction.

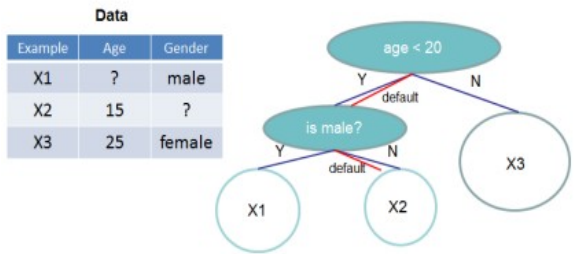


Figure 4: Tree structure with default directions. An example will be classified into the default direction when the feature needed for the split is missing.

* Source : Algorithm 3, Figure 4 of the paper

MX-AI

▪ Santander Customer Satisfaction dataset

- This is the data requested by Santander Bank for the Kaggle contest.
- The number of data points is 76,020 and the number of features is 370. the goal is to predict the TARGET: customer satisfaction.
- TARGET = 1 means the customer is dissatisfied, 0 means the customer is satisfied.
- The TARGET is mostly 0 (more than 90%), some 1. This data is very sparse. That means it contains a lot of zeros.

ID	var3	var15	imp_ent_var16_ult1	imp_op_var39_comer_ult1	imp_op_var39_comer_ult3	imp_op_var40_comer_ult1	imp_op_var40_comer_ult3	imp_op_var40_efect_ult1	imp_op_var40_efect_ult3	imp_op_var40_ult1	imp_op_var41_comer_ult1	imp_op_var41_comer_ult3	imp_op_var41_efect_ult1	imp_op_var41_efect_ult3	imp_op_var41_ult1	....	TARGET
1	2	23	0	0	0	0	0	0	0	0	0	0	0	0	0		0
3	2	34	0	0	0	0	0	0	0	0	0	0	0	0	0		0
4	2	23	0	0	0	0	0	0	0	0	0	0	0	0	0		0
8	2	37	0	195	195	0	0	0	0	0	195	195	0	0	195		0
10	2	39	0	0	0	0	0	0	0	0	0	0	0	0	0		0
13	2	23	0	0	0	0	0	0	0	0	0	0	0	0	0		0
14	2	27	0	0	0	0	0	0	0	0	0	0	0	0	0		0
18	2	26	0	0	0	0	0	0	0	0	0	0	0	0	0		0
20	2	45	0	0	0	0	0	0	0	0	0	0	0	0	0		0
23	2	25	0	0	0	0	0	0	0	0	0	0	0	0	0		0
25	2	42	0	0	0	0	0	0	0	0	0	0	0	0	0		0
26	2	26	0	0	0	0	0	0	0	0	0	0	0	0	0		0
29	2	51	0	0	0	0	0	0	0	0	0	0	0	0	0		0
31	2	43	0	0	0	0	0	0	0	0	0	0	0	0	0		0
32	2	33	600	1086.48	1952.91	0	0	0	0	0	1086.48	1952.91	360	750	1446.48		0
34	2	30	0	0	0	0	0	0	0	0	0	0	0	0	0		0

MX-AI

▪ Santander Customer Satisfaction

```python
# [MXML-11-09] 7.santander.py
import pandas as pd
import numpy as np
from xgboost import XGBClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Read the Santander Customer Satisfaction Dataset.
# df.shape = (76020, 371)
df = pd.read_csv("data/santander.csv", encoding='latin-1')

# Replace the values of the 'var3' feature containing -99999999 with 2
# and drop the 'ID' feature.
df['var3'].replace(-999999, 2, inplace=True)
df.drop('ID', axis = 1, inplace=True)

# Separate features and label from the dataset
# and generate training and test data.
x = df.drop('TARGET', axis=1)
y = df['TARGET']
x_train, x_test, y_train, y_test = train_test_split(x, y)

TREES = 200  # the number of trees
DEPTH = 5    # the depth of tree
ETA = 0.1    # learning rate, eta
LAMB = 1.0   # regularization constant
GAMMA = 0.1  # pruning constant
EPS = 0.03   # epsilon for approximate and weighted quantile sketch
```

```python
# Create an XGBoost classification model and fit it to the
# training data
model = XGBClassifier(n_estimators = TREES,
            max_depth = DEPTH,
            learning_rate = ETA,     # η
            gamma = GAMMA,           # γ for pruning
            reg_lambda = LAMB,       # λ for regularization
            base_score = 0.5,        # initial prediction
            missing = 0.0,           # for sparsity-aware
            subsample = 0.5,         # Subsample ratio of the
                                     # training instance
            colsample_bynode = 0.5, # Subsample ratio of
                                     # columns for each split
            max_bin = int(1/EPS),    # sketch_eps is replaced
                                     # by max_bin
            tree_method = 'approx') # weighted quantile sketch

model.fit(x_train, y_train)

# Predict the test data and measure the performance with AUC.
y_prob = model.predict_proba(x_test)[:, 1]
auc = roc_auc_score(y_test, y_prob)
print('\nROC-AUC = {:.4f}'.format(auc))
```

```
Result:
ROC-AUC = 0.8340
```