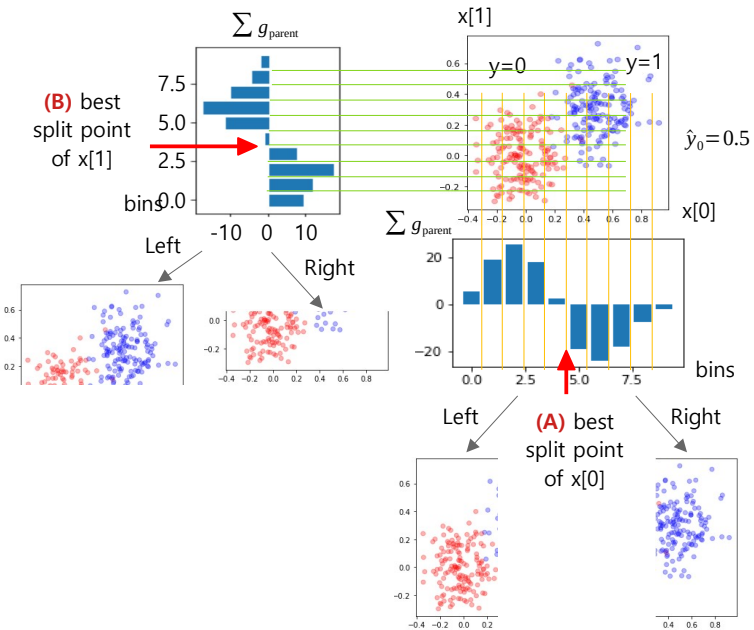


12. Light GBM

Part 1: Histogram-based split finding

- You should watch the XGBoost videos first because it covers only the parts added to XGBoost.
- This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

[MXML-12-01]
- **1. Histogram-based Algorithm**
  - 1-1. Overview
  - 1-2. Histogram-based split finding
  - 1-3. Parallel processing
  - 1-4. Simple implementation

[MXML-12-02]
- **2. Gradient-based One-Side Sampling (GOSS)**
  - 2-1. Algorithm analysis
  - 2-2. Implementation of GOSS from scratch
  - 2-3. Implementation of LGBM with GOSS using LGBMClassifier

[MXML-12-03]
- **3. Exclusive Feature Bundling (EFB)**
  - 3-1. Greedy Bundling Algorithm analysis
  - 3-2. Implementation of Greedy Bundling

[MXML-12-04]
- 3-3. Merge Exclusive Features Algorithm analysis
- 3-4. Implementation of Merge Exclusive Features

[MXML-12-05]
- 3-5. Implementation of EFB
  - Greedy Bundling + Merge Exclusive Features
- **4. Depth-wise vs. Leaf-wise tree growth**
- **5. LightGBM library**
  - 5-1. Santander Customer Transaction Prediction

MX-AI

- LightGBM : Overview
  - LightGBM was proposed by Microsoft's Guolin Ke and others in 2017 to handle large amounts of data more effectively than XGBoost.
  - To process large amounts of data, features such as Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) were proposed.

LightGBM: A Highly Efficient Gradient Boosting Decision Tree

Guolin Ke[1], Qi Meng[2], Thomas Finley[3], Taifeng Wang[1]
Wei Chen[1], Weidong Ma[1], Qiwei Ye[1], Tie-Yan Liu[1]
[1]Microsoft Research [2]Peking University [3]Microsoft Redmond
[1]{guolin.ke, taifengw, wche, weima, qiwye, tie-yan.liu}@microsoft.com;
[2]qimeng13@pku.edu.cn; [3]tfinely@microsoft.com;

ABSTRACT

Gradient Boosting Decision Tree (GBDT) is a popular machine learning algorithm and has quite a few effective implementations such as XGBoost and pGBRT. Although many engineering optimizations have been adopted in these implementations, the efficiency and scalability are still unsatisfactory when the feature dimension is high and data size is large. A major reason is that for each feature, they need to scan all the data instances to estimate the information gain of all possible split points, which is very time consuming. To tackle this problem, we propose two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). With GOSS, we exclude a significant proportion of data instances with small gradients, and only use the rest to estimate the information gain. We prove that, since the data instances with larger gradients play a more important role in the computation of information gain, GOSS can obtain quite accurate estimation of the information gain with a much smaller data size. With EFB, we bundle mutually exclusive features (i.e., they rarely take nonzero values simultaneously), to reduce the number of features. We prove that finding the optimal bundling of exclusive features is NP-hard, but a greedy algorithm can achieve quite good approximation ratio (and thus can effectively reduce the number of features without hurting the accuracy of split point determination by much). We call our new GBDT implementation with GOSS and EFB LightGBM. Our experiments on multiple public datasets show that, LightGBM speeds up the training process of conventional GBDT by up to over 20 times while achieving almost the same accuracy.

An Experimental Evaluation of Large Scale GBDT Systems

Fangcheng Fu Jiawei Jiang Yingxia Shao Bin Cui
Department of Computer Science and Technology & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
Center for Data Science & National Engineering Laboratory for Big Data Analysis and Applications, Peking University
†Beijing University of Posts and Telecommunications #Tencent Inc.
{ccchengff,bin.cui}@pku.edu.cn shaoyx@bupt.edu.cn jeremyjiang@tencent.com

ABSTRACT

Gradient boosting decision tree (GBDT) is a widely-used machine learning algorithm in both data analytic competitions and real-world industrial applications. Further, driven by the rapid increase in data volume, efforts have been made to train GBDT in a distributed setting to support large-scale workloads. However, we find it surprising that the existing systems manage the training dataset in different ways, but none of them have studied the impact of data management. To that end, this paper aims to study the pros and cons of different data management methods regarding the performance of distributed GBDT. We first introduce a quadrant categorization of data management policies based on data partitioning and data storage. Then we conduct an in-depth systematic analysis and summarize the advantageous scenarios of the quadrants. Based on the analysis, we further propose a novel distributed GBDT system named Vero, which adopts the unexplored composition of vertical partitioning and row-store and suits for many large-scale cases. To validate our analysis empirically, we implement different quadrants in the same code base and compare them under extensive workloads, and finally compare Vero with other state-of-the-art systems over a wide range of datasets. Our theoretical and experimental results provide a guideline on choosing a proper data management policy for a given workload.

MX-AI

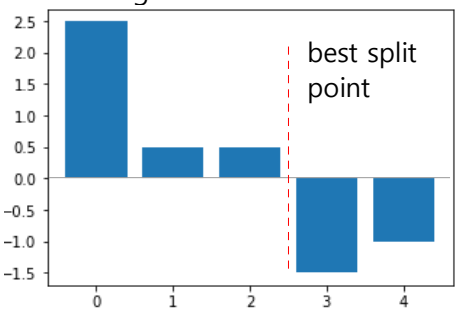
▪ Histogram-based split finding - Example

```
    data id = [0,    1,    2,    3,    4,    5,    6,    7,    8,    9,    10,   11,   12,   13,   14,   15,   16,   17,   18,   19]
 feature (x) = [0.41, 0.22, 0.7 , 0.06, 0.48, 0.8 , 0.05, 0.72, 0.34, 0.15, 0.6 , 0.31, 0.01, 0.02, 0.29, 0.55, 0.53, 0.22, 0.4 , 0.98]
  target (y) = [0,    0,    1,    0,    0,    1,    0,    1,    0,    0,    1,    1,    0,    0,    1,    1,    1,    0,    0,    1]
          y0 = 0.5  ← initial prediction
-residual (g) = [0.5,  0.5, -0.5,  0.5,  0.5, -0.5,  0.5, -0.5,  0.5,  0.5, -0.5, -0.5,  0.5,  0.5, -0.5, -0.5, -0.5,  0.5,  0.5, -0.5]
           h = 0.5 * (1 - 0.5) = 0.25
```

* frequency distribution table

구간	bin	id	Σg
0 ≤ x < 0.2	0	3,6,9,12,13	0.5+0.5+0.5+0.5+0.5 = 2.5
0.2 ≤ x < 0.4	1	1,8,11,14,17	0.5+0.5-0.5-0.5+0.5 = 0.5
0.4 ≤ x < 0.6	2	0,4,15,16,18	0.5+0.5-0.5-0.5+0.5 = 0.5
0.6 ≤ x < 0.8	3	2,7,10	-0.5-0.5-0.5 = -1.5
0.8 ≤ x < 1.0	4	5,19	-0.5-0.5 = -1.0

* histogram



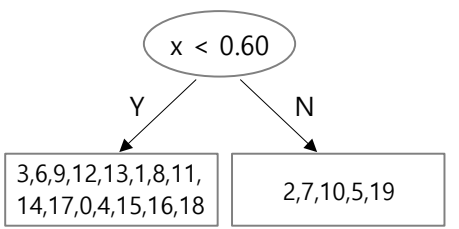

```
g_sum, bins = np.histogram(x, 5, weights=g)

plt.figure(figsize=(5, 4))
plt.bar(np.arange(0, g_sum.shape[0]), g_sum)
plt.show()
```

* score table (λ = 0)

candidate split points	Σg (left)	Σg (right)	Score (left)	Score (right)	Score (left + right)
0.2	2.5	-1.5	2.5^2 / (0.25*5) = 5.0	$(-1.5)^2$ / (0.25*15) = 0.6	5.6
0.4	3.0	-2.0	3.0^2 / (0.25*10) = 3.6	$(-2.0)^2$ / (0.25*10) = 1.6	5.2
0.6	3.5	-2.5	3.5^2 / (0.25*15) = 3.27	$(-2.5)^2$ / (0.25*5) = 5.0	9.94
0.8	2.0	-1.0	2.0^2 / (0.25*18) = 0.89	$(-1.0)^2$ / (0.25*2) = 2.0	2.89

← best split point

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



MX-AI

▪ Histogram-based split finding - Algorithm

- If you create a histogram with 10 bins for each feature, as in the example on the right, only nine split attempts are required.

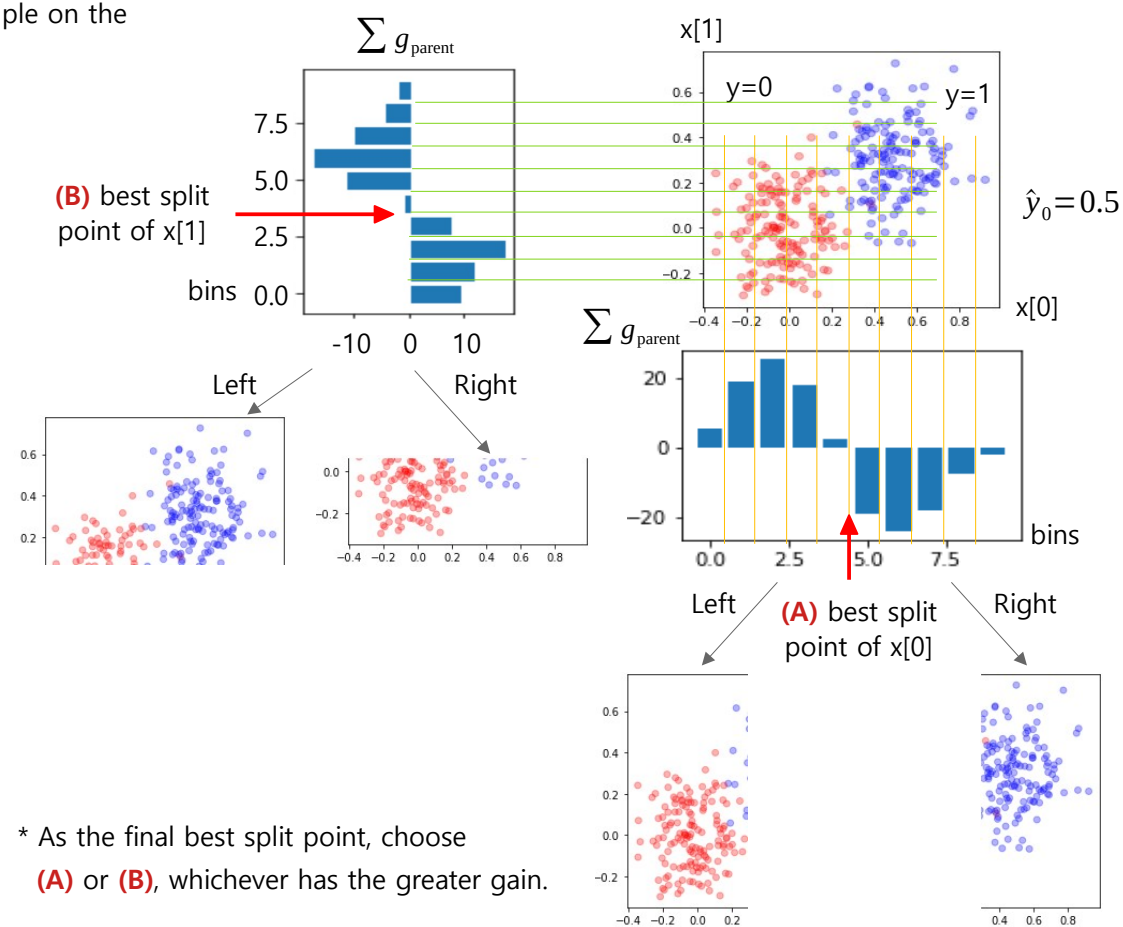
Algorithm 1: Histogram-based Algorithm

Input: I: training data, d: max depth
Input: m: feature dimension
$nodeSet \leftarrow \{0\}$ ▷ tree nodes in current level
$rowSet \leftarrow \{\{0, 1, 2, ...\}\}$ ▷ data indices in tree nodes
for $i = 1$ ***to*** d **do**
  for $node$ ***in*** $nodeSet$ **do**
    usedRows $\leftarrow rowSet[node]$
    for $k = 1$ ***to*** m **do**
      $H \leftarrow$ new Histogram()
      ▷ Build histogram
      for j ***in*** $usedRows$ **do**
        bin $\leftarrow I$.f[k][j].bin
        H[bin].y $\leftarrow H$[bin].y + I.y[j]
        H[bin].n $\leftarrow H$[bin].n + 1
      Find the best split on histogram H.
      ...
  Update $rowSet$ and $nodeSet$ according to the best split points.
  ...

y means g.
accumulate g.
$g = -(y - \hat{y})$

accumulate h.
$mse \rightarrow 1$
$bce \rightarrow \hat{y}(1-\hat{y})$

* Source : Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree



* As the final best split point, choose **(A)** or **(B)**, whichever has the greater gain.

MX-AI

▪ Histogram-based split finding - Parallel processing

- Histogram-based algorithm takes a long time to generate a histogram each time a node is split, but the time can be reduced through parallel processing. It is advantageous when processing large amounts of data.

* Source: Fangcheng Fu, et, al., 2019, An Experimental Evaluation of Large Scale GBDT Systems. (Figure 3, 4: Illustration of data partitioning and storage)

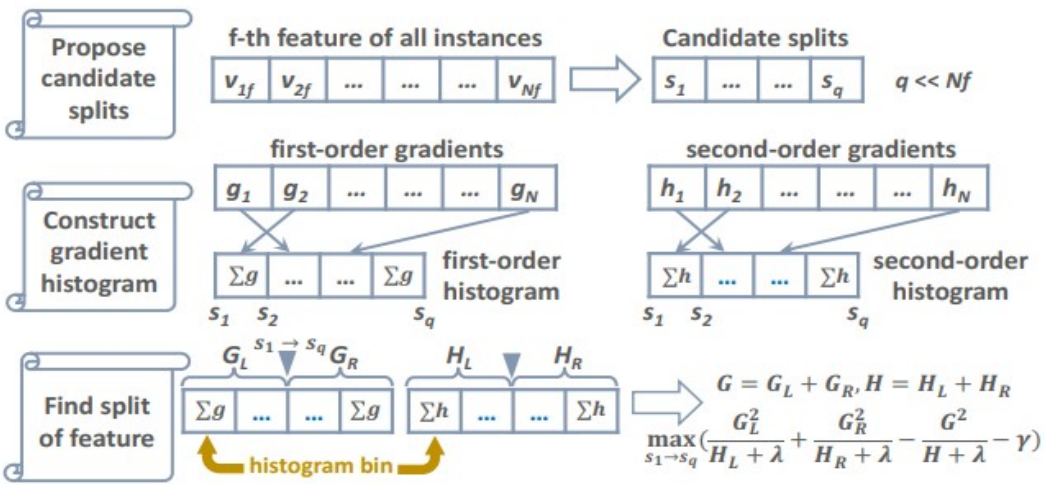


Figure 3: Histogram-based split finding for one feature

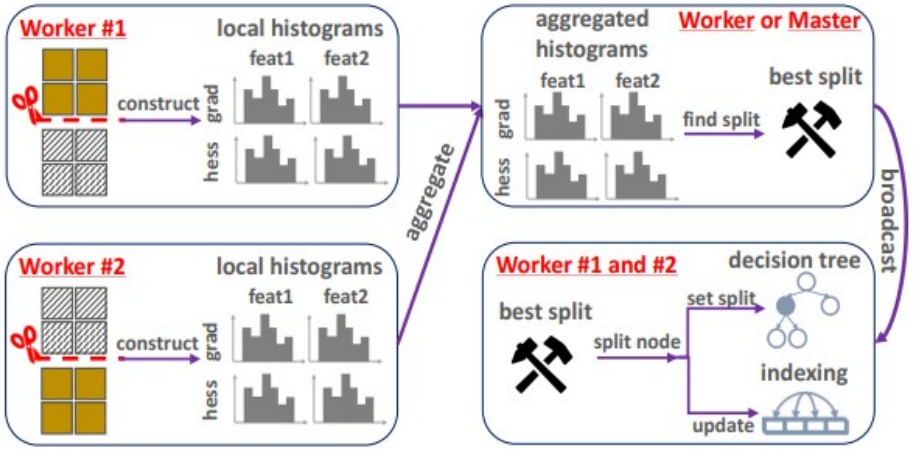


Figure 4 (a): Horizontal partitioning. Workers construct local histograms for all features and aggregate into global ones.

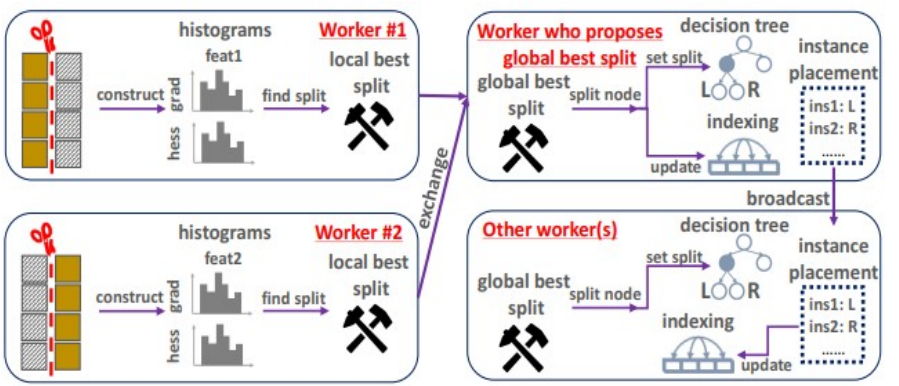
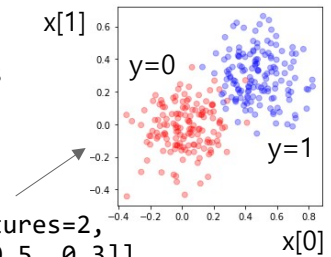


Figure 4 (b): Vertical partitioning. Worker who proposes global best split broadcasts the instance placement after node splitting.

MX-AI

- Simple Implementation of Histogram-based split finding

```
# [MXML-12-01] 1.histogram-based.py
import numpy as np
from sklearn.datasets import make_blobs
from multiprocessing.pool import Pool
import matplotlib.pyplot as plt

# Create a training data set.
x, y = make_blobs(n_samples=300, n_features=2,
                  centers=[[0., 0.], [0.5, 0.3]],
                  cluster_std=0.15, center_box=(-1., 1.))

plt.figure(figsize=(4,4))
color = [['red', 'blue'][a] for a in y]
plt.scatter(x[:,0], x[:,1], c=color, alpha=0.3)
plt.show()

def find_local_split_point(f, s_point):
    GL = HL = 0.0
    l_bound = -np.inf           # lower left bound
    max_gain = -np.inf          # initialize max_gain

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

        # After splitting the parent node, calculate the scores
        # of its children.
        GL += g[left].sum()
        HL += h[left].sum()
        GR = G - GL
        HR = H - HL
```




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

        # Find the maximum gain.
        if gain > max_gain:
            max_gain = gain
            b_point = j      # local best split point
        l_bound = j

    return b_point, max_gain

y0 = np.ones(shape=y.shape) * 0.5  # initial prediction
g = -(y - y0)                      # negative residual.
h = y0 * (1. - y0)                 # Hessian.

# Create a histogram of the parent node for each feature
n_bin = 30  # the number of bins
g0_parent, f0_bin = np.histogram(x[:, 0], n_bin, weights=g)  # feature 0
g1_parent, f1_bin = np.histogram(x[:, 1], n_bin, weights=g)  # feature 1

# Find the best split point of each feature
G = g.sum()
H = h.sum()
r = 0.01
p_score = (G ** 2) / (H + r) # parent's score before splitting the node

# Find global best split point through parallel processing
# vertical partitioning method is used.
mp = Pool(2)
args = [[x[:, 0], f0_bin], [x[:, 1], f1_bin]]
ret = mp.starmap_async(find_local_split_point, args)
```

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

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

MX-AI

- Simple Implementation of Histogram-based split finding

```
mp.close()
mp.join()

results = ret.get()
p1 = results[0][0];    p2 = results[1][0]
gain1 = results[0][1]; gain2 = results[1][1]

if gain1 > gain2:
    b_fid = 0
    b_point = p1
else:
    b_fid = 1
    b_point = p2

print('\nbest feature id =', b_fid)
print('best split point =', b_point.round(3))


Result:
best feature id = 0
best split point = 0.231
```

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models ← {}, fact ← $\frac{1-a}{b}$
topN ← a × len(I) , randN ← b × len(I)
for $i = 1$ ***to*** d **do**
  preds ← models.predict(I)
  g ← $loss$(I, preds), w ← {1,1,...}
  sorted ← GetSortedIndices(abs(g))
  topSet ← sorted[1:topN]
  randSet ← RandomPick(sorted[topN:len(I)], randN)
  usedSet ← topSet + randSet
  w[randSet] × = fact ▷ Assign weight $fact$ to the small gradient data.
  newModel ← L(I[usedSet], − g[usedSet], w[usedSet])
  models.append(newModel)

12. Light GBM

Part 2: Gradient-based One-Side Sampling (GOSS)

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

www.youtube.com/@meanxai

MX-AI

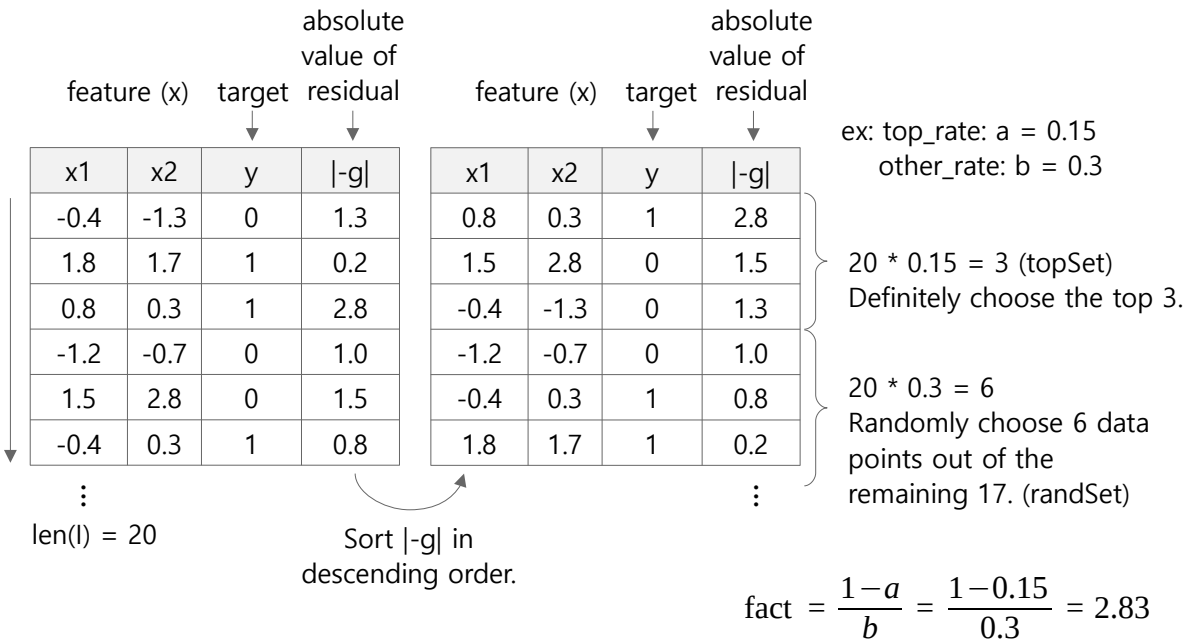
▪ Gradient-based One-Side Sampling (GOSS)

- Most ensemble learning methods, such as RandomForest, AdaBoost, SGBM, and XGBoost, randomly sample data to increase the efficiency of training.
- LightGBM uses GOSS to sample data. Sampling is performed using the size of the absolute value of the residuals. And performance does not drop significantly.
- **Core idea**: The data points with small residuals can be excluded from training in the next iteration round because the predictions are already reliable. Those that do not should be selected for further training in the next round.

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models ← {}, fact ← $\frac{1-a}{b}$
topN ← a × len(I) , randN ← b × len(I)
for $i = 1$ ***to*** d **do**
  preds ← models.predict(I)
  g ← $loss$(I, preds), w ← {1,1,...}
  sorted ← GetSortedIndices(abs(g))
  topSet ← sorted[1:topN]
  randSet ← RandomPick(sorted[topN:len(I)], randN)
  usedSet ← topSet + randSet
  w[randSet] × = fact ▷ Assign weight $fact$ to the small gradient data.
  newModel ← L(I[usedSet], − g[usedSet], w[usedSet])
  models.append(newModel)

* Source: Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree



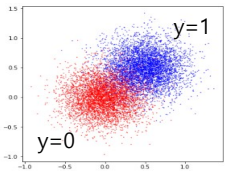
$$\text{fact} = \frac{1-a}{b} = \frac{1-0.15}{0.3} = 2.83$$

The data points in topSet are given a weight of 1.0, and those in randSet are given a weight of 2.83. When calculating Σg and Σh for the data in randSet, 17 g's and 17 h's must be added, but since only 6 were added, multiply by 2.83. (2.83 * 6 = 17). This would have the effect of adding 17 g's and 17 h's.

MX-AI

▪ Implementation of GOSS

```
# [MXML-12-02] 2.goss.py
# Implement GOSS algorithm presented in the paper.
# 1. Adding GOSS feature to XGBClassifier
# 2. Using the LGBMClassifier library
# Source: www.youtube.com/@meanxai
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
```




```
# Create a training dataset
x, y = make_blobs(n_samples=10000, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.25, center_box=(-1., 1.))

n_boost = 50   # the number of boosting
eta = 0.3      # learning rate
max_depth = 2  # max_depth of a tree

def base_model(x, y, weights, F0):
    model = XGBClassifier(n_estimators=1,     # just 1 round
                          learning_rate=eta,
                          max_depth=max_depth,
                          max_bin=20, tree_method='hist',
                          base_score=None)

    # g and h are multiplied by their weights.
    model.fit(x, y, sample_weight=weights, base_margin=F0)
    return model
```

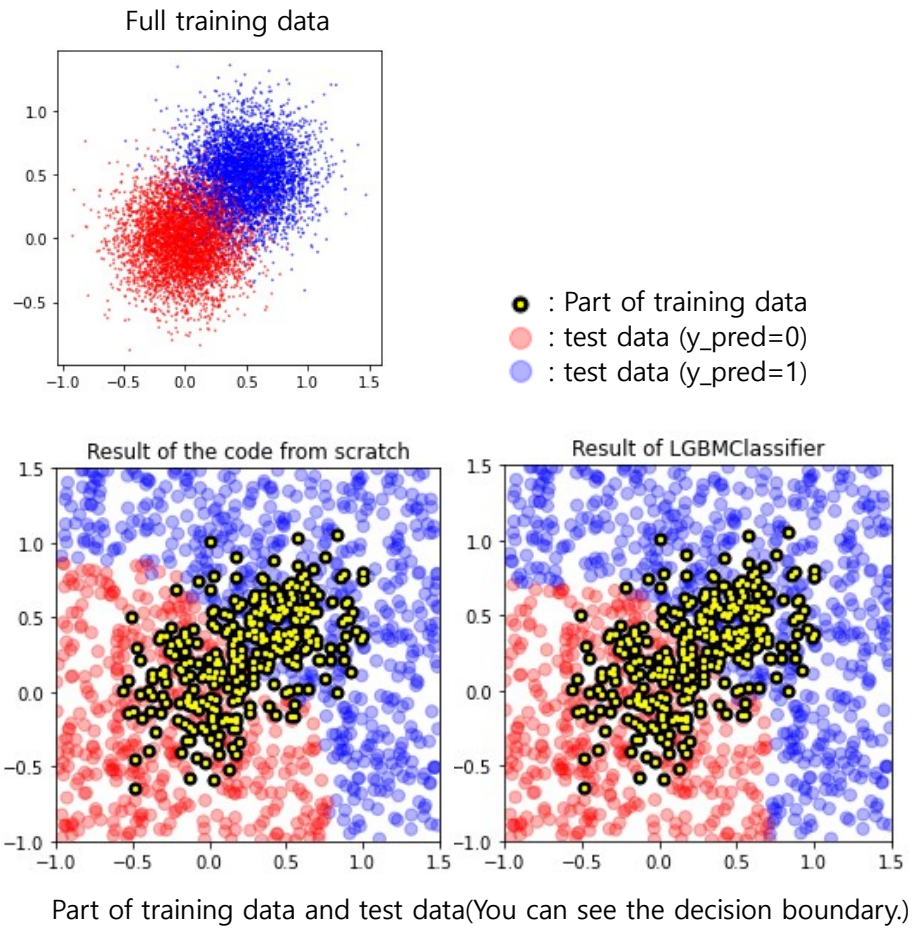
$$\tilde{L}^{(t)} \simeq \sum_{i=1}^{n} weight_i \left(g_i w_{x_i} + \frac{1}{2} h_i w_{x_i}^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

```
# Algorithm 2: Gradient-based One-Side Sampling (GOSS)
a = 0.3          # sampling ratio of large gradient data
b = 0.2          # sampling ratio of small gradient data
fact = (1. - a) / b
topN = int(a * x.shape[0])
randN = int(b * x.shape[0])
models = []
Fm = np.zeros(y.shape)  # initial log(odds) prediction

for i in range(n_boost):
    y_prev = 1. / (1. + np.exp(-Fm))
    g = -(y - y_prev)     # negative residual. first order gradients
    w = np.ones(shape=x.shape[0])  # initial sample weights.
    sorted_g = np.argsort(np.abs(g))[::-1]
    topSet = sorted_g[:topN]
    randSet = np.random.choice(sorted_g[topN:], size=randN,replace=False)
    usedSet = np.hstack([topSet, randSet])
    w[randSet] *= fact  # Assign weight fact to the small gradient data

    newModel = base_model(x[usedSet], y[usedSet],
                          w[usedSet], F0=Fm[usedSet])
    Fm += newModel.predict(x, output_margin=True)
    models.append(newModel)

# Create a test dataset and predict the class of test data
x_test = np.random.uniform(-1.0, 1.5, (1000, 2))

test_Fm = np.zeros(x_test.shape[0])
for model in models:
    test_Fm += model.predict(x_test, output_margin=True)
```

MX-AI

▪ Implementation of GOSS

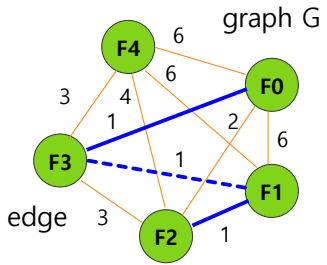
```
y_prob = 1. / (1. + np.exp(-test_Fm))  # log(odds) --> probability
y_pred = (y_prob > 0.5) * 1

# Check the prediction results and the decision boundary.
def check_result(x, y, x_test, y_pred, title):
    plt.figure(figsize=(4,4))
    color2 = [['red', 'blue'][a] for a in y_pred]
    plt.scatter(x_test[:, 0], x_test[:, 1], s=50, c=color2,
                alpha=0.3)

    # Only part of the training data is drawn.
    plt.scatter(x[:300, 0], x[:300, 1], s=50, c='black')
    plt.scatter(x[:300, 0], x[:300, 1], s=5, c='yellow')
    plt.xlim(-1.0, 1.5)
    plt.ylim(-1.0, 1.5)
    plt.title(title)
    plt.show()

check_result(x, y, x_test, y_pred, "Result of the code from scratch")

# Use LGBMClassifier library and compare the result from above code
model = LGBMClassifier(n_estimators = 20,
                       max_depth=max_depth,
                       learning_rate=eta,
                       max_bins=20,
                       boosting_type="goss",
                       top_rate=0.3,
                       other_rate=0.2)

model.fit(x, y)
y_pred = model.predict(x_test)
check_result(x, y, x_test, y_pred, "Result of LGBMClassifier")
```



Part of training data and test data(You can see the decision boundary.)

Features

	F0	F1	F2	F3	F4
i0	1	1	0	0	1
i1	0	0	1	1	1
i2	1	2	0	0	2
i3	0	0	2	3	1
i4	2	1	0	0	3
i5	3	3	0	0	1
i6	0	0	3	0	2
i7	1	2	3	4	3
i8	1	0	1	0	0
i9	2	3	0	0	2



	F0	F1	F2	F3	F4
F0		6	2	1	6
F1	6		1	1	6
F2	2	1		3	4
F3	1	1	3		3
F4	6	6	4	3	
∑	15	14	10	8	19

conflict count matrix

12. Light GBM

Part 3: Exclusive Feature Bundling (EFB) Greedy Bundling Algorithm

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

www.youtube.com/@meanxai

MX-AI

▪ Exclusive Feature Bundling (EFB) & Merge Exclusive Features

Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree

4 Exclusive Feature Bundling

In this section, we propose a novel method to effectively reduce the number of features.

High-dimensional data are usually very sparse. The sparsity of the feature space provides us a possibility of designing a nearly lossless approach to reduce the number of features. Specifically, in a sparse feature space, many features are mutually exclusive, i.e., they never take nonzero values simultaneously. We can safely bundle exclusive features into a single feature (which we call an exclusive feature bundle). By a carefully designed feature scanning algorithm, we can build the same feature histograms from the feature bundles as those from individual features.

There are two issues to be addressed. The first one is to determine which features should be bundled together. The second is how to construct the bundle.

5 features

	F0	F1	F2	F3	F4
i0	1	1	0	0	1
i1	0	0	1	1	1
i2	1	2	0	0	2
i3	0	0	2	3	1
i4	2	1	0	0	3
i5	3	3	0	0	1
i6	0	0	3	0	2
i7	1	2	3	4	3
i8	1	0	1	0	0
i9	2	3	0	0	2

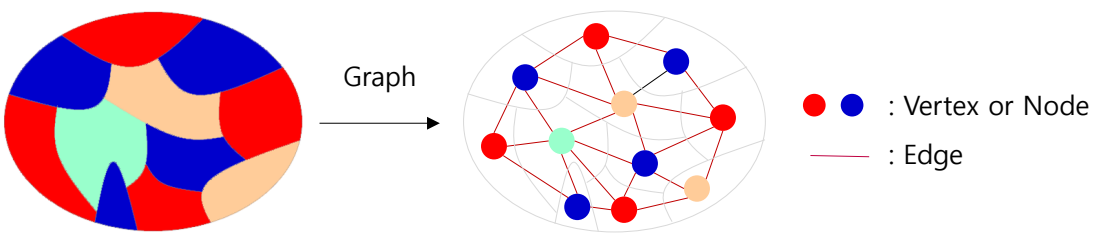
→

3 features

	F0_F3	F1_F2	F4
i0	1	1	1
i1	4	4	1
i2	1	2	2
i3	6	5	1
i4	2	1	3
i5	3	3	1
i6	0	6	2
i7	7	6	3
i8	1	4	0
i9	2	3	2

▪ Graph coloring problem

Graph coloring refers to the problem of coloring the vertices of a graph so that the colors of two adjacent vertices are not the same.



MX-AI

▪ Exclusive Feature Bundling (EFB) - Greedy Bundling

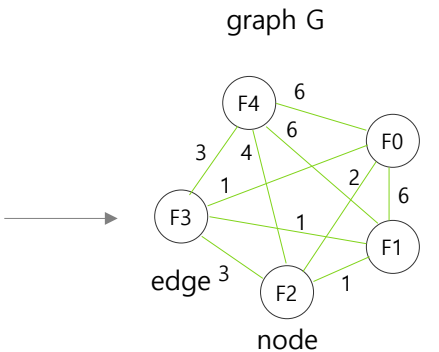
- When feature values simultaneously take on non-zero values, this is called a "conflict". Otherwise, they are mutually exclusive. Exclusive features can be bundled into a single feature. In the example below, the number of conflicts for features F0 and F1 is 6. For F1 and F2, it is 1. Features with less conflict can be bundled.
- Create a conflict count matrix and then sort it in descending order by the sum of the conflict counts. In the left algorithm, the sum of the conflict counts is called "degree", the sorted thing is called "searchOrder".
- Feature bundling is performed by sequentially searching the features in the searchOrder matrix. F4 → F0 → F1 → F2 → F3

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← { }, bundlesConflict ← { }
for i **in** searchOrder **do**
    needNew ← True
    for j = 1 **to** len(bundles) **do**
        cnt ← ConflictCnt(bundles[j], F[i])
        if cnt + bundlesConflict[j] ≤ K **then**
            bundles[j].add(F[i]), needNew ← False
            bundlesConflict[j] += cnt
            break
    if needNew **then**
        Add F[i] as a new bundle to bundles
        bundlesConflict.append(0)
Output: bundles

Features

	F0	F1	F2	F3	F4
i0	1	1	0	0	1
i1	0	0	1	1	1
i2	1	2	0	0	2
i3	0	0	2	3	1
i4	2	1	0	0	3
i5	3	3	0	0	1
i6	0	0	3	0	2
i7	1	2	3	4	3
i8	1	0	1	0	0
i9	2	3	0	0	2



conflict count matrix

	F0	F1	F2	F3	F4
F0		6	2	1	6
F1	6		1	1	6
F2	2	1		3	4
F3	1	1	3		3
F4	6	6	4	3	
∑	15	14	10	8	19

sort

searchOrder

	F4	F0	F1	F2	F3
∑	19	15	14	10	8

* Source : Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree. (Fixed some typos and omissions: red)

* Example source: https://www.youtube.com/watch?v=4C8SUZJPlMY

MX-AI

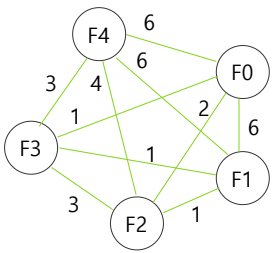
▪ Exclusive Feature Bundling (EFB) - Greedy Bundling

- Set the maximum number of conflicts (K), search the graph in searchOrder order, and perform the graph coloring following the Algorithm-3 (Greedy Bundling).
- In the example below, K is set to 1 and all edges with conflict count greater than 1 are removed.
- F1 and F3 are connected with conflict count = 1, but F3 is already connected to F0, so the connection between F1 and F3 has been removed.
- As a result of performing Algorithm 3, the 5 features were bundled into 3. {F4}, {F0, F3}, {F1, F2}

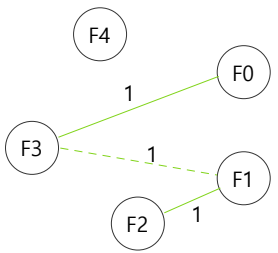
Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← { }, bundlesConflict ← { }
for i **in** searchOrder **do**
    needNew ← True
    for j = 1 **to** len(bundles) **do**
        cnt ← ConflictCnt(bundles[j], F[i])
        if cnt + bundlesConflict[j] ≤ K **then**
            bundles[j].add(F[i]), needNew ← False
            bundlesConflict[j] += cnt
            break
    if needNew **then**
        Add F[i] as a new bundle to bundles
        bundlesConflict.append(0)
Output: bundles

graph G



graph G



searchOrder

	F4	F0	F1	F2	F3
∑	19	15	14	10	8

While searching for searchOrder, drop the connections with conflictCnt > 1.

Input

	F0	F1	F2	F3	F4
i0	1	1	0	0	1
i1	0	0	1	1	1
i2	1	2	0	0	2
i3	0	0	2	3	1
i4	2	1	0	0	3
i5	3	3	0	0	1
i6	0	0	3	0	2
i7	1	2	3	4	3
i8	1	0	1	0	0
i9	2	3	0	0	2

	j=1	j=2	j=3	
bundles:	F4	F0 F3	F1 F2	← If needNew = True, add a new bundle.
bundlesConflict:	0	1	1	

* Source : Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree. (Fixed some typos and omissions: red)

* Example source: https://www.youtube.com/watch?v=4C8SUZJPlMY

MX-AI

▪ Implementation of Greedy Bundling

```
# [MXML-12-03] 3.greedy_bundling.py
# Algorithm 3: Greedy Bundling
import numpy as np

x = np.array([[1, 1, 0, 0, 1],
              [0, 0, 1, 1, 1],
              [1, 2, 0, 0, 2],
              [0, 0, 2, 3, 1],
              [2, 1, 0, 0, 3],
              [3, 3, 0, 0, 1],
              [0, 0, 3, 0, 2],
              [1, 2, 3, 4, 3],
              [1, 0, 1, 0, 0],
              [2, 3, 0, 0, 2]])

# Create a conflict count matrix
n_row = x.shape[0]
n_col = x.shape[1]
conflictCnt = np.zeros((n_col, n_col))

for i in range(n_col):
    for j in range(i+1, n_col):
        # Count the number of conflicts.
        conflictCnt[i, j] = len(np.where(x[:, i] * x[:, j] > 0)[0])

# Copy upper triangle to lower triangle
# iu = (array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3]),
#       array([1, 2, 3, 4, 2, 3, 4, 3, 4, 4]))
iu = np.triu_indices(n_col, 1)
il = (iu[1], iu[0])
conflictCnt[il] = conflictCnt[iu]
```

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← { }, bundlesConflict ← { }
for i **in** searchOrder **do**
  needNew ← True
  for j = 1 **to** len(bundles) **do**
    cnt ← ConflictCnt(bundles[j], F[i])
    if cnt + bundlesConflict[j] ≤ K **then**
      bundles[j].add(F[i]), needNew ← False
      bundlesConflict[j] += cnt
      break
  if needNew **then**
    Add F[i] as a new bundle to bundles
    bundlesConflict.append(0)
Output: bundles

	F0	F1	F2	F3	F4
F0		6	2	1	6
F1	6		1	1	6
F2	2	1		3	4
F3	1	1	3		3
F4	6	6	4	3	
∑	15	14	10	8	19

```
# Create a search order matrix
degree = conflictCnt.sum(axis=0)
searchOrder = np.argsort(degree)[::-1]  # descending order

K = 1        # max conflict count
bundles, bundlesConflict = [], []
for i in searchOrder:  # i = [4, 0, 1, 2, 3]
    needNew = True
    for j in range(len(bundles)):
        cnt = conflictCnt[bundles[j][-1], i]
        # Only edges less than or equal to K are considered.
        if cnt + bundlesConflict[j] <= K:
            # Add the feature number i to the j-th bundle.
            bundles[j].append(i)

            # Update the number of conflicts of features in the
            # j-th bundle.
            bundlesConflict[j] += cnt
            needNew = False
            break

    if needNew:
        bundles.append([i])
        bundlesConflict.append(0.)

print('bundles:', bundles)
print('bundlesConflict:', bundlesConflict)

        bundles: [[4], [0, 3], [1, 2]]
bundlesConflict: [0.0,   1.0,    1.0]
```

searchOrder

	F4	F0	F1	F2	F3
∑	19	15	14	10	8

Algorithm 4: Merge Exclusive Features (modified)

Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin
    binRang
newBin ← F
for i = 1 **to**
    for j = 2
        if F[j]
            new
Output: new

Algorithm 4: Merge Exclusive Features

Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new Bin(numData)
for i = 1 **to** numData **do**
    newBin[i] ← 0
    for j = 1 **to** len(F) **do**
        if F[j].bin[i] ≠ 0 **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges

12. Light GBM

Part 4: Merge Exclusive Features

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

www.youtube.com/@meanxai

```
bundles: [[4], [0, 3], [1, 2]]
bundlesConflict: [ 0 ,   1 ,     1   ]
```

MX-AI

▪ Merge Exclusive Features

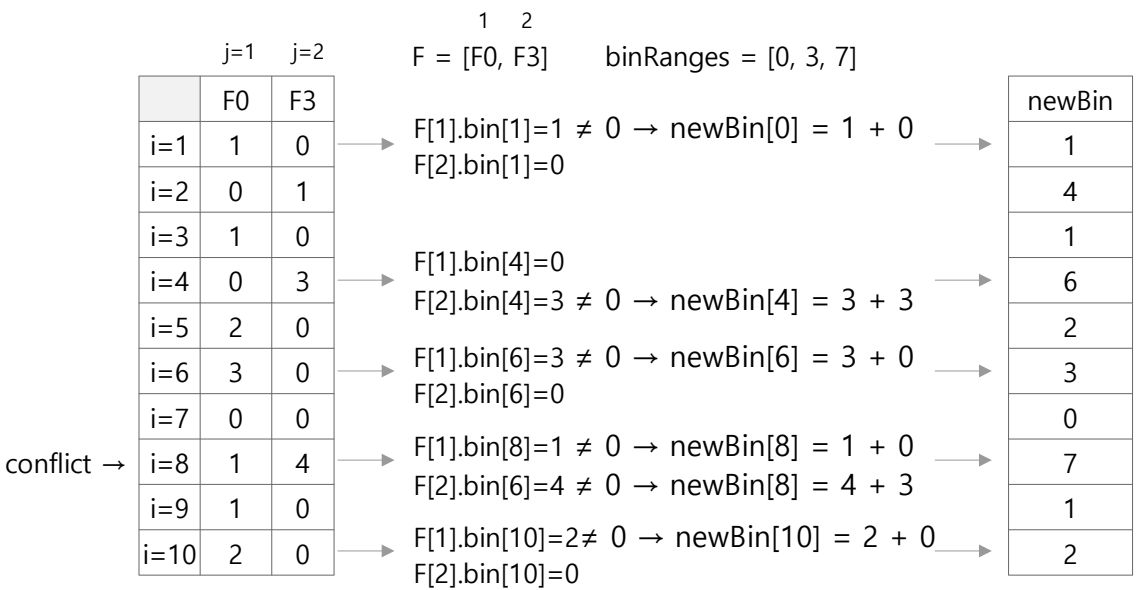
4 Exclusive Feature Bundling Guolin Ke, et, al., 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree

<page 6>

For the second issues, we need a good way of merging the features in the same bundle in order to reduce the corresponding training complexity. The key is to ensure that the values of the original features can be identified from the feature bundles. Since the histogram-based algorithm stores discrete bins instead of continuous values of the features, we can construct a feature bundle by letting exclusive features reside in different bins. This can be done by adding offsets to the original values of the features. For example, suppose we have two features in a feature bundle. Originally, feature A takes value from [0, 10) and feature B takes value [0, 20). We then add an offset of 10 to the values of feature B so that the refined feature takes values from [10, 30). After that, it is safe to merge features A and B, and use a feature bundle with range [0, 30] to replace the original features A and B. The detailed algorithm is shown in Alg. 4.

Algorithm 4: Merge Exclusive Features

Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new Bin(numData)
for i = 1 **to** numData **do**
    newBin[i] ← 0
    for j = 1 **to** len(F) **do**
        if F[j].bin[i] ≠ 0 **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges



* Example source: https://www.youtube.com/watch?v=4C8SUZJPlMY

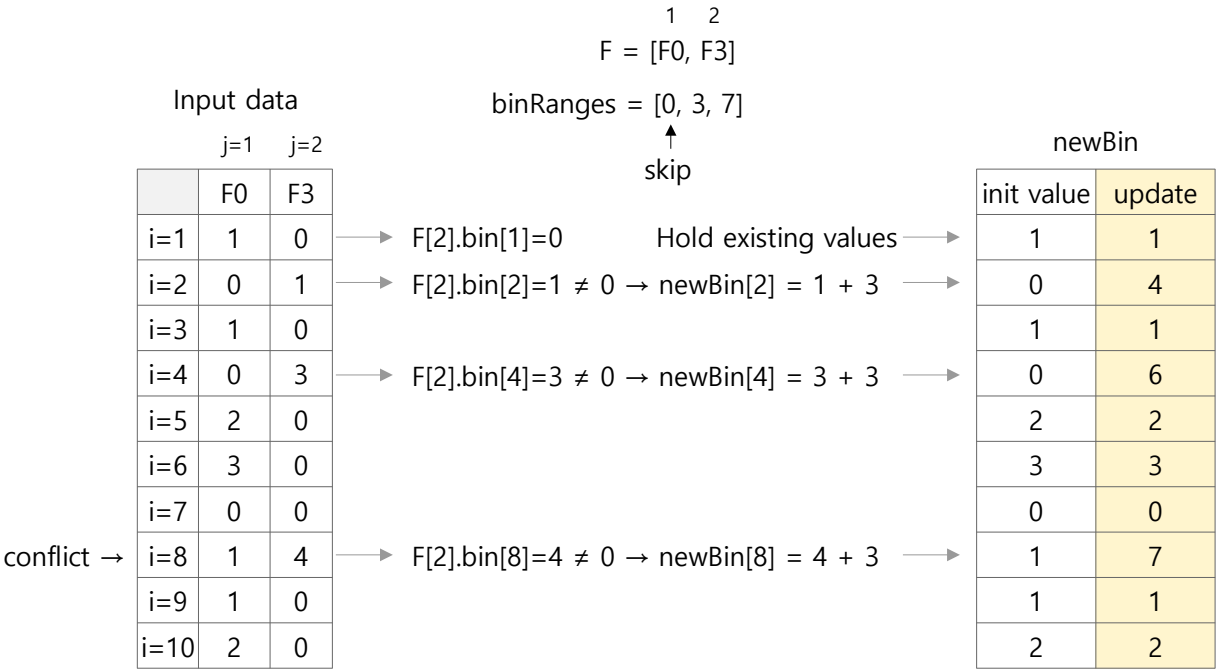
MX-AI

▪ Merge Exclusive Features (modified: skip-zero-version)

- The computation time can be slightly shortened by modifying Algorithm 4 as follows.
- The previous version initialized newBin to 0, and repeated from j=1. This version initializes newBin to F0, and repeated from j=2. This reduces the number of iterations since we can skip the first 0 in binRange = [0, 3, 7].

Algorithm 4: Merge Exclusive Features (modified)

Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← F0 # initialize with F0
for i = 1 **to** numData **do**
    for j = 2 **to** len(F) **do** # skip zero, binRanges[1]
        if F[j].bin[i] ≠ 0 **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges



* Example source: https://www.youtube.com/watch?v=4C8SUZJPlMY

MX-AI

▪ Implementation of Algorithm 4: Merge Exclusive Features

```python
# [MXML-12-04] 4.merge_features.py
# Implementation of Algorithm 4
import numpy as np

x = np.array([[1, 1, 0, 0, 1],
              [0, 0, 1, 1, 1],
              [1, 2, 0, 0, 2],
              [0, 0, 2, 3, 1],
              [2, 1, 0, 0, 3],
              [3, 3, 0, 0, 1],
              [0, 0, 3, 0, 2],
              [1, 2, 3, 4, 3],
              [1, 0, 1, 0, 0],
              [2, 3, 0, 0, 2]])

# The result of Greedy Bundling
bundles = [[4], [0, 3], [1, 2]]

# Algorithm 4: Merge Exclusive Features
def merge_features(numData, F):
    binRanges = [0]
    totalBin = 0
    for f in F:
        totalBin += np.max(f)
        binRanges.append(totalBin)

    newBin = np.zeros(numData, dtype=int)
    for i in range(numData):
        newBin[i] = 0
        for j in range(len(F)):
            if F[j][i] != 0:
                newBin[i] = F[j][i] + binRanges[j]
    return newBin, binRanges
```

Algorithm 4: Merge Exclusive Features (modified)

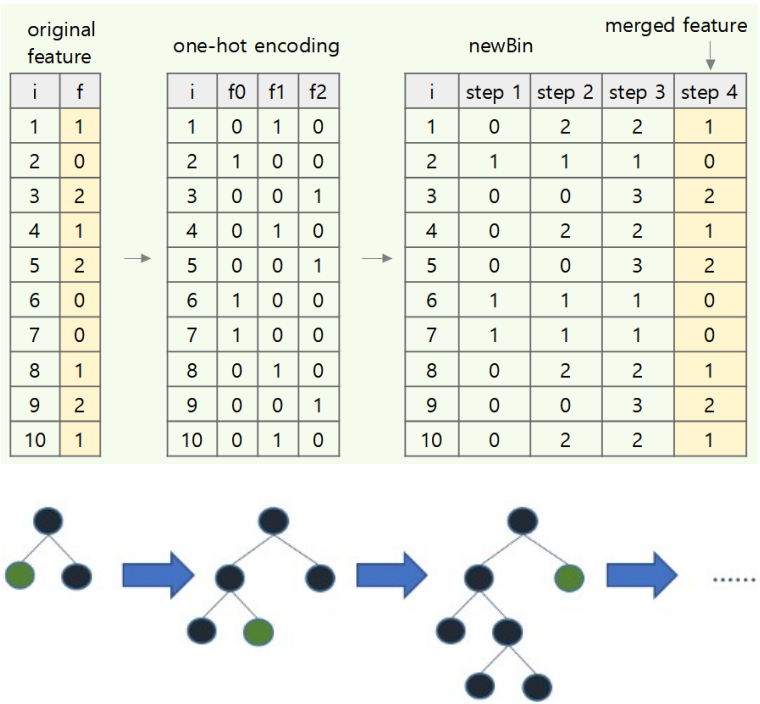
Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← F0 # initialize with F0
for i = 1 **to** numData **do**
    for j = 2 **to** len(F) **do** # skip zero, binRanges[1]
        if F[j].bin[i] ≠ 0 **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges

Algorithm 4: Merge Exclusive Features

Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f **in** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new Bin(numData)
for i = 1 **to** numData **do**
    newBin[i] ← 0
    for j = 1 **to** len(F) **do**
        if F[j].bin[i] ≠ 0 **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges

```python
# modified Algorithm 4 (skip-zero-version)
def merge_features2(numData, F):
    binRanges = [0]
    totalBin = 0
    for f in F:
        totalBin += np.max(f)
        binRanges.append(totalBin)

    newBin = F[0]  # initialize newBin to F[0]
    for i in range(numData):
        for j in range(1, len(F)):
            if F[j][i] != 0:
                newBin[i] = F[j][i] + binRanges[j]
    return newBin, binRanges

F = [x[:, i] for i in bundles[1]]
newBin, binRanges = merge_features(x.shape[0], F)
print('\nnewBin:', newBin)
print('binRanges:', binRanges)

newBin, binRanges = merge_features2(x.shape[0], F)
print('\nnewBin:', newBin)
print('binRanges:', binRanges)
```

```
Results:
newBin: [1 4 1 6 2 3 0 7 1 2]
binRanges: [0, 3, 7]

newBin: [1 4 1 6 2 3 0 7 1 2]
binRanges: [0, 3, 7]
```



i	f
1	1
2	0
3	2
4	1
5	2
6	0
7	0
8	1
9	2
10	1

i	f0	f1	f2
1	0	1	0
2	1	0	0
3	0	0	1
4	0	1	0
5	0	0	1
6	1	0	0
7	1	0	0
8	0	1	0
9	0	0	1
10	0	1	0

i	step 1	step 2	step 3	step 4
1	0	2	2	1
2	1	1	1	0
3	0	0	3	2
4	0	2	2	1
5	0	0	3	2
6	1	1	1	0
7	1	1	1	0
8	0	2	2	1
9	0	0	3	2
10	0	2	2	1

12. Light GBM

Part 5: Greedy Bundling + Merge Exclusive Features

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

www.youtube.com/@meanxai

MX-AI

▪ Merge Exclusive Features – One-hot encoding

- Merging one-hot encoded features allows us to better understand how this algorithm works and what its effects are.
- The merge exclusive features algorithm restores the one-hot encoded features to the original feature.
- One-hot encoded features are 100% mutually exclusive with no conflicts whatsoever. Therefore, they can be perfectly restored to the original feature.
- However, when multiple features are one-hot encoded, individual features are not 100% restored because the Greedy Bundling algorithm is not perfect.
- This is because the Greedy Bundling is an approximate algorithm that finds a reasonably good bundling strategy within polynomial time.

original feature

i	f
1	1
2	0
3	2
4	1
5	2
6	0
7	0
8	1
9	2
10	1

one-hot →

one-hot encoding

i	f0	f1	f2
1	0	1	0
2	1	0	0
3	0	0	1
4	0	1	0
5	0	0	1
6	1	0	0
7	1	0	0
8	0	1	0
9	0	0	1
10	0	1	0

merge →

newBin

merged feature ↓

i	step 1	step 2	step 3	step 4
1	0	2	2	1
2	1	1	1	0
3	0	0	3	2
4	0	2	2	1
5	0	0	3	2
6	1	1	1	0
7	1	1	1	0
8	0	2	2	1
9	0	0	3	2
10	0	2	2	1

j=1, 2, 3

F = [f0, f1, f2] binRanges = [0, 1, 2, 3] ← offsets

- step-1 : initialize newBin to f0 values
- step-2 : update newBin using f1 values
  - f1.bin ≠ 0, newBin = f1.bin + binRanges[1]
  - f1.bin = 0, newBin = Hold existing values
- step-3 : update newBin using f1 values
  - f2.bin ≠ 0, newBin = f2.bin + binRanges[2]
  - f2.bin = 0, newBin = Hold existing values
- setp-4 : newBin = newBin - 1

* Merged feature values are identical to the original feature values.

MX-AI

▪ Implementation of EFB: Greedy Bundling & Merge Exclusive Features

```python
# [MXML-12-05] 5.efb_onehot.py
# Merge one-hot encoded features using EFB
import numpy as np
from sklearn.preprocessing import OneHotEncoder

# Algorithm 3: Greedy Bundling
def greedy_bundling(x, K):
    # Create a conflict count matrix
    n_row = x.shape[0]
    n_col = x.shape[1]
    conflictCnt = np.zeros((n_col, n_col))

    for i in range(n_col):
      for j in range(i+1, n_col):
          # Count the number of conflicts.
          conflictCnt[i, j]=len(np.where(x[:, i] * x[:, j] > 0)[0])

    # Copy upper triangle to lower triangle
    iu = np.triu_indices(n_col, 1)
    il = (iu[1], iu[0])
    conflictCnt[il] = conflictCnt[iu]

    # Create a search order matrix
    degree = conflictCnt.sum(axis=0)
    searchOrder = np.argsort(degree)[::-1]  # descending order

    bundles = []
    bundlesConflict = []
    for i in searchOrder:
        needNew = True
        for j in range(len(bundles)):
            cnt = conflictCnt[bundles[j][-1], i]
            if cnt + bundlesConflict[j] <= K:
                bundles[j].append(i)
                bundlesConflict[j] += cnt
                needNew = False
                break

        if needNew:
            bundles.append([i])
            bundlesConflict.append(0.)
    return bundles

# Algorithm 4: Merge Exclusive Features (skip-zero-version)
def merge_features(numData, F):
    binRanges = [0]
    totalBin = 0
    for f in F:
        totalBin += np.max(f)
        binRanges.append(totalBin)

    newBin = F[0] # initialize newBin to F[0]
    for i in range(numData):
        for j in range(1, len(F)):
            if F[j][i] != 0:
                newBin[i] = F[j][i] + binRanges[j]
    return newBin, binRanges
```

MX-AI

▪ Implementation of EFB: Greedy Bundling & Merge Exclusive Features

```
# Generate random data and perform one-hot encoding.
n_samples = 100
n_features = 4
x = np.random.randint(low=0, high=4, size=(n_samples, n_features))
enc = OneHotEncoder()
x_ohe = enc.fit_transform(x).toarray()

print('Original features [:5]:'); print(x[:5])
print('\nOne-hot encoding [:5]:'); print(x_ohe[:5])

# bundling
bundles = greedy_bundling(x_ohe, K=1)

# If we know the bundles exactly, like this,
# bundles = [[0,1,2,3], [4,5,6,7], [8,9,10,11], [12,13,14,15]]
# we can get the original features from the merged features.

print('\nbundles:', bundles)
# [[14, 12, 15, 13], [10, 8, 11, 9], [5, 4, 6, 7], [3, 2, 1, 0]]

# Feature를 병합한다.
x_efb = np.zeros(shape=x.shape).astype('int')
for i, bundle in enumerate(bundles):
    F = [x_ohe[:, i] for i in bundle]
    newBin, binRanges = merge_features(x_ohe.shape[0], F)
    x_efb[:, i] = np.array(newBin) - 1
```

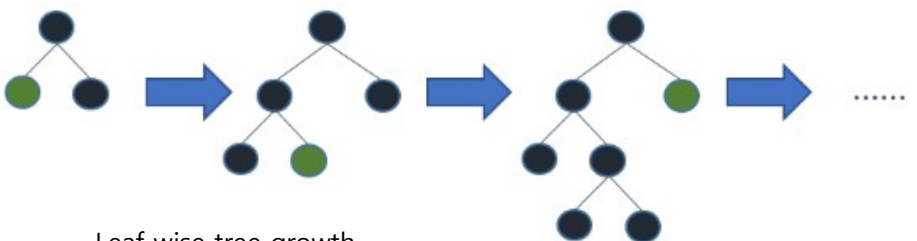
```
print('\nOriginal features [:5]:'); print(x[:5])
print('\nMerged features [:5]:'); print(x_efb[:5])

The result of Greedy Bundling:
bundles: [[11,8,9,10], [4,5,6,7], [14,13,12,15], [0,1,2,3]]

Original features [:5]:
[[3 1 3 1]
 [2 2 0 3]
 [3 1 1 2]
 [0 0 3 0]
 [3 0 3 2]]

One-hot encoding [:5]:
[[0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0.]
 [0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0.]]



Original features [:5]:        Merged features [:5]:
[[3 1 3 1]                     [[0 1 1 3]
 [2 2 0 3]                      [1 2 3 2]
 [3 1 1 2]                      [2 1 0 3]
 [0 0 3 0]                      [0 0 2 0]
 [3 0 3 2]]                     [0 0 0 3]]
```

MX-AI

▪ Depth-wise vs. Leaf-wise tree growth

- Most decision tree learning algorithms grow trees by level (depth)-wise, as shown in the left picture below.
- LightGBM grows trees leaf-wise (best-first). It will choose the leaf with max delta loss to grow. Holding the number of leaves fixed, leaf-wise algorithms tend to achieve lower loss than level-wise algorithms.
- Leaf-wise method can converge much faster, but can lead to overfitting when the number of data points is small, so LightGBM includes the max_depth parameter to limit tree depth. However, trees still grow leaf-wise even when max_depth is specified.
- Leaf-wise was proposed by Haijan Shi in 2007 in a paper titled "Best-first decision tree learning."
- XGBoost uses depth-wise tree growth. To use the leaf-wise method in the XGBoost library, use parameter tree_method = hist, grow_policy = lossguide.

Level-wise tree growth



Leaf-wise tree growth

* Source: https://lightgbm.readthedocs.io/en/v4.3.0/Features.html

MX-AI

▪ Santander Customer Transaction Prediction using XGBoost and LightGBM

```
# [MXML-12-05] 6.santander.py
import pandas as pd
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import time

# 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_feat = df.drop('TARGET', axis=1)
y_target = df['TARGET']
x_train, x_test, y_train, y_test = train_test_split(x_feat,
                                                    y_target)

# 1. XGBoost
# Create an XGBoost model and fit it to the training data
start_time = time.time()
```

```
model = XGBClassifier(n_estimators = 200,
                      max_depth = 5,
                      learning_rate = 0.1,     # η
                      gamma = 0.1,             # γ for pruning
                      reg_lambda = 1.0,        # λ for regularization
                      base_score = 0.5,        # initial prediction value
                      subsample = 0.5,         # Subsample ratio of the
                                               # training instance
                      colsample_bynode = 0.5, # Subsample ratio of
                                               # columns for each split
                      max_bin = int(1/0.03),   # ε = 0.03, bins = 33
                      tree_method = 'approx') # weighted quantile sketch

# training
model.fit(x_train, y_train)

# Predict the test data and measure the performance with ROC-AUC.
y_prob = model.predict_proba(x_test)[:, 1]
auc = roc_auc_score(y_test, y_prob)

print('\nXGBoost results:')
print('running time = {:.2f} seconds'.format(time.time() -
start_time))
print('ROC-AUC = {:.4f}'.format(auc))
```

MX-AI

▪ Santander Customer Transaction Prediction using XGBoost and LightGBM

```
# 2. LightGBM
# Create a LightGBM model
start_time = time.time()
model = LGBMClassifier(n_estimators = 200,
              max_depth = 5,
              learning_rate = 0.1,
              boosting_type="goss",  # default: gbdt – traditional
                                     # gradient based decision tree
              top_rate=0.3,
              other_rate=0.2,
              enable_bundle=True,    # default: True. enable EFB
              is_unbalance = True)

# training
model.fit(x_train, y_train)

# Predict the test data and measure the performance with AUC.
y_pred = model.predict_proba(x_test)[:, 1]
auc = roc_auc_score(y_test, y_pred)

print('\nLightGBM results:')
print('running time = {:.2f} seconds'.format(time.time() -
start_time))
print("ROC AUC = {0:.4f}".format(auc))
```

```
# Draw the ROC curve
fprs, tprs, thresholds = roc_curve(y_test, y_pred)

plt.plot(fprs, tprs, label = 'ROC')
plt.plot([0,1], [0,1], '--', label = 'Random')
plt.legend()
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```

```
XGBoost results:
running time = 5.44 seconds
ROC-AUC = 0.8421

LightGBM results:
running time = 0.63 seconds
ROC AUC = 0.8316
```