# LightGBM: A Highly Efficient Gradient Boosting Decision Tree

Ke, Guolin, et al.

경상국립대학교 석사과정 오서영

# Abstract

GBDT(Gradient boosting decision tree)는 모든 가능한 분할 지점들의 information gain을 추정정하기 위해 모든 데이터 instance를 스캔할 필요가 있어, 많은 시간이 소모된다.
→ Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB)

1. GOSS를 사용하면 작은 gradient가 있는 데이터 instance의 상당 부분을 제외하고 나머지를 사용하여 information gain 을 추정한다.
2. EFB를 사용하면 feature의 수를 줄이기 위해 상호 배타적인 기능을 제공한다.
(i.e., 동시에 0이 아닌 값을 거의 사용하지 않음).

기존의 GBDT는 모든 feature에 대해 모든 데이터 instance를 스캔하여,
가능한 모든 분할 점의 정보 이득을 추정해야 한다.
→ Computational complexity는 feature수와 instance수들에 비례

# 2. GOSS

**Gradient-based One-Side Sampling (GOSS)**

- 더 큰 gradient (under-trained instances)가 있는 instanc는 information gain에 더 많이 기여함

→ 데이터 instanc를 샘플링할때, information gain 추정의 accuracy를 유지하려면, 해당 instanc의 gradien를 크게 유지해야하며, 작은 gradien를 가진 instanc를 무작위로 drop하면 된다.

그러나 데이터 분포가 변경되어 accuray를 손상시킬 수 있음

→ GOSS 는 모든 instanc를 큰 gradient로 유지하고, 작은 gradient가 있는 instance에 무작위 샘플링을 수행함

→ 데이터 분포에 대한 영향을 보상하기 위해, information gain을 계산할 때, 작은 gradient를 갖는 instance에 constant multiplier를 도입함.

# 2. GOSS

**Gradient-based One-Side Sampling (GOSS)**

1. 데이터 instance를 gradient의 절대값에 따라 정렬

2. 상위 a*100% instance를 선택

3. 나머지 데이터에서 b*100% instance를 무작위로 샘플링

4. information gain을 계산할 때 작은 gradient를 가진 데이터에 (1-a)/b를 곱해 증폭시킴

**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)

# 2. GOSS

**Theoretical Analysis**

- $\{g_1, \dots, g_n\}$ : 모델 output에 대한 loss function의 음의 gradient
- Decision tree 모델은 가장 큰 information gain으로 각 노드를 분할한다.

→ GDBT의 경우 INFORMATION GAIN은 일반적으로 분할 후 분산으로 측정됨

**Definition 3.1** *Let $O$ be the training dataset on a fixed node of the decision tree. The variance gain of splitting feature $j$ at point $d$ for this node is defined as*

$$V_{j|O}(d) = \frac{1}{n_O}\left(\frac{\left(\sum_{\{x_i \in O : x_{ij} \le d\}} g_i\right)^2}{n^j_{l|O}(d)} + \frac{\left(\sum_{\{x_i \in O : x_{ij} > d\}} g_i\right)^2}{n^j_{r|O}(d)}\right),$$

*where* $n_O = \sum I[x_i \in O]$, $n^j_{l|O}(d) = \sum I[x_i \in O : x_{ij} \le d]$ *and* $n^j_{r|O}(d) = \sum I[x_i \in O : x_{ij} > d]$.

# 2. GOSS

**Theoretical Analysis**
- Feature j에 대해, decision tree 모델은 $d_j^* = argmax_d V_j(d)$ 로 가장 큰 gain $V_j(d_j^*)$ 를 계산

A : 상위 a*100% instance, 큰 gradient
B : (1-a)*100% instance, 작은 gradient

$$\tilde{V}_j(d) = \frac{1}{n}\left(\frac{(\sum_{x_i \in A_l} g_i + \frac{1-a}{b}\sum_{x_i \in B_l} g_i)^2}{n_l^j(d)} + \frac{(\sum_{x_i \in A_r} g_i + \frac{1-a}{b}\sum_{x_i \in B_r} g_i)^2}{n_r^j(d)}\right), \quad (1)$$

where $A_l = \{x_i \in A : x_{ij} \leq d\}, A_r = \{x_i \in A : x_{ij} > d\}, B_l = \{x_i \in B : x_{ij} \leq d\}, B_r = \{x_i \in B : x_{ij} > d\}$, and the coefficient $\frac{1-a}{b}$ is used to normalize the sum of the gradients over $B$ back to the size of $A^c$.

# 2. GOSS

GOSS 는 train accuracy를 크게 잃게하지 않으며, 무작위 샘플링보다 우수한 성능을 가짐 :

**Definition 3.1** *Let $O$ be the training dataset on a fixed node of the decision tree. The variance gain of splitting feature $j$ at point $d$ for this node is defined as*

$$V_{j|O}(d) = \frac{1}{n_O}\left(\frac{(\sum_{\{x_i \in O: x_{ij} \le d\}} g_i)^2}{n^j_{l|O}(d)} + \frac{(\sum_{\{x_i \in O: x_{ij} > d\}} g_i)^2}{n^j_{r|O}(d)}\right),$$

*where* $n_O = \sum I[x_i \in O]$, $n^j_{l|O}(d) = \sum I[x_i \in O : x_{ij} \le d]$ *and* $n^j_{r|O}(d) = \sum I[x_i \in O : x_{ij} > d]$.

- GOSS를 사용한 generalization error는 GOSS 근사가 정확한 경우, 전체 데이터 instance를 사용하여 계산된 값에 근접할 것이다.

- 반면에, 샘플링은 기본 learner의 다양성을 증가시켜 일반화 성능을 향상시키는데 도움을 준다.
→ 더 큰 n을 사용하고, 더 균등하게 instance를 왼쪽, 오른쪽 leaf로 분할할수록, approximation error가 감소

# 3. EFB

**Exclusive Feature Bundling (EFB)**
- 고차원 데이터는 보통 매우 sparse 하다.
- 무손실 접근방식을 설계하여 feature 수를 줄인다.
- Sparse feature 공간에서, 많은 feature 들이 상호 배타적이다.
(i.e., 0이 아닌값을 동시에 취하지 않는다)

- Optimal bundle problem을 graph coloring 문제로 줄여서 품
→ feature가 상호 배타적이지 않은 경우, feature을 vertice로 취하고, edge를 모든 두 feature에 대해 추가함
→ Greedy algorithm ; graph coloring에 대해 합리적으로 좋은 결과를 만들어 bundle을 생성함

**Theorem 4.1** *The problem of partitioning features into a smallest number of exclusive bundles is NP-hard.*

*Proof:* We will reduce the graph coloring problem [25] to our problem. Since graph coloring problem is NP-hard, we can then deduce our conclusion.

# 3. EFB

일반적으로, 100% 상호 배타적이지는 않지만, 매우 적은 수의 feature들이 동시에 0이 아닌 값을 취하는 경우는 드물다.
→ 일부의 conflict를 허용하여, 더 적은 수의 feature bundle을 얻을 수 있으며 계산 효율을 더욱 향상시킬 수 있습니다.
→ **Random polluting**

1. feature들의 상호 배타 여부를 고려하여 그래프를 생성 (서로 상호배타적인 featur들은 edge로 연결안됨)
(각 edge의 weigh는 edge와 연결된 두 vertex 사이의 conflict)

2. 그래프의 degree를 내림차순으로 정렬하고 conflict 계산

3. conflict가 기준 미만이 될때까지 기존 bundle에 할당, conflict가 기준을 넘어가면 새 bundle 생성

**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[i]* $\leq K$ **then**
            bundles[j].add($F$[i]), needNew ← False
            break
    **if** *needNew* **then**
        Add $F[i]$ as a new bundle to *bundles*
**Output**: *bundles*

# 3. EFB

1. 기존의 feature값은 feature bundle에서 식별가능
2. 배타적인 feature들이 다른 bin에 존재하게 함으로써 feature bundle을 구성할 수 있음
→ feature 의 원래 값에 offset을 추가하여 수행

**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** $F[j].bin[i] \neq 0$ **then**
            newBin[i] ← $F$[j].bin[i] + binRanges[j]
**Output**: $newBin, binRanges$

# 4. Experiment

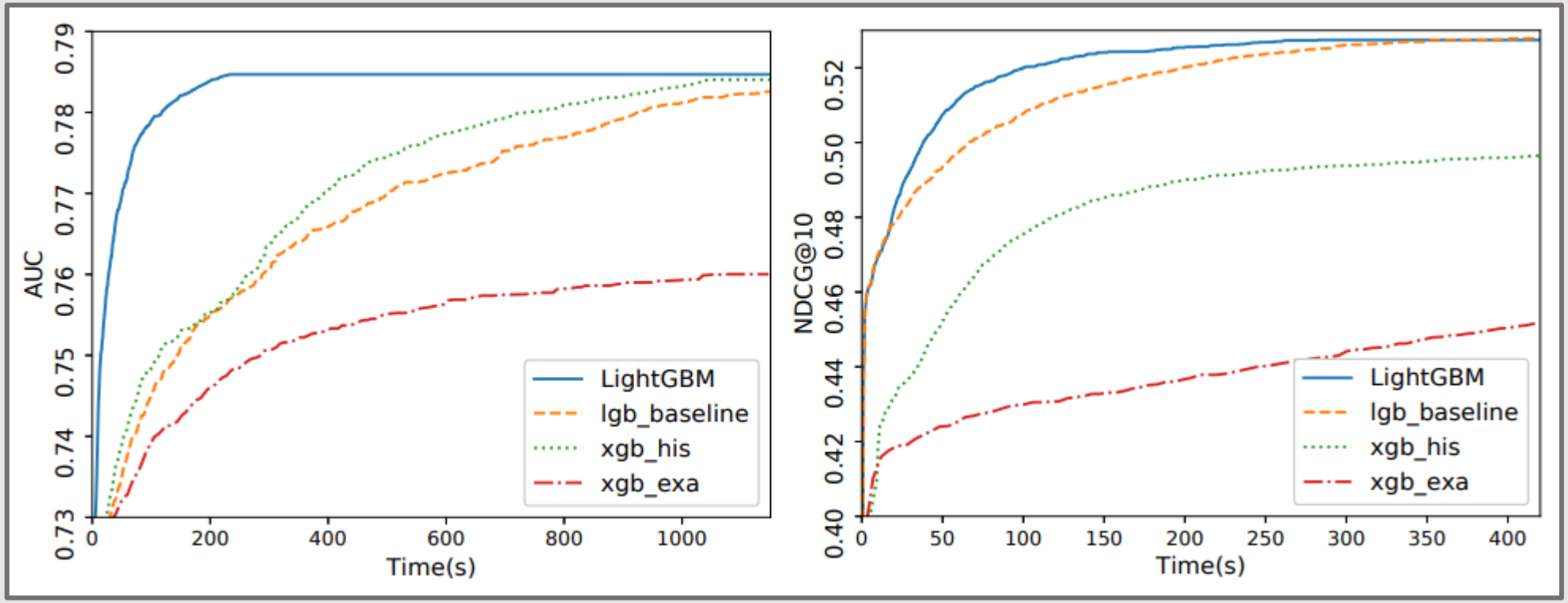
Table 2: Overall training time cost comparison. LightGBM is lgb_baseline with GOSS and EFB. EFB_only is lgb_baseline with EFB. The values in the table are the average time cost (seconds) for training one iteration.

| | xgb_exa | xgb_his | lgb_baseline | EFB_only | **LightGBM** |
|---|---|---|---|---|---|
| Allstate | 10.85 | 2.63 | 6.07 | 0.71 | **0.28** |
| Flight Delay | 5.94 | 1.05 | 1.39 | 0.27 | **0.22** |
| LETOR | 5.55 | 0.63 | 0.49 | 0.46 | **0.31** |
| KDD10 | 108.27 | OOM | 39.85 | 6.33 | **2.85** |
| KDD12 | 191.99 | OOM | 168.26 | 20.23 | **12.67** |

Table 3: Overall accuracy comparison on test datasets. Use AUC for classification task and NDCG@10 for ranking task. SGB is lgb_baseline with Stochastic Gradient Boosting, and its sampling ratio is the same as LightGBM.

| | xgb_exa | xgb_his | lgb_baseline | SGB | **LightGBM** |
|---|---|---|---|---|---|
| Allstate | 0.6070 | 0.6089 | 0.6093 | 0.6064 $\pm$ 7e-4 | **0.6093 $\pm$ 9e-5** |
| Flight Delay | 0.7601 | 0.7840 | 0.7847 | 0.7780 $\pm$ 8e-4 | **0.7846 $\pm$ 4e-5** |
| LETOR | 0.4977 | 0.4982 | 0.5277 | 0.5239 $\pm$ 6e-4 | **0.5275 $\pm$ 5e-4** |
| KDD10 | 0.7796 | OOM | 0.78735 | 0.7759 $\pm$ 3e-4 | **0.78732 $\pm$ 1e-4** |
| KDD12 | 0.7029 | OOM | 0.7049 | 0.6989 $\pm$ 8e-4 | **0.7051 $\pm$ 5e-5** |

# 4. Experiment




LightGBM은 baseline과 거의 동일한 accuracy를 유지하면서 가장 빠름