XGBoost: A Scalable Tree Boosting System

Tianqi Chen, Carlos Guestrin

KDD '16: Proceedings of the 22nd ACM SIGKDD
International Conference on Knowledge Discovery and Data Mining

August 2016 ,Pages 785–794

송영석

Financial Engineering Lab

Department of Industrial Engineering

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

CONTENTS

1. Introduction

2. Tree Boosting in a nutshell

3. Split Finding Algorithms

4. System Design

5. Conclusion

0. Background

(1) Decision Tree

분류 규칙을 통해 데이터를 분류,회귀하는 지도 학습 모델

(2) Bagging (Bootstrap aggregation)

샘플을 여러 번 뽑고 각 모델을 학습시켜 결과물을 집계

(3) Random Forest

Bagging의 일종으로 설명변수를 무작위로 하여 상관관계를 줄이고 결과물을 집계(평균내거나 다수결)

0. Background

(4) Boosting

Bagging은 독립적인 결과를 합한다면 Boosting은 예측한 결과가 다음 예측에 가중치를 부여한다

(5) Gradient Boosting

loss function을 줄이는 방향의 negative gradient를 얻고,

이를 활용해 boosting을 하는 것이기 때문에 gradient descent와 boosting

문제 : 느리다 , 과적합(Overfitting)

1. Introduction

(1) What is XGBOOST (eXtreme Gradient Boosting)?

- A **scalable** tree boosting system

(store sales prediction, web text classification, customer behavior prediction, motion detection, ad click through rate prediction, product categorization, hazard risk prediction…)

- 단일 시스템에서 사용한 기존 일반적 솔루션보다 10배 이상 빠르게 실행
- 분산 또는 메모리가 제한된 상황에서도 수십억 가지의 예제로 확장

XGBoost의 Scalability는 중요한 시스템과 알고리즘의 최적화 덕이다.

(Gradient boosting이 base이다.)

1. Introduction

(2) XGBOOST (eXtreme Gradient Boosting) in competitions

- Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015**, 17 solutions used XGBoost**.

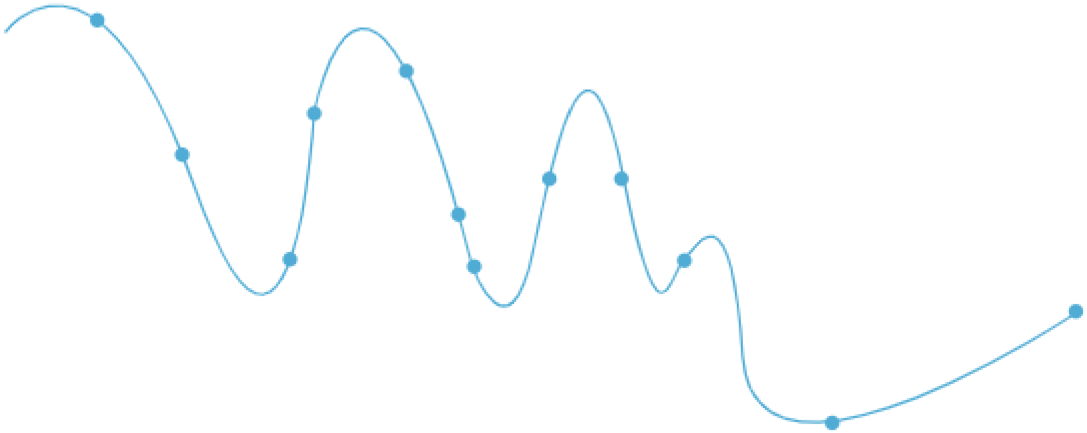
(deep neural nets, was used in 11 solutions.)

- In KDDCup 2015, where XGBoost was used by every winning team **in the top-10**.

2. Tree Boosting in a nutshell

Everything from Chapter 2…

Prevent Overfitting



2. Tree Boosting in a nutshell

Tree Boosting 기본모양

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F}$$

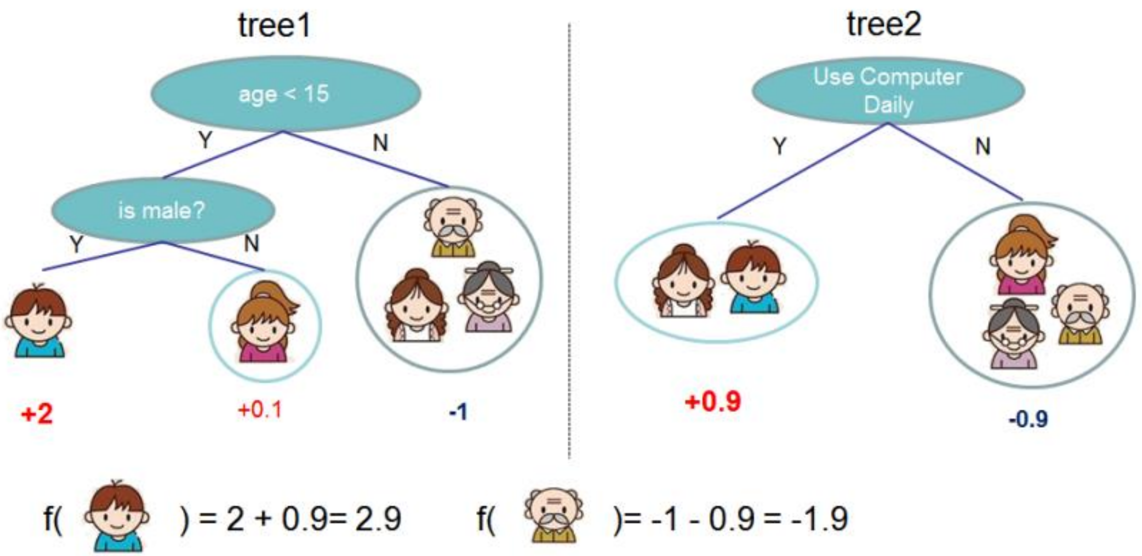
$$= f_1(x_i) + f_2(x_i) + f_3(x_i) + \cdots + f_K(x_i)$$

- $\phi(\mathbf{x}_i)$ 는 Tree boosting 전체를 의미한다. (결과를 의미)
- 각각 $f_K(x_i)$ 는 개별적인 Tree 를 의미한다.

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\} \ (|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R})$$

- 주어진 Data : n개의 Example 과 m 개의 Featrue

2. Tree Boosting in a nutshell



Function f에 input x 를 넣었을 때 나오는 결과가 Tree q(x)의 Node W

$$\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}(q : \mathbb{R}^m \rightarrow T, w \in \mathbb{R}^T)$$

q: Structure of each tree

T : Number of Leaves

W : Leaf Weights

2. Tree Boosting in a nutshell

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \boxed{\sum_k \Omega(f_k)}$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda\|w\|^2$$

→ Ω penalizes the complexity of the model

- $\sum_i l(\hat{y}_i, y_i)$ is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i.

- The additional regularization term helps to smooth the final learnt weights to avoid over-fitting

2. Tree Boosting in a nutshell

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F}$$

$$= f_1(x_i) + f_2(x_i) + f_3(x_i) + \cdots + f_K(x_i)$$

새로운 함수는 기존의 함수 집합에 더해졌을때, Loss Function이 최소가 되는 함수

$$\mathcal{L}(\boxed{\phi}) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda\|w\|^2$$

그런데 Loss function을 보니

Loss function이 Phi 값으로 Function을 사용하여
Euclidean space를 활용하여 optimization이 불가능

따라서 Additive 특성을 이용하여 식 변형

2. Tree Boosting in a nutshell

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \underline{\hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)}) + \Omega(f_t)$$

$$\sum_{i} l(\hat{y}_i, y_i)$$

원래 모양

t번째 Prediction t-1번째 Prediction Added Function

$$\hat{y}_i^t = \textstyle\sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{t-1} + f_t(x_i)$$

Taylor Series는 미지의 함수 f(x)를 다항함수로 근사하여 이해할 수 있는 깔끔한 표현식으로 만들어준다.

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a)$$

$$f(x+\Delta x) \simeq f(x) + f'(x)\Delta x + \tfrac{1}{2}f''(x)\Delta x^2$$

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$

2. Tree Boosting in a nutshell

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n}[\cancel{l(y_i, \hat{y}^{(t-1)})} + g_i f_t(\mathbf{x}_i) + \frac{1}{2}h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

미분할거니깐
constant 제외

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n}[g_i f_t(\mathbf{x}_i) + \frac{1}{2}h_i f_t^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$$

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

$I_j = \{i|q(\mathbf{x}_i) = j\}$ as the instance set of leaf j.

T : Leaf 개수, n=instances
Tree의 관점으로 Loss계산을 위해 T로 합

2. Tree Boosting in a nutshell

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n}[g_i f_t(\mathbf{x}_i) + \frac{1}{2}h_i f_t^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$$

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

$I_j = \{i|q(\mathbf{x}_i) = j\}$ as the instance set of leaf j

$$\sum_{i=1}^{n}[g_i f_t(\mathbf{x}_i) = \sum_{j=1}^{T}(\sum_{i\in I_j} g_i)w_j$$

*Remind
Function f 에 input x를 넣었을 때 나오는 결과, Node w

$$\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}$$

- 좌변은 f(x) 는 regression tree 거친 결과이고, 우변은 leaf의 가중치인 w_j 를 이용해 구한 결과값의 합의 합

2. Tree Boosting in a nutshell

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

Tree q(x)가 고정일때,
w_j에 대하여 미분하여 Leaf의
Optimal Weight, w_j를 구함

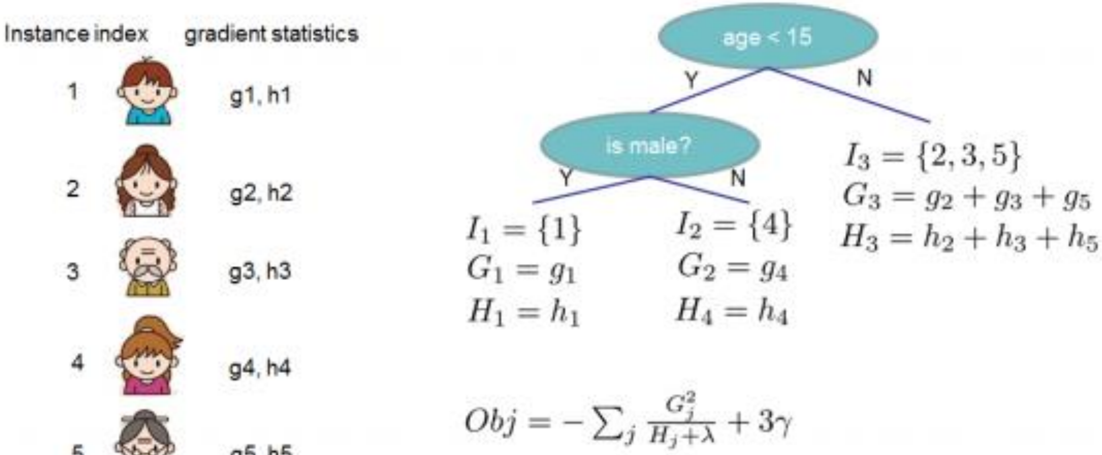
$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda}$$

Optimal W를 loss objective function에 넣었을 때의 식을 통해 첫 항이 (-) 이므로 값이 커지면 Loss 감소

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

Tree structure의 성능 척도

2. Tree Boosting in a nutshell



$$Obj = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

Structure Score of tree q

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

Optimal leaf weight

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

2. Tree Boosting in a nutshell

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

트리 Split 기준을 정하여, 최적의 트리를 만들어야 한다.

하나의 split을 진행하였을때 왼쪽과 오른쪽에서 얻은 점수와 분할되기 전의 점수를 계산하여 Gain을 구하듯이 Loss Reduction을 구함.

감소폭이 큰 분할을 선택한다!

2. Tree Boosting in a nutshell

2.3 Shrinkage and Column Subsampling

Prevent overfitting

Shrinkage

Tree에 η (Shrinkage factor)를 곱하여 Tree의 영향력을 낮춘다.
미래에 더해지는 Tree의 영향력이 모델을 잘 개선 할 수 있게 해준다.

Subsampling

Random Forest에서 사용 되어온 기법
Feature를 일부만 사용하여 over-fitting을 막고 계산 속도를 올려준다.

3. SPLIT FINDING ALGORITHMS

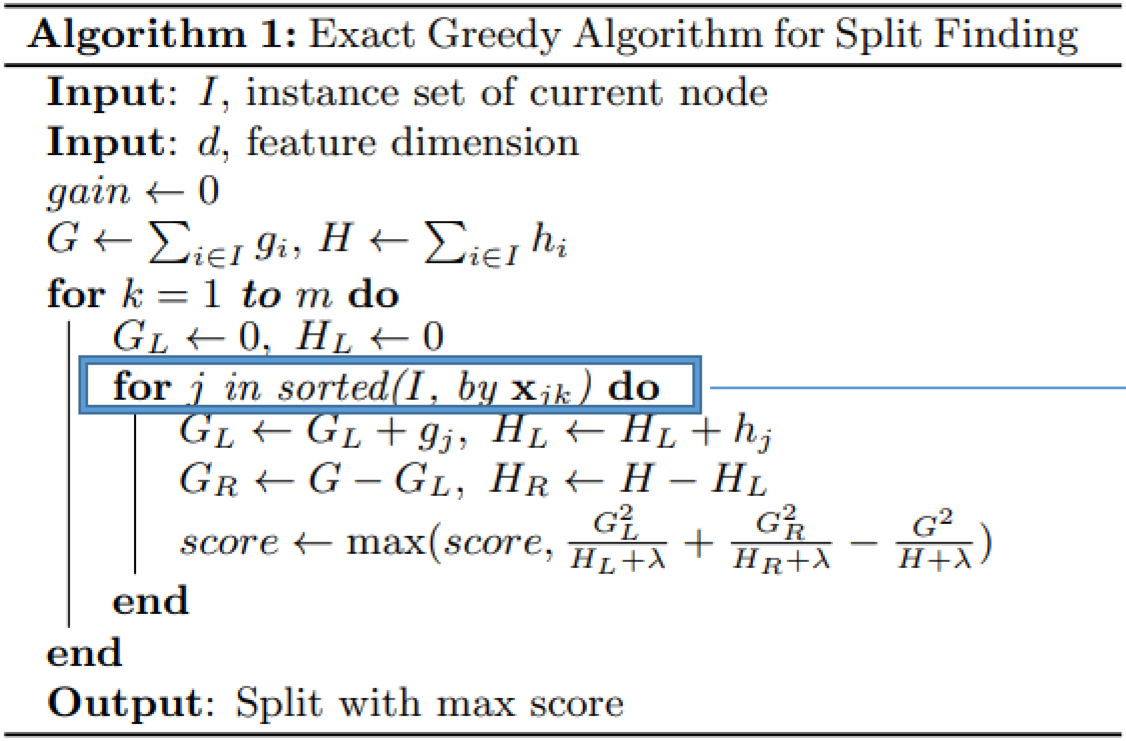
속도를 높이자!

Handling
Missing data

3. SPLIT FINDING ALGORITHMS

3.1 Basic Exact Greedy Algorithm

모든 경우의 수를 다 탐색하여
Feature에 대해 split을 찾는 과정!!



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
```

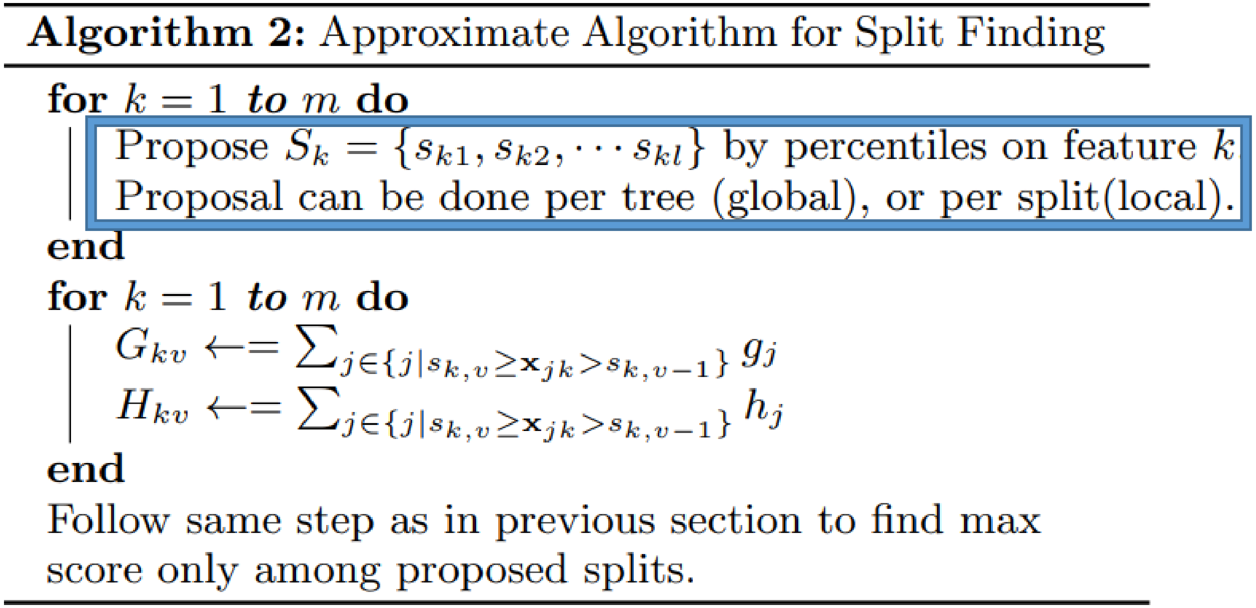
모든 경우의 수를 탐색
한다.

최적해를 보장하지만
시간이 많이 걸립니다...

3. SPLIT FINDING ALGORITHMS

3.2 Approximate Algorithm

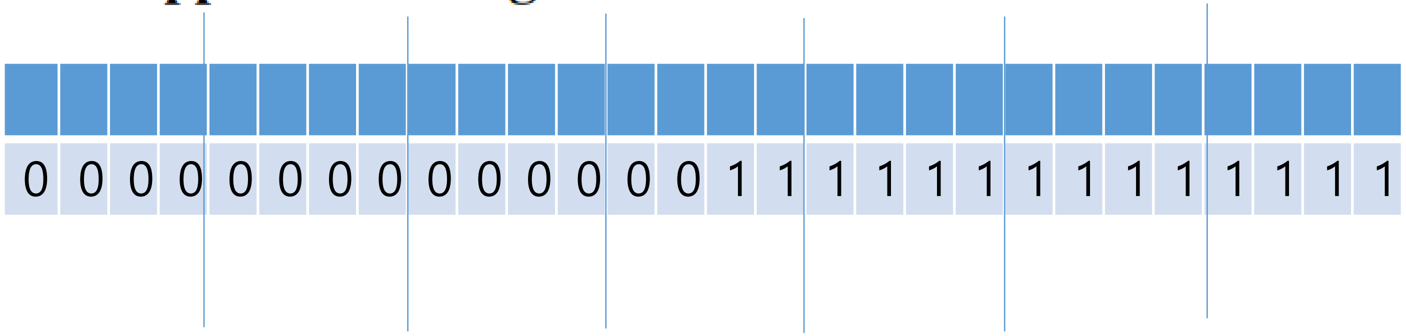
속도를 높이자!



```
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.
```

3. SPLIT FINDING ALGORITHMS

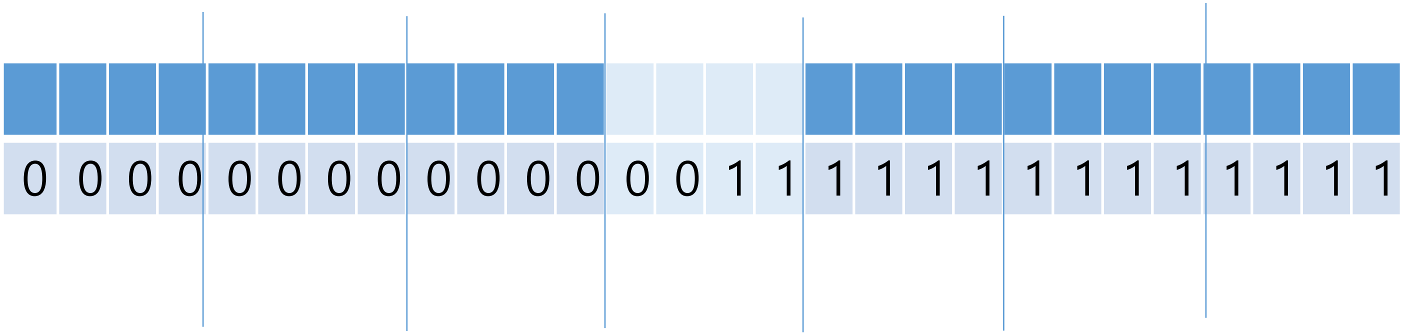
3.2 Approximate Algorithm



0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Greedy
= 27회 확인

- Greedy 의 경우 27번의 split을 전부 확인하여 Best를 찾음.
- 7개의 Buckets 으로 나누었습니다.



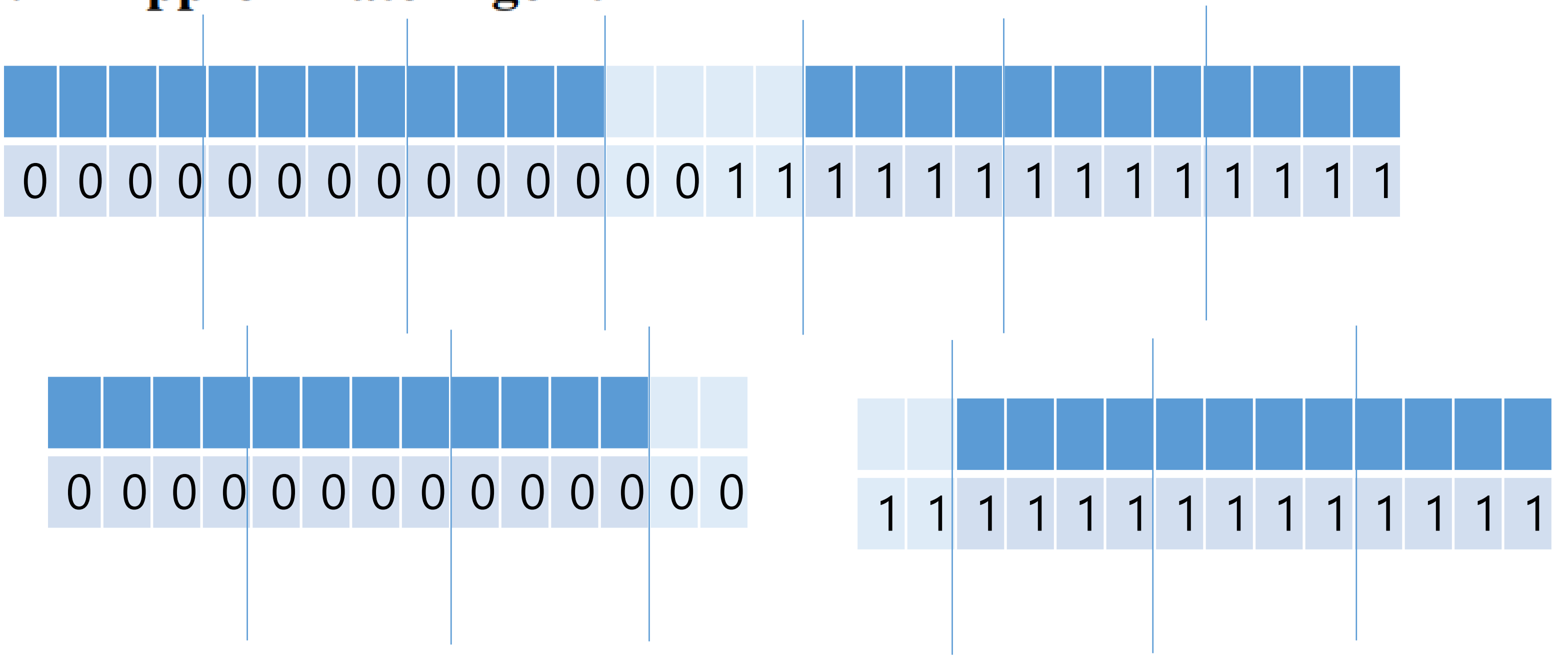
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Approximation
= 3번을 7번
= 21회 확인

- Approximate의 경우 각각의 Bucket에 대해 gradient를 따로 구하여 Best split을 찾음
- Bucket들을 thread 1,2,3...에 각각 병렬로 계산하여 시간이 단축

3. SPLIT FINDING ALGORITHMS

3.2 Approximate Algorithm



0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1

Global variant(per Tree)

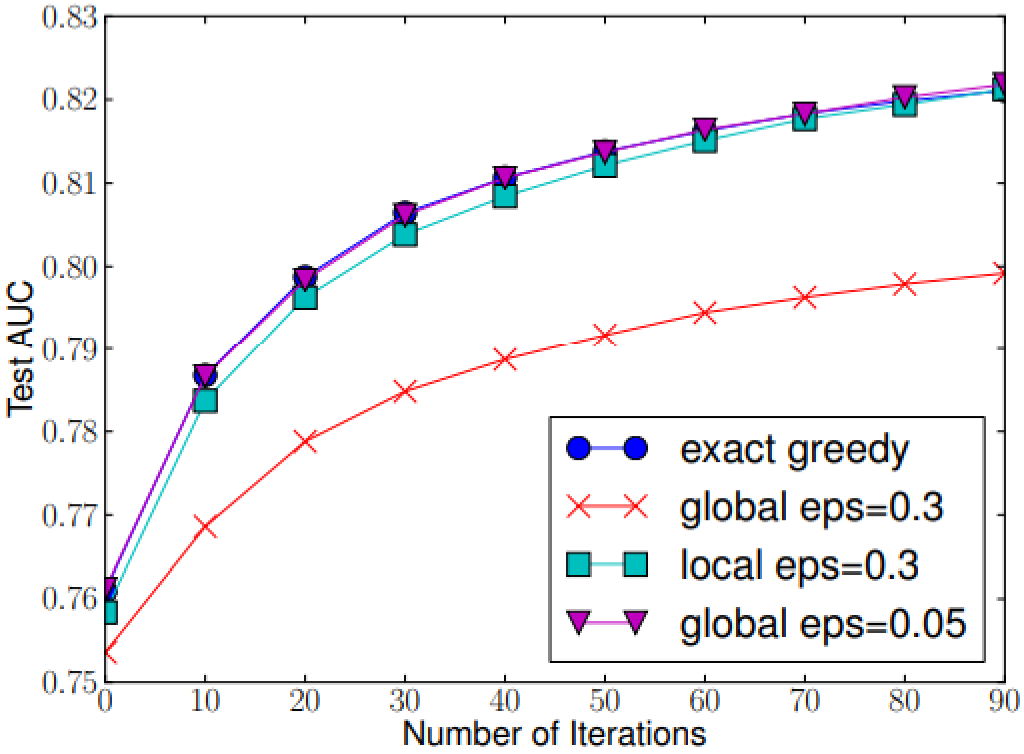
- Bucket 기준을 동일하게 유지 나눠지면 Bucket이 4개가 되는 것
- Tree depth가 깊어질수록 탐색해야 하는 Bucket 개수가 줄어든다.

Local variant(per split)

- Bucket size를 고정하여 사용 분리 전에 7개면 후에도 7개가 된다.
- Tree depth가 깊어질수록 Bucket 안의 example 개수가 줄어든다.

3. SPLIT FINDING ALGORITHMS

3.2 Approximate Algorithm



$\frac{1}{\epsilon}$개가 Bucket 개수

Global variant를 사용할 때는 local variant 사용할 때보다 bucket 개수를 많게 해야 한다.

3. SPLIT FINDING ALGORITHMS

3.3 Weighted Quantile Sketch

Sketch Algorithm : 표본 데이터로 Sketch를 하여 Original 데이터의 분포 파악

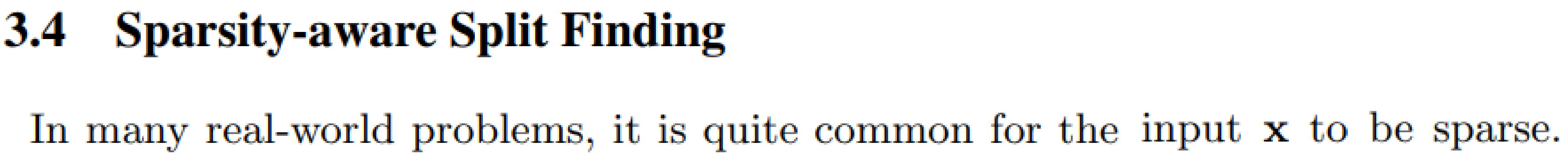
Quantile Sketch Algorithm : 각 quantile의 Sketch로 Original 데이터의 분포를 파악
(나눠서 각각 파악함)

Weighted Quantile Sketch Algorithm : 일반적으로 Qunatile Sketch Algorithm 이 각 Quantile의 데이터 개수가 같다면 Weighted quantile은 각 qunatile의 sum of weights 가 같다.

$$\sum_{i=1}^{n} \frac{1}{2} \boxed{h_i} (f_t(\mathbf{x}_i) - g_i/h_i)^2 + \Omega(f_t) + constant$$

h_i가 weight의 역할을 한다.

3. SPLIT FINDING ALGORITHMS



3.4 Sparsity-aware Split Finding

In many real-world problems, it is quite common for the input $\mathbf{x}$ to be sparse.

Sparsity의 원인

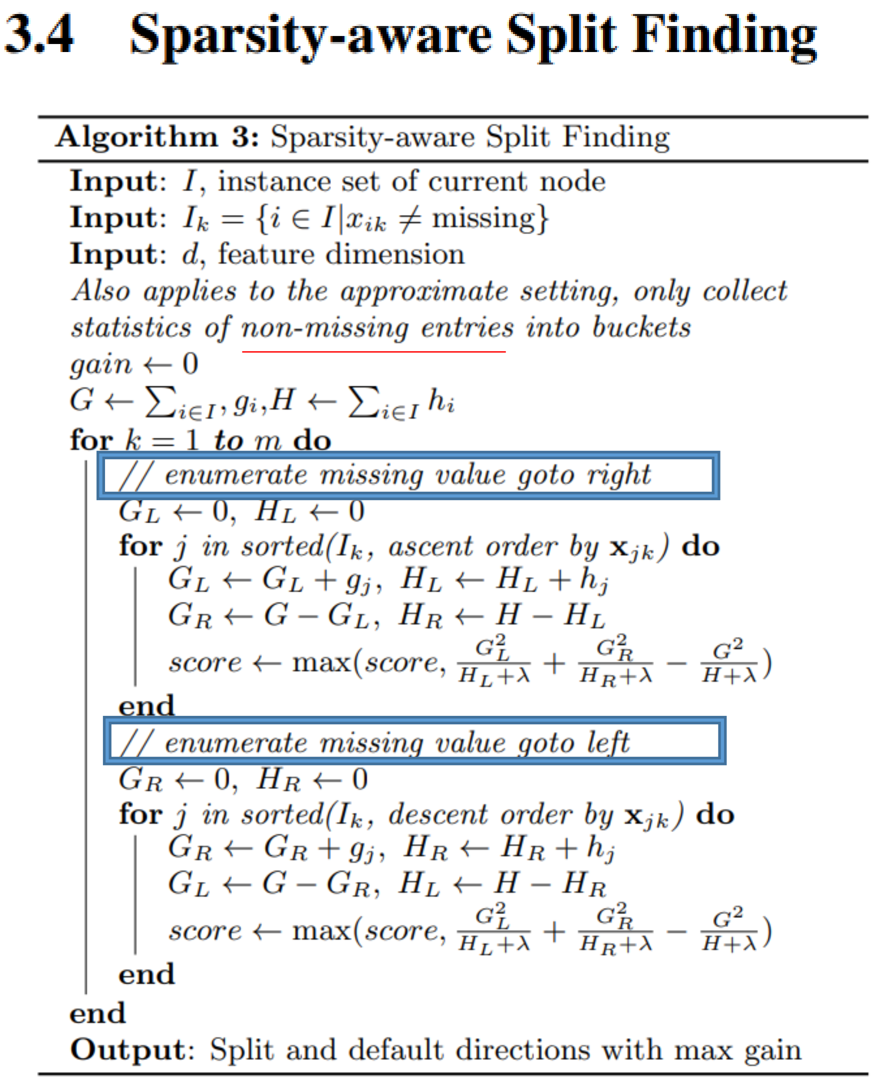
1. Presence of missing values in the data
2. Frequent zero entries in the statistics
3. Artifacts of feature engineering such as one-hot encoding

논문에서 제시한 해결책

Set the default direction in each tree node

Handling
Missing data

3. SPLIT FINDING ALGORITHMS



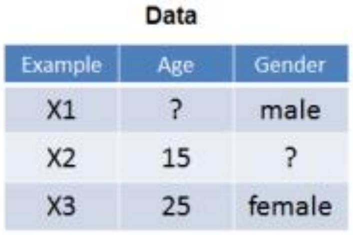
3.4 Sparsity-aware Split Finding

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
```

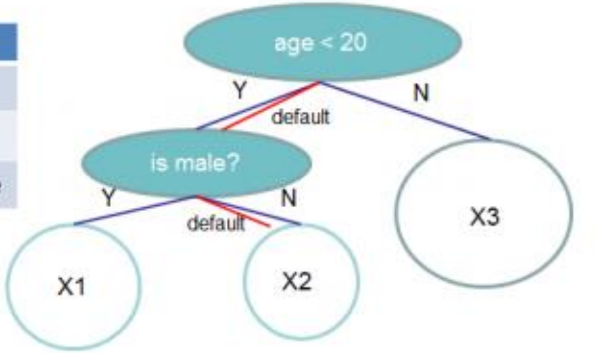
모든 missing value를 오른쪽 node로 이동하고 오름차순으로 정리한 뒤 Score를 계산하고 max를 통해 update

모든 missing value를 왼쪽 node로 이동하고 오름차순으로 정리한 뒤 Score를 계산하고 max를 통해 update



Data

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



3. SPLIT FINDING ALGORITHMS

3.4 Sparsity-aware Split Finding

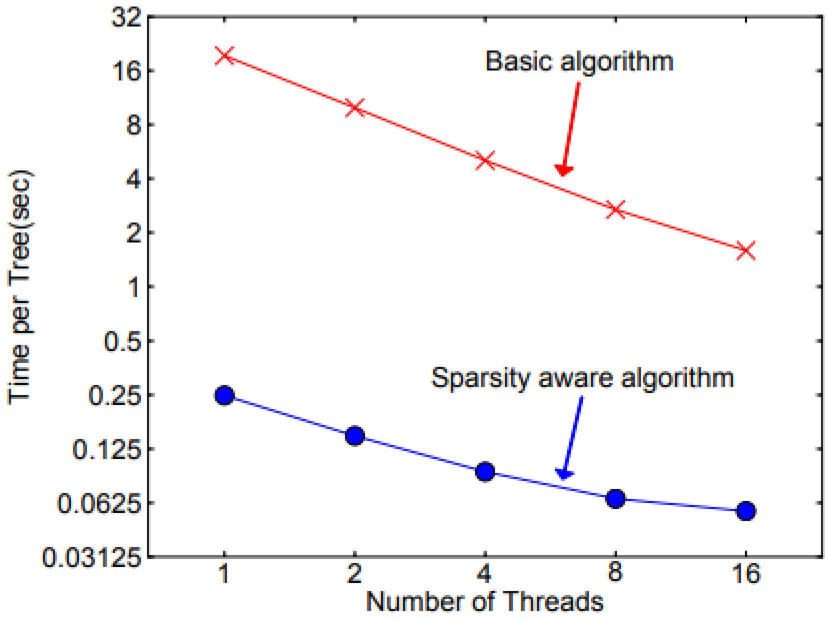
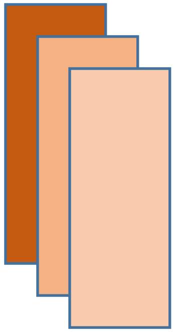
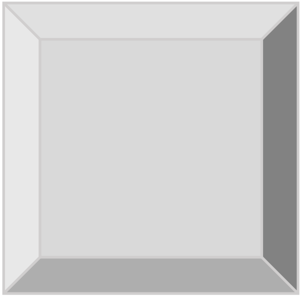


Figure 5: Impact of the sparsity aware algorithm on Allstate-10K. The dataset is sparse mainly due to one-hot encoding. The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

4. SYSTEM DESIGN 컴퓨터 공학 느낌의 파트...



Parallelized Building



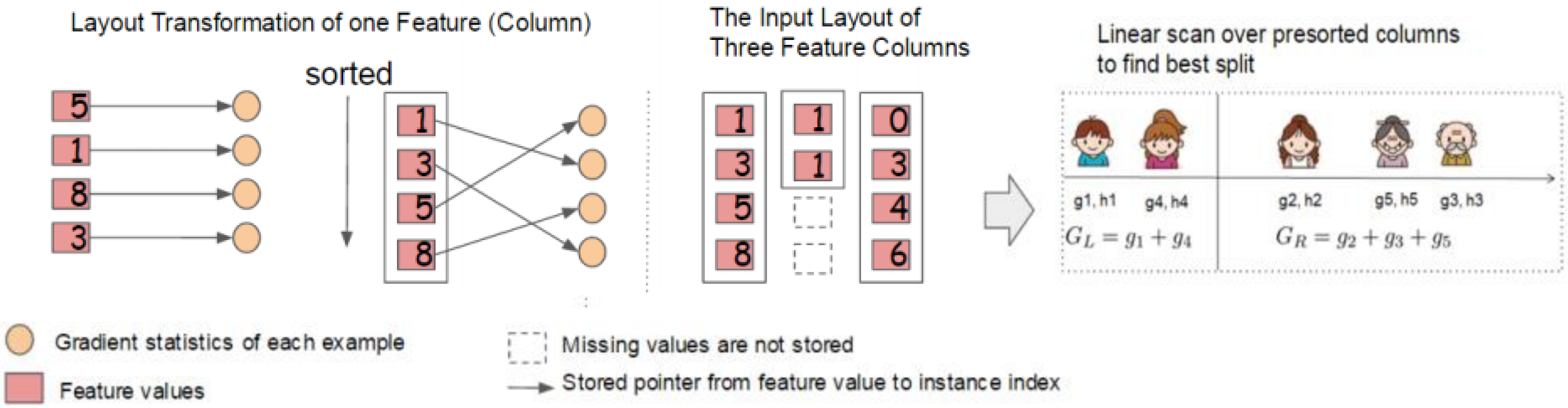
Cache Controll

4. SYSTEM DESIGN

4.1 Column Block for Parallel Learning

- The most time consuming part of tree learning is to get the data into sorted order.

- XGBoost propose to store the data in in-memory units, which we called block.
  -> Data 는 compressed column (CSC) format 으로 저장된다.
  (with each column sorted by the corresponding feature value)
  -> 이 layout은 training전에 한번만 계산되면 된다.

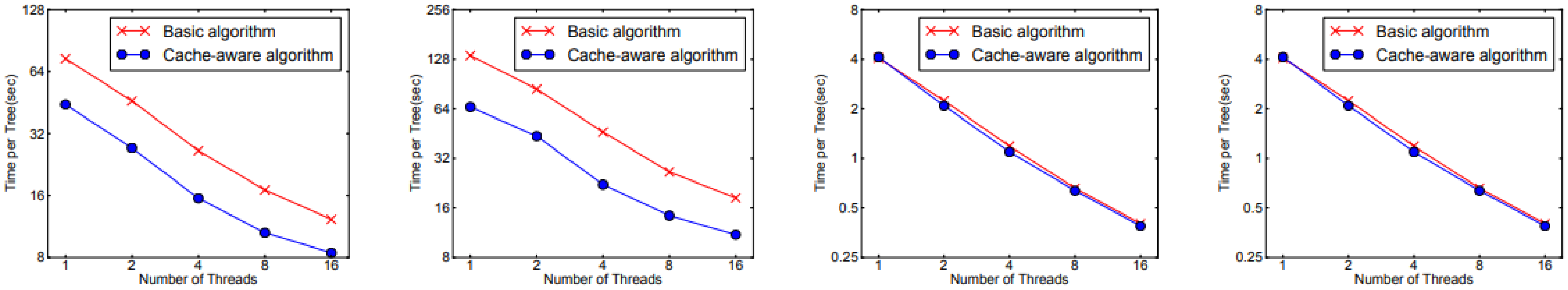

Layout Transformation of one Feature (Column)
sorted
5
1
8
3
1
3
5
8
The Input Layout of Three Feature Columns
1
3
5
8
1
1
0
3
4
6
Linear scan over presorted columns to find best split
g1, h1
g4, h4
g2, h2
g5, h5
g3, h3
$G_L = g_1 + g_4$
$G_R = g_2 + g_3 + g_5$
Gradient statistics of each example
Feature values
Missing values are not stored
Stored pointer from feature value to instance index

4. SYSTEM DESIGN

4.2 Cache-aware Access

I/O 장치 speed : CPU Cache > Memory > Disk

->가장 빠른 Cache로만 Gradient Statistics 계산을 할 수 있으면 속도 발전에 도움



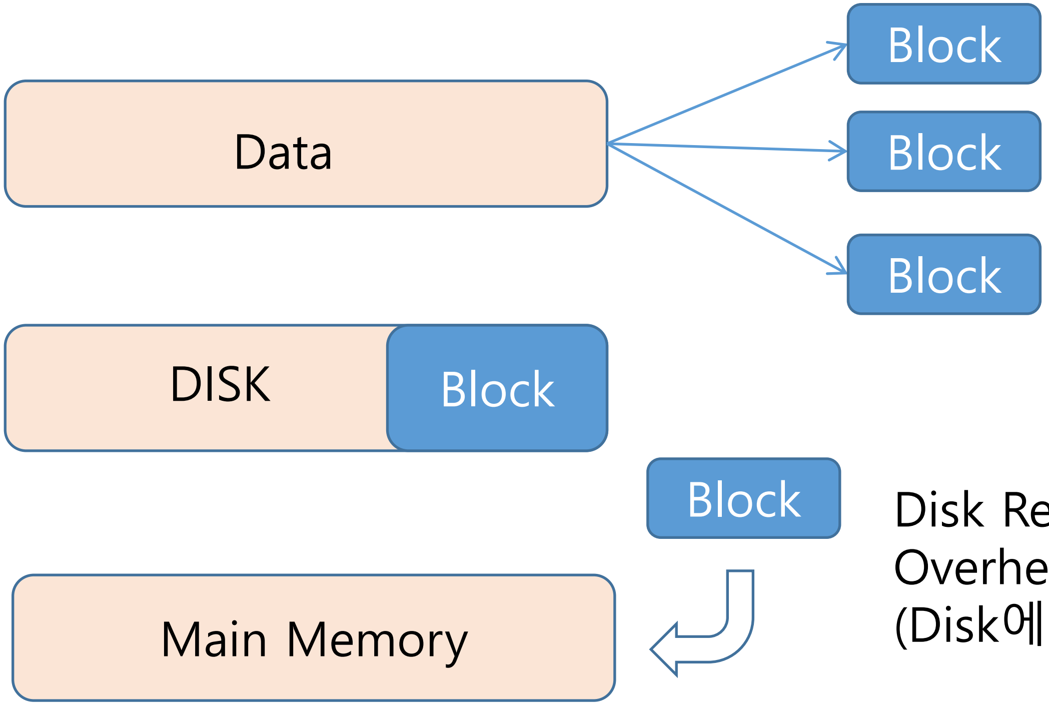
(a) Allstate 10M (b) Higgs 10M (c) Allstate 1M (d) Higgs 1M

Figure 7: Impact of cache-aware prefetching in exact greedy algorithm. We find that the cache-miss effect impacts the performance on the large datasets (10 million instances). Using cache aware prefetching improves the performance by factor of two when the dataset is large.

4. SYSTEM DESIGN

4.3 Blocks for Out-of-core Computation

Cache와 Memory 만으로 어려우면 Disk를 사용



Disk Reading에 많은 시간이 걸려 Overhead 발생
(Disk에서 불러오는 시간)

이를 Block Compression과 Block Sharding으로 해결

5. Conclusion

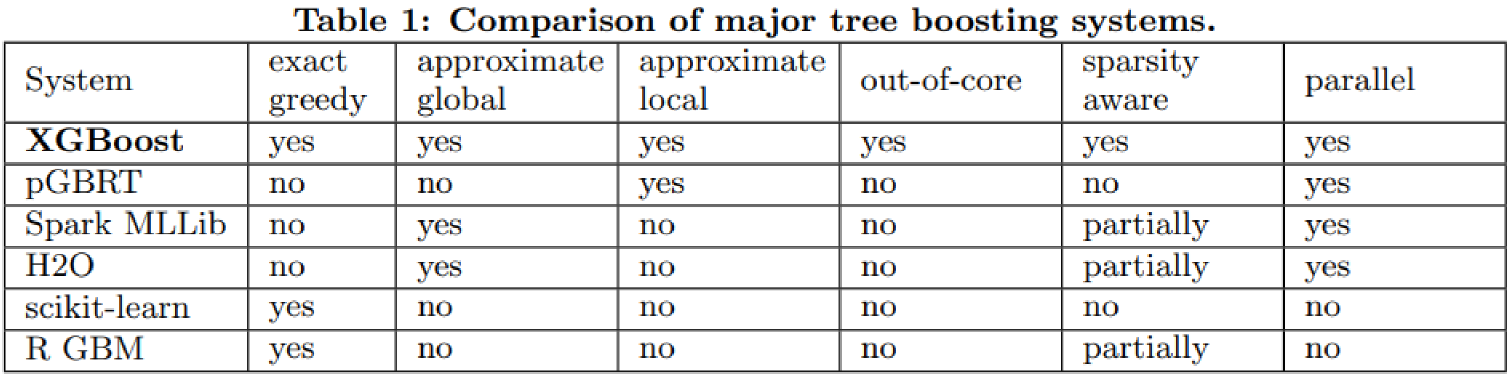


Table 1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLLib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

Gradient Boosting
(Additive Optimization in Functional Space)

Regularization (Prevent Overfitting)

Column sampling (Random forest)

Sparsity-Aware Learning

Parallel Tree Learning

Systemic Improvement

5. Conclusion

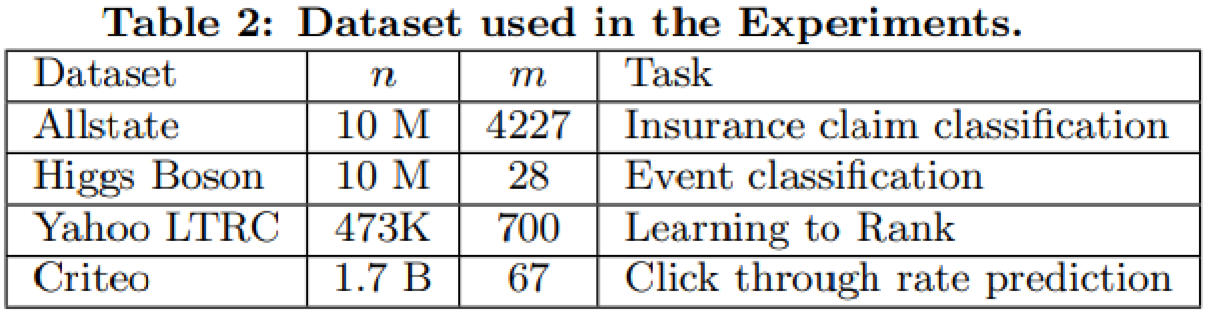


Table 2: Dataset used in the Experiments.

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

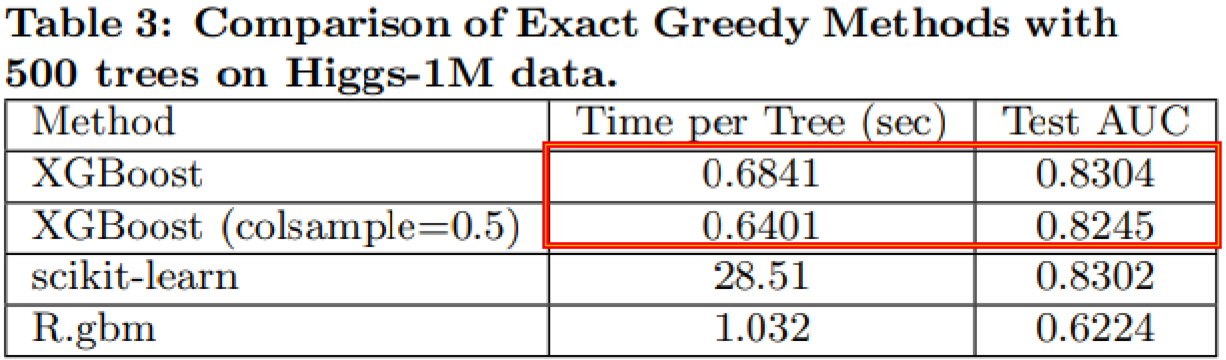


Table 3: Comparison of Exact Greedy Methods with 500 trees on Higgs-1M data.

Method	Time per Tree (sec)	Test AUC
XGBoost	0.6841	0.8304
XGBoost (colsample=0.5)	0.6401	0.8245
scikit-learn	28.51	0.8302
R.gbm	1.032	0.6224

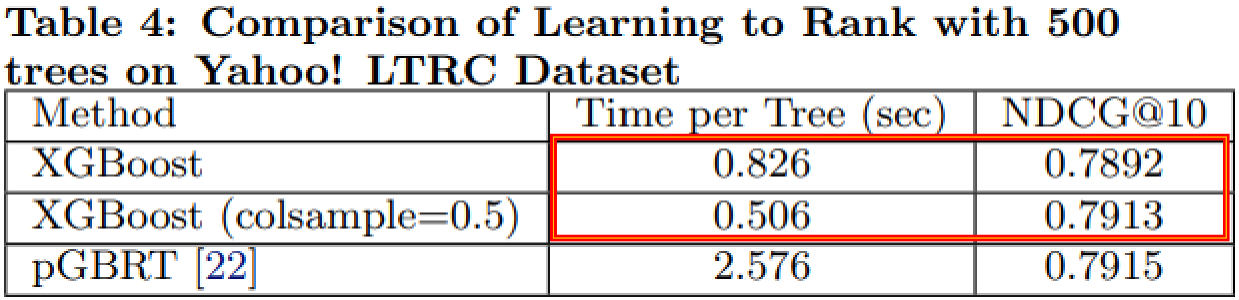


Table 4: Comparison of Learning to Rank with 500 trees on Yahoo! LTRC Dataset

Method	Time per Tree (sec)	NDCG@10
XGBoost	0.826	0.7892
XGBoost (colsample=0.5)	0.506	0.7913
pGBRT [22]	2.576	0.7915

5. Conclusion

pGBRT : Parallel boosted regression trees

여기서 XGBoost(exact greedy) 임에도 불구하고 시스템적으로 빠르다.

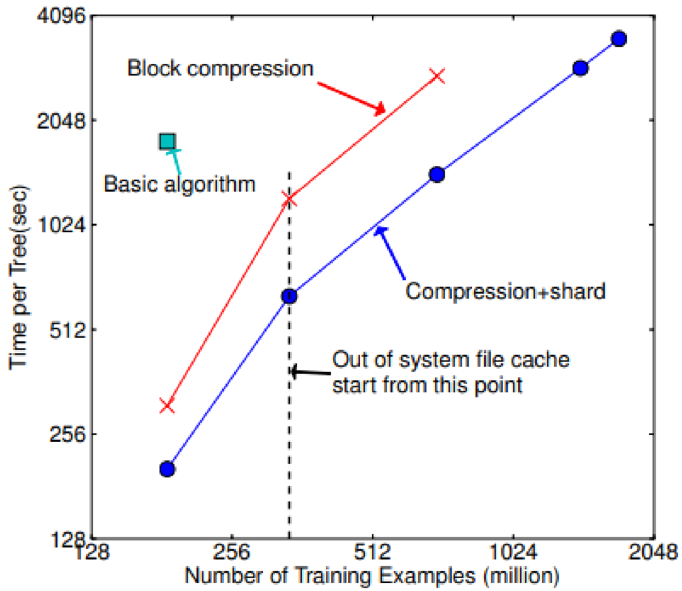


Figure 11: Comparison of out-of-core methods on different subsets of criteo data. The missing data points are due to out of disk space. We can find that basic algorithm can only handle 200M examples. Adding compression gives 3x speedup, and sharding into two disks gives another 2x speedup. The system runs out of file cache start from 400M examples. The algorithm really has to rely on disk after this point. The compression+shard method has a less dramatic slowdown when running out of file cache, and exhibits a linear trend afterwards.

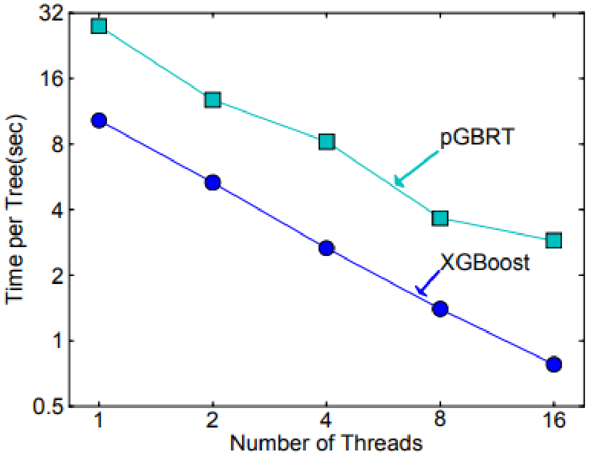
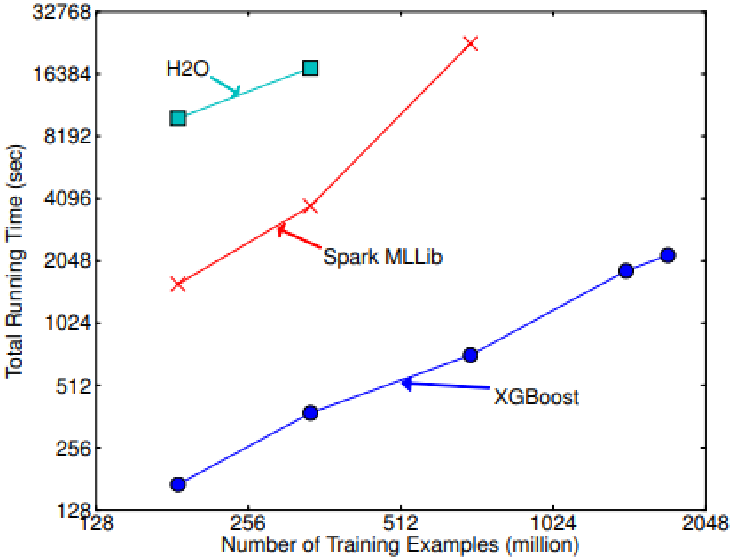


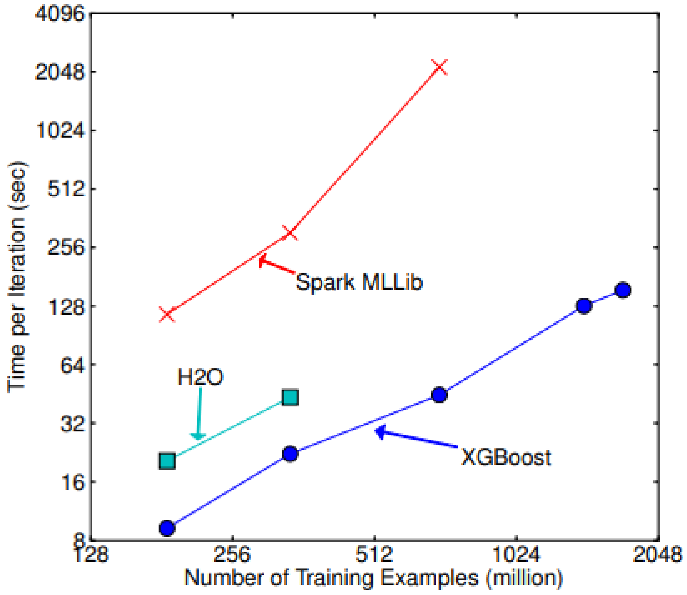
Figure 10: Comparison between XGBoost and pG
BRT on Yahoo LTRC dataset.

5. Conclusion

H2O : 유명한 머신러닝 라이브러리들이 모인 패키지



(a) End-to-end time cost include data loading



(b) Per iteration cost exclude data loading

Figure 12: Comparison of different distributed systems on 32 EC2 nodes for 10 iterations on different subset of criteo data. XGBoost runs more 10x than spark per iteration and 2.2x as H2O's optimized version (However, H2O is slow in loading the data, getting worse end-to-end time). Note that spark suffers from drastic slow down when running out of memory. XGBoost runs faster and scales smoothly to the full 1.7 billion examples with given resources by utilizing out-of-core computation.

UNIST

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY
2007

Thank you for listening!