XGBoost: A Scalable Tree Boosting System

COMP7404 Project Group 32 Presentation

Group Members

Kunyu Wang 3036372992

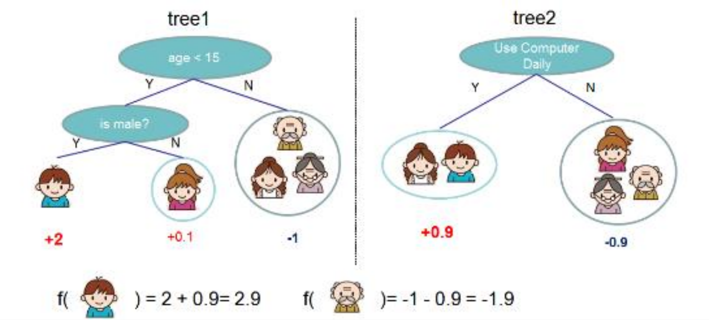
Ruixi Liu 3036371223

TABLE OF CONTENT

TREE BOOSTING

TREE BOOSTING

- **Regression Tree**
  - Pick one feature on each split
  - Divide the samples into “pure” leaf nodes
  - One score in each leaf node



- **Regression Tree Ensemble**
  - Sum up the predicted score from each tree

TREE BOOSTING

- **Model: Additive functions**
  - $\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in F$
  - $F = \{f(x) = w_{q(x)}\}, q: R^m \rightarrow T, w \in R^T$
  - K trees
  - q: the structure of each tree to map an example to the leaf index
  - T: the number of leaf nodes in each tree
  - w: leaf weights, or predicted scores
- **Objective (Regularization):**
  - $\min L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(\mathrm{f_k})$
  - $where\ \Omega(\mathrm{f_k}) = \gamma T + \frac{1}{2}\lambda||w||^2$
  - We penalize the complexity of the model, the tree size T and the weights w

GRADIENT TREE BOOSTING

- **Optimization: Boosting**
  - At t iteration (tree), $\boldsymbol{\widehat{y}_i}^{(\boldsymbol{t})} = \boldsymbol{\widehat{y}_i}^{(\boldsymbol{t-1})} + \boldsymbol{f_t}(\boldsymbol{x_i})$
  - Objective: $L^{(t)} = \sum_{i=1}^{n} l\left(y_i, \boldsymbol{\widehat{y}_i}^{(\boldsymbol{t-1})} + \boldsymbol{f_t}(\boldsymbol{x_i})\right) + \Omega(f_t)$
  - Second-order approximation:

$$L^{(t)} \cong \sum_{i=1}^{n} \left[l\left(y_i, \boldsymbol{\widehat{y}_i}^{(\boldsymbol{t-1})}\right) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

where gi and hi are the first and second order gradient statistics.

- **Leave the constant term:**
  - $\tilde{L}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$

GRADIENT TREE BOOSTING

- **Treat each leaf node as a group:**
  - $I_j = \{i|q(x_i) = j\}$: the instance set of leaf j
  - $\tilde{L}^{(t)} = \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_j + \frac{1}{2}\left(\sum_{i\in I_j} h_i + \lambda\right) w_j^2] + \gamma T$
- **By Calculus,**
  - we can get the optimal w_j^*
  - Substitute the wj, we can have an optimal value $\tilde{L}^{(t)}$
- Use it as **Split Criteria**,
  - When we split one node, the loss reduction is expressed as

$$L_{split} = \tilde{L}^{(t)}(parent) - \tilde{L}^{(t)}(left\ child) + \tilde{L}^{(t)}(right\ child)$$

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

SPLIT FINDING ALGORITHMS

SPLIT FINDING ALGORITHMS

- **Basic Exact Greedy Algorithm:**
  - Grow a tree from the root
  - At each internal node, we enumerate **all the possible splits** on **all the features**
  - **Improvement:**
    - Sort the feature values firstly
    - In python, we can use **np.cumsum** to accumulate the gradient statistics efficiently

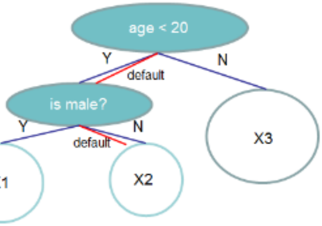
SPLIT FINDING ALGORITHMS

- **Approximate Algorithm:**
  - Not to check all the possible points for split, we only consider **candidates**.
  - How to find candidates: **Weighted Quantile Sketch**
  - **Variants:**
    - **Global Variant:** propose all the candidate splits in the initial phase of tree construction
    - **Local Variant:** re-propose candidates after each split

- **Sparsity-aware Split Finding:**
  - Background: sometimes we face a sparse input
    - Onehot encoding of categorical variables
    - Missing values in data
    - Zero values of feature
  - Solution: **Assign a default direction at each split for the missing values**

Data

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



SPLIT FINDING ALGORITHMS

- **How to get the default direction:**
  - We only consider non-missing values
  - Enumerate non-missing values of each feature
  - If loss is reduced the most (or gain most) with a split, then, all missing values **go to the opposite direction**

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

SPLIT FINDING ALGORITHMS

- **Experiments:**

  Settings:

  1. Python, i7-9750H CPU, 6 cores, 12 logical processors
  2. Dataset:
      1. Allstate insurance claim dataset
      2. predict the likelihood of an insurance claim (i.e. claim or not)
      3. Sample Size: 10K
      4. Features: 4226[1] , most are sparse features (onehot encoded)

1: Original paper has 4227 features. After our verification, for the 10K samples randomly selected from the whole dataset, one categorical feature lack missing value level (i.e. Cat6 =‘?’) . It should have been 7 levels but our selected data only has 6 levels.

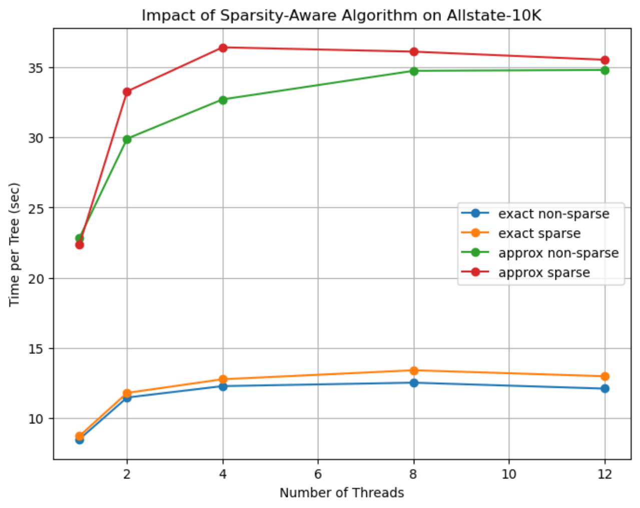
SPLIT FINDING ALGORITHMS

- **Our attempts to reproduce the results:**
  - **KEY: Parallel Learning (Feature Parallel):**

1. For each feature in split finding process, we follow the same algorithm to calculate the max split gain (loss reduction)
2. We can use different threads in CPU to process different features parallelly

SPLIT FINDING ALGORITHMS

- **Failure**:



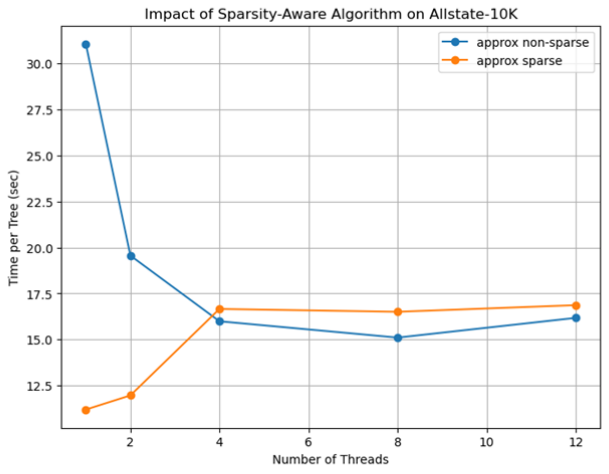
- **Difficulty** in python:

Python has Global Interpreter Lock (GIL), which ensures that **only one thread** can execute **Python bytecode** at any given moment.

SPLIT FINDING ALGORITHMS

- **Solutions**:
  1. We use **numba (@jit(nogil=True))** to unlock the GIL's restriction. **Numba** is a **Just-In-Time (JIT) compiler** for Python that optimizes numerical computations by converting Python functions into **high-performance machine code**
  2. **Column Blocks** structure (see later introduction): each column is sorted by the feature values
  3. **Approximate** Algorithm **Improvement**: use **binary search** to find the location of the threshold value

SPLIT FINDING ALGORITHMS



Conclusion:

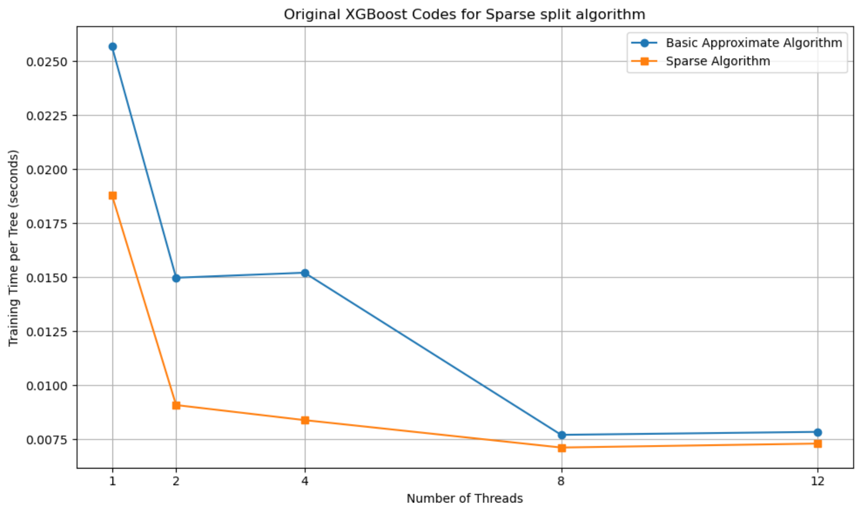
1. Parallel learning decreases the time cost for tree construction.
2. Sparse aware algorithm shows high efficiency.

Problem left:

The increasing trend of sparse algorithm may be because of the **high thread management overhead of** Python and more **memory cost** for non-missing mask. It can be improved with low level languages.

SPLIT FINDING ALGORITHMS

- **Verify by original codes:**



System Design

Column Block for Parallel Learning

- **Problem:** Sorting data is the most time-consuming step in tree learning, and traditional methods require repeated sorting.

- **Block Structure Design：**

(1) Data is stored in Compressed Sparse Column (CSC) format, with each column pre-sorted by feature values.

(2) The block structure requires only a single preprocessing step before training and is reused in subsequent iterations.

- **Algorithm Adaptation:**

(1) Exact Greedy Algorithm: Stores the full dataset in a single block.

(2) Approximate Algorithm: Partitions data into multiple blocks

Column Block for Parallel Learning

- **Time Complexity Optimization:**

Exact Greedy Algorithm:

Original space-aware algorithm: $O(Kd\|\mathbf{x}\|_0 \log n)$

Optimized: $O(Kd\|\mathbf{x}\|_0 + \|\mathbf{x}\|_0 \log n)$

Approximate Algorithm:

Original binary search: $O(Kd\|\mathbf{x}\|_0 \log q)$

(*q* = number of candidates, typically 32–100).

Optimized: $O(Kd\|\mathbf{x}\|_0 + \|\mathbf{x}\|_0 \log B)$

(eliminates the *logq* factor; *B* = block size).

Cache-aware Access

- **Problem:** Direct read/write dependencies lead to performance degradation.

- **For the exact greedy algorithm:**

Allocate an internal buffer for each thread to prefetch gradient statistics in batches.

- **For the approximate algorithm:**

Define the block size as the max-number of examples contained in a block.

Too small blocks: Low thread workload and poor parallel efficiency.

Too large blocks: Gradient statistics exceed cache capacity, causing cache misses.

Blocks for Out-of-core Computation

- **Objective:**

Utilize disk space beyond memory capacity to process data exceeding memory limits, enabling scalable learning.

- **Block Compression:**

Blocks are compressed column-wise and decompressed on-the-fly by an independent thread when loaded into main memory.

- **For row indices:**

Subtract the row index by the starting index of the block. Store each offset using a 16-bit integer. Achieves a compression ratio of approximately 26% to 29%.

END TO END EVALUATIONS

END TO END EVALUATIONS

- **DataSets:**

Dataset	*n*	*m*	Task
Allstate	10 M	4227	Insurance claim classification
Higgs Boson	10 M	28	Event classification
Yahoo LTRC	473K	700	Learning to Rank
Criteo	1.7 B	67	Click through rate prediction

END TO END EVALUATIONS

- **Classification:**

Method	Time per Tree (sec)	Test AUC
XGBoost	0.6328	0.8124
scikit-learn	28.51	0.8302
R.gbm	1.032	0.6224

END TO END EVALUATIONS

- **Learning to Rank:**

Method	Time per Tree (sec)	NDCG@10
XGBoost	0.813	0.7839
pGBRT [22]	2.576	0.7915

- **Single-machine scenario:** XGBoost is significantly faster than competitors in classification and ranking tasks.
- **Out-of-core scenario:** Compression and sharding techniques enable single machines to handle large-scale data efficiently.