Why xgboost is so fast?

Yafei Zhang

kimmyzhang@tencent.com

August 1, 2016

Outline

Goal

A training set $\mathcal{D} = \{(x_i, y_i)\}_1^N$. A loss function L. The model F.

Objective Function

$$\mathcal{L} = \underbrace{\sum_{i=1}^{N} L(y_i, F(x_i; w))}_{Training\ \ loss} + \underbrace{\sum_{k=1}^{K} \Omega(f_k)}_{Regularization} \tag{1}$$

where

$$F(x; w) = \sum_{k=0}^{K} f_k(x; w_k) \tag{2}$$

Goal: Learn F Greedily

$$F^* = \arg\min_F \mathcal{L} \tag{3}$$

Tree Splitting Algorithm

At iteration k, f_k is wanted

$$
\begin{aligned}
L(y_i, F_k(x_i; w)) &= L(y_i, F_{k-1}(x_i; w) + f_k(x_i)) && (4)\\
\approx L(y_i, F_{k-1}(x_i; w)) &+ \underbrace{\frac{\partial L(y_i, F_{k-1}(x_i; w))}{\partial F_{k-1}}}_{:=g_i} f_k(x_i) \\
&+ \frac{1}{2}\underbrace{\frac{\partial^2 L(y_i, F_{k-1}(x_i; w))}{\partial F_{k-1}^2}}_{:=h_i} f_k^2(x_i) && (5)\\
&= L(y_i, F_{k-1}(x_i; w)) + g_i f_k(x_i) + \frac{1}{2} h_i f_k^2(x_i) && (6)
\end{aligned}
$$

where

$$F_k = \sum_{j=1}^{k} f_j \quad (7)$$

Tree Splitting Algorithm

Overall Loss w.r.t. f_k

$$\mathcal{L}(f_k) = \sum_{i=1}^{N} \left[g_i f_k(x_i) + \frac{1}{2} h_i f_k^2(x_i) \right] + \Omega(f_k) \tag{8}$$

$\Omega()$ for a decision tree f

$$\Omega(f) = \frac{\gamma}{2} J + \frac{\lambda}{2} \sum_{j=1}^{J} b_j^2 \tag{9}$$

γ, λ: regularization coefficient.
J: number of leaf nodes in f.
R_j: regions of leaf nodes in f.
b_j: values of R_j.

Tree Splitting Algorithm

With $\{R_j\}_1^J$ known, we are to optimize $\{b_j\}_1^J$.
How to get $\{R_j\}_1^J$ will be described later.

The optimal leaf values are given by minimizing $\mathcal{L}$

$$\mathcal{L}(f_k) = \mathcal{L}(\{b_j\}_1^J, \{R_j\}_1^J) = \sum_{j=1}^{J} \left[\underbrace{\sum_{x_i \in R_j} g_i}_{:=G_j} b_j + \frac{1}{2} \left(\underbrace{\sum_{x_i \in R_j} h_i}_{:=H_j} + \lambda \right) b_j^2 \right] + \frac{\gamma}{2} J \tag{10}$$

To keep symbols uncluttered, J, b_j and R_j are parameters of f_k.

Tree Splitting Algorithm: Optimize b_j

The optimal leaf value of R_j

$$b_j^* = \arg\min_{b_j} \mathcal{L}(\{b_j\}_1^J, \{R_j\}_1^J) = -\frac{G_j}{H_j + \lambda} \tag{11}$$

The minimal loss function

$$\mathcal{L}(\{b_j^*\}_1^J, \{R_j\}_1^J) = -\frac{1}{2}\sum_{j=1}^{J} \frac{G_j^2}{H_j + \lambda} + \frac{\gamma}{2} J \tag{12}$$

Tree Splitting Algorithm: Get R_j

With $\{R_j\}_1^J$ and $\{b_j^*\}_1^J$, now we consider splitting R_i to R_L and R_R.

Define $\mathbf{R}$ and $\mathbf{R'}$: old and new splitting scheme

$$\mathbf{R} : \{R_j\}_1^J \tag{13}$$

$$\mathbf{R'} : \{R_j\}_{1, j \neq i}^J \cup \{R_L, R_R\} \tag{14}$$

Define $\mathbf{b}^*$ and $\mathbf{b}^{*\prime}$

$$\mathbf{b}^* : \{b_j^*\}_1^J \tag{15}$$

$$\mathbf{b}^{*\prime} : \{b_j^*\}_{1, j \neq i}^J \cup \{b_L^*, b_R^*\} \tag{16}$$

Tree Splitting Algorithm: Get R_j

Gain is defined to be the decrease in $\mathcal{L}$ after splitting R_i, and use (12).

Gain

$$Gain = 2\mathcal{L}(\mathbf{b}^*, \mathbf{R}) - 2\mathcal{L}(\mathbf{b}^{*'}, \mathbf{R}') \tag{17}$$

$$= \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma \tag{18}$$

3 Terms:

- Gain from splitting
- Gain from not splitting
- The complexity introduced by splitting

Tree Splitting Algorithm: Put all together

From (18), we have the algorithm

- Calculate g_i and h_i for all instances.
- Carry out split finding algorithm to get some proposed splits.
  - Iterate over all proposed splits.
    - Calculate gain according to (18).
- Select the "best" split, which results in the maximal gain.
- Stop splitting if the maximal gain is negative or some criterions are met.

GBDT to tree splitting algorithm to split finding algorithm.
Split finding algorithm will be described next section.

Outline

Split Finding Algorithm

Trivial for binary features

Only one split. We can skip this section:)

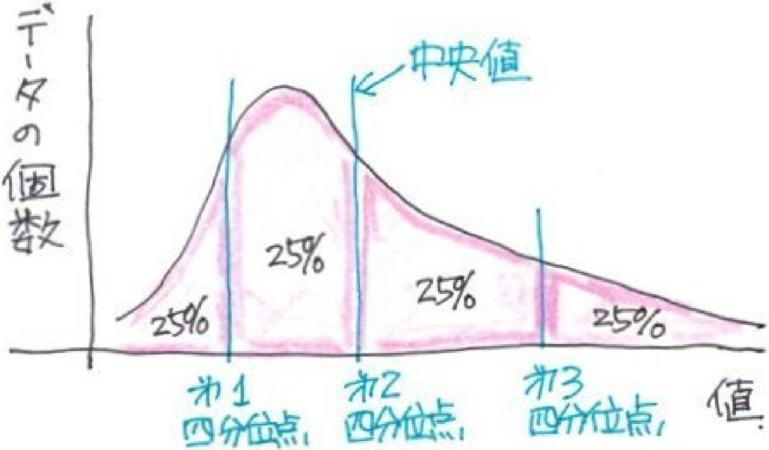
How to find proposed splits?

- Exact Greedy Algorithm
  - Enumerate over all possible splits by brute force.
- Approximate Algorithm
  - Propose percentiles on one feature.
    - GK01(M. Greenwald & S. Khanna, 2001, Space-Efficient Online Computation of Quantile Summaries).
    - GK01's variations and extensions.

In the approximate algorithm, can we reuse the proposed splits?

- Global: proposal will be carried out per tree.
- Local: proposal will be carried out per split.

A Percentiles Demo



from Internet

Exact Greedy Algorithm

Algorithm 1: Exact Greedy Algorithm for Split Finding

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

Time complexity

$O(N_u)$: N_u is the number of unique values of this feature.

Approximate Algorithm

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ ***to*** m **do**

  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.

  Proposal can be done per tree (global), or per split(local).

end

for $k = 1$ ***to*** m **do**

  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$

  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

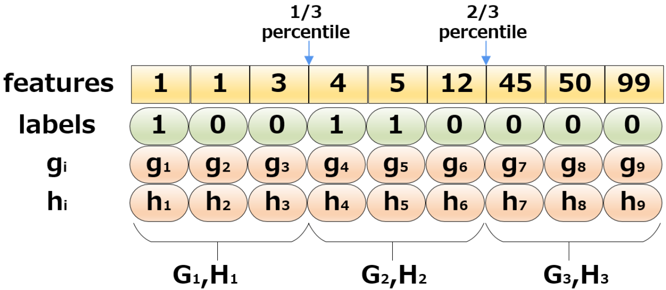
end

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

Time complexity

$O(N_p)$: N_p is the number of proposed splits of this feature.

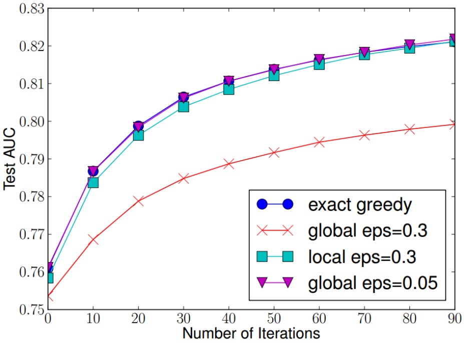
Approximate Algorithm: Demo



$$Gain = \max\{Gain, \frac{G_1^2}{H_1+\lambda} + \frac{G_{23}^2}{H_{23}+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma, \tag{19}$$

$$\frac{G_{12}^2}{H_{12}+\lambda} + \frac{G_3^2}{H_3+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma\} \tag{20}$$

Split Finding Algorithm: Conclusions



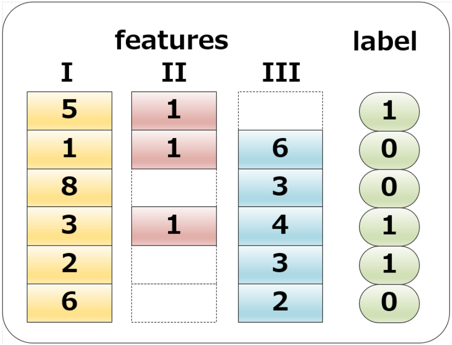
eps: bucket width, $1/eps$: number of percentile buckets.

Conclusions

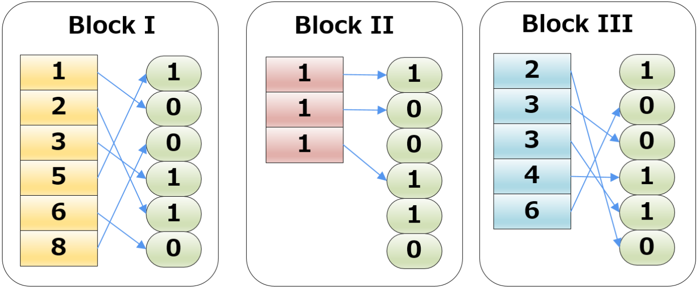
Approximate approach can be as well as the exact one.

Outline

Column Blocks and Parallelization



Column Blocks and Parallelization



- Feature values are sorted.
- A block contains one or more feature values.
- Instance indices are stored in blocks.
- Missing features are not stored.
- With column blocks, a parallel split finding algorithm is easy to design.

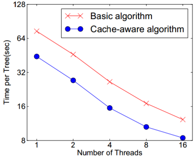
Outline

Cache Aware Access

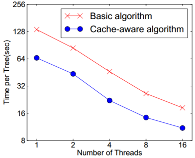
Tricks

- A thread pre-fetches data from non-continuous memory into a continuous buffer.
- The main thread accumulates gradients statistics in the continuous buffer.

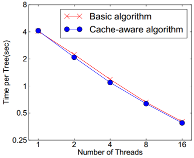
Cache Aware Access: Conclusions



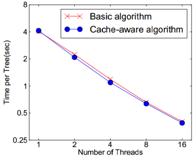
(a) Allstate 10M



(b) Higgs 10M



(c) Allstate 1M



(d) Higgs 1M

Conclusions

- Cache aware algorithm is always no worse.

Outline

System Tricks

- Block pre-fetching.
- Utilize multiple disks to parallelize disk operations.
- LZ4 compression(popular recent years for outstanding performance).
- Unrolling loops.
- OpenMP.

LZ4 Compression

Compress blocks with LZ4.

Compressor	Ratio	Compression	Decompression
memcpy	1	4200 MB/s	4200 MB/s
LZ4 fast 17 (r129)	1.607	690 MB/s	2220 MB/s
LZ4 default (r129)	2.101	385 MB/s	1850 MB/s
LZO 2.06	2.108	350 MB/s	510 MB/s
QuickLZ 1.5.1.b6	2.238	320 MB/s	380 MB/s
Snappy 1.1.0	2.091	250 MB/s	960 MB/s
LZF v3.6	2.073	175 MB/s	500 MB/s
zlib 1.2.8 -1	2.73	59 MB/s	250 MB/s
LZ4 HC (r129)	2.72	22 MB/s	1830 MB/s
zlib 1.2.8 -6	3.099	18 MB/s	270 MB/s

Unrolling Loops: Program I

```c
#include <stdio.h>
int main() {
  int sum = 0;
  int i;
  for (i = 0; i < 100000000; i++) {
    sum += i;
  }
  printf("%d\n", sum);
  return 0;
}
```

Unrolling Loops: Program II

```c
#include <stdio.h>
int main() {
  int sum = 0;
  int i;
  for (i = 0; i < 100000000; i += 8) {
    sum += i;
    sum += i + 1;
    sum += i + 2;
    sum += i + 3;
    ...
    sum += i + 7;
  }
  printf("%d\n", sum);
  return 0;
}
```

Unrolling Loops

Conclusions

Program II is 40%-50% faster.
Performance can benefit from unrolling loops with light loop bodies.

A simple solution

CFLAGS=-funroll-loops
CXXFLAGS=-funroll-loops
Use flags above to compile Program I.

OpenMP

A shared-memory parallel programming primitive.

```
#pragma omp parallel for
for (int i = 0; i < 1024; i++) {
  // do something in parallel
}
```

Not always faster, programs need to be tuned carefully.

CFLAGS=-fopenmp
CXXFLAGS=-fopenmp
LDFLAGS=-fopenmp

Outline

Summary

- GBDT recall
  - tree splitting algorithm: f_k, b_j, R_j
- Split finding algorithm
  - exact vs approximate
  - global vs local
- Column blocks and parallelization
  - one block = one or multiple features
  - sorted feature value
  - easy to parallelize
- Cache aware access
  - pre-fetch
  - alleviate non-continuous memory access
- System Tricks
  - pre-fetch
  - utilize multiple disks
  - LZ4 compression
  - Unrolling loops
  - OpenMP

Outline

Reference

J. Friedman(1999). Greedy Function Approximation: A Gradient Boosting Machine.

J. Friedman(1999). Stochastic Gradient Boosting.

T. Hastie, R. Tibshirani, J. Friedman(2008). Chapter 10 of The Elements of Statistical Learning(2e).

T. Chen, C. Guestrin(2016). XGBoost: A Scalable Tree Boosting System.