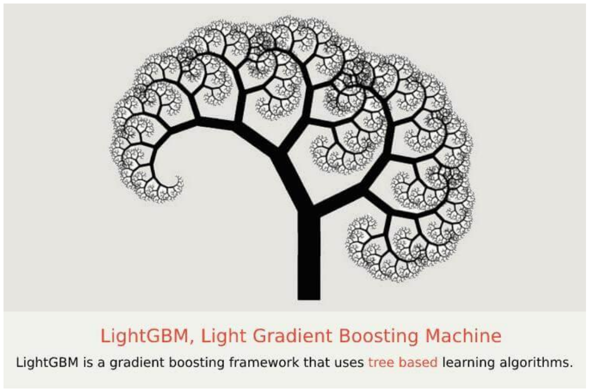
School of Computing

XGBoost: A Scalable Tree Boosting System

Associate Professor
HUANG, Ke-Wei


LightGBM, Light Gradient Boosting Machine
LightGBM is a gradient boosting framework that uses tree based learning algorithms.

eXtreme Gradient Boosting
custom tree building algorithm
XGBoost
by Tianqi Chen
Used for:
• classification
• regression
• ranking
with custom loss functions
Interfaces for Python and R, can be executed on YARN

XGBoost and LightGBM

1. Overview
    1. What is XGBoost?
    2. How good?
    3. Why good?
2. The Algorithm: gradient boosting regression tree with regularization and several settings to improve speed and scalability
3. Parameters Tuning in R
4. LightGBM

References

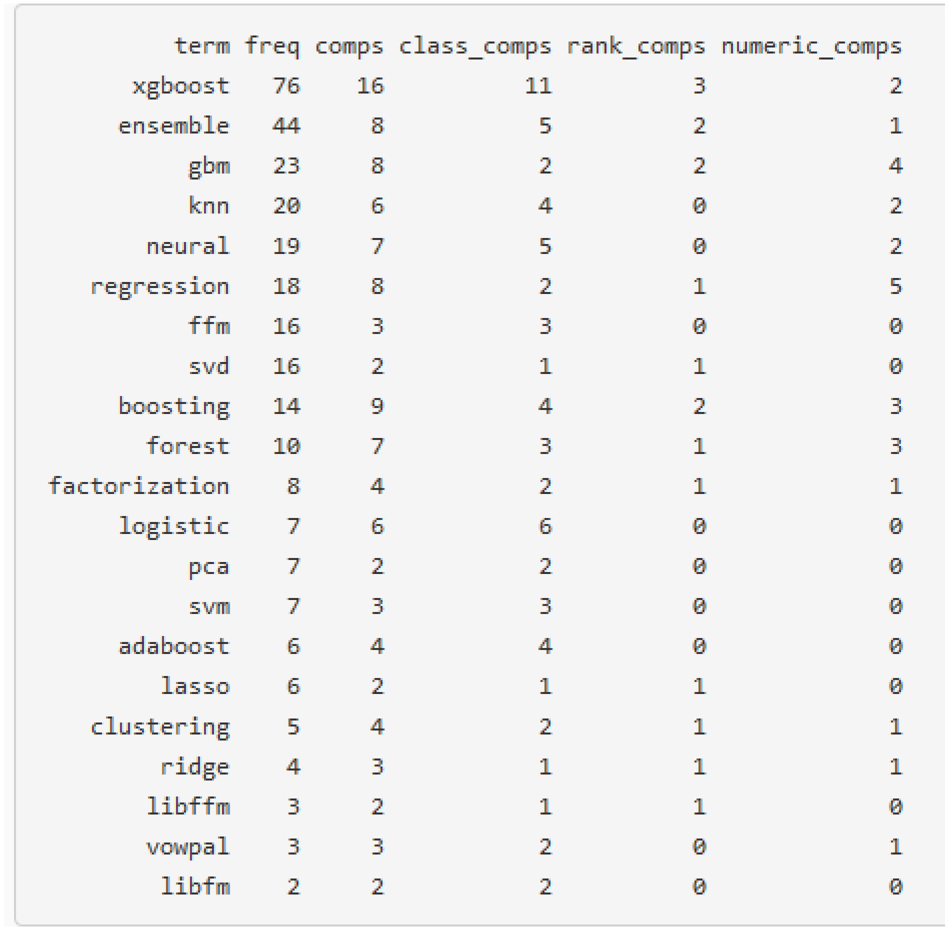
- The Algorithm part is from the original paper by the authors (uploaded to IVLE, optional reading)
  - Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 2016.
- The author's XGBoost site is an important reference:
  - http://xgboost.readthedocs.io/en/latest/model.html
- Parameter Tuning part is from the following link
  - **http://xgboost.readthedocs.io/en/latest/parameter.htm**
  - http://xgboost.readthedocs.io/en/latest/how_to/param_tuning.html
- A playground to help you visualize GBM (not demoed)
  - http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html

What is XGBoost?

- XGBoost (**Ex**treme **G**radient **Boost**ing) is an optimized distributed gradient boosting library.
- The core algorithm is very similar to other GBM packages and it uses gradient boosting (GBM) framework at core.
- Yet, it does better than GBM framework alone because of optimized regularization.
- XGBoost was created by Tianqi Chen, CS PhD Student at University of Washington.
- The author used this package to win a Kaggle competition in May 2014, published a KDD paper in 2016, making XGBoost popular.

How Good is XGBoost?

- End of 2016, the winning algorithms at Kaggle.com by simple term frequency in tm package.



```
          term freq comps class_comps rank_comps numeric_comps
       xgboost   76    16          11          3             2
      ensemble   44     8           5          2             1
           gbm   23     8           2          2             4
           knn   20     6           4          0             2
        neural   19     7           5          0             2
    regression   18     8           2          1             5
           ffm   16     3           3          0             0
           svd   16     2           1          1             0
      boosting   14     9           4          2             3
        forest   10     7           3          1             3
 factorization    8     4           2          1             1
      logistic    7     6           6          0             0
           pca    7     2           2          0             0
           svm    7     3           3          0             0
      adaboost    6     4           4          0             0
         lasso    6     2           1          1             0
    clustering    5     4           2          1             1
         ridge    4     3           1          1             1
        libffm    3     2           1          1             0
        vowpal    3     3           2          0             1
         libfm    2     2           2          0             0
```

How Good is XGBoost?

According to the authors,

- Take Kaggle for example. Among the 29 challenge winning solutions published at Kaggle‵s blog during 2015, 17 solutions used XGBoost. Among these solutions, eight solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles.
- For comparison, the second most popular method, deep neural nets, was used in 11 solutions.
- The success of the system was also witnessed in KDDCup 2015, where XGBoost was used by every winning team in the top-10.

Why is it so good?

- **(FAST) Parallel Computing:** It is enabled with parallel processing (using OpenMP); i.e., when you run xgboost, by default, it would use all cores of your laptop/machine (R used only one of your processor).
- Speed is important because

1. you can tune the algorithm thoroughly with many combinations of parameters
2. You can conduct different forms of validation testing within manageable time (e.g., 10-fold)
3. You can try different feature engineering strategies, different features selection, explore various data mining procedures…etc

Why is it so good?

- **(Accurate) Regularization:** I believe this is the biggest advantage of xgboost package in terms of statistics. GBM has no provision for regularization. Regularization is a technique used to avoid overfitting in tree-based models, among other data mnining methods.

- **Enabled Cross Validation:** In R, we usually use external packages such as caret and mlr to obtain CV results. But, xgboost is enabled with internal CV function.

Why is it so good?

- **Flexibility in Features:** it takes both numerical and categorical features (with onehot needed for some cases) without the need to do scaling.

- **Flexibility in Predicted values:** In addition to regression, classification, and ranking problems (ranking is not covered in BT5152), it supports user-defined objective functions also. Furthermore, it supports user defined evaluation metrics as well.

- **Availability:** Currently, it is available for programming languages such as R, Python, Java, Julia, and Scala.

XGBoost?

1. Overview
   1. What is XGBoost?
   2. How good?
   3. Why good?
2. The Algorithm: gradient boosting regression tree with regularization and several settings to improve speed and scalability
3. Parameters Tuning in R
4. Brief Overview of LightGBM

The Algorithm's objective function

Lambda parameter

Gamma parameter

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{2}$$

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

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

(1) l(,) is a typical loss function between predicted y and actual y.
(2) The second term omega in the first equation is the regularization term (function).
(3) Omega function has two components. The first is the number of leaves of each decision tree. The second is the typical regularization term that penalizes a complicated tree. w_i is the score of the ith leaf of a regression tree.
(4) When those two parameters are zero, XGBoost is the same as traditional gradient boosting tree.
(5) In (1), each function f_k is a regression tree. See more on the next slide.

More on the regression tree equation

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

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

- The predicted value of y is the sum of K regression trees.
- Each regression tree is by CART algorithm. CART is the algorithm behind rpart in R.
- T is the number of leaves of each tree.
- q is the tree structure and q(x) maps x to one leaf.
- w_i is the score (part of the predicted value of y) of that leaf. So $f(\mathbf{x}) = w_{q(\mathbf{x})}$

Training of Gradient Boosting Trees

- Similar to the traditional gradient boosting tree, the solution is derived by greedy approach: we find the tree that most minimize the objective function in each iteration.
- Formally, we will need to add $f_t(x)$ to minimize the following objective function.

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

- Second-order approximation (Taylor expansion) can be used to quickly optimize the objective in the general setting

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

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

Deriving the "Information Gain" Scoring Function (Optional Reading)

Define $I_j = \{i | q(\mathbf{x}_i) = j\}$ as the instance set of leaf j. We can rewrite Eq (3) by expanding Ω as follows

$$\begin{aligned}\tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i\in I_j} g_i) w_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda) w_j^2] + \gamma T\end{aligned} \qquad (4)$$

This sum over all examples.

Sum over each leaf

Sum over each case of each leaf

All examples classified under this leaf are assigned predicted value w_j, which is also = $f_t(x_i)$. $w_j = f_t(x_i)$ is a key point how we move from the first equation to the second equation

Deriving the "Information Gain" Scoring Function (Optional Reading)

- Equation (4) on the previous slide is a quadratic function in terms of w_j , which is also the predicted value that we assign to each leaf.
- The predicted value is w_j decided to minimize the loss function and is given by

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \qquad (5)$$

- Inserting (5) into (4) leads to

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in I_j} g_i\right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \qquad (6)$$

Deriving the "Information Gain" Scoring Function

- Equation (6) is the main scoring function for splitting our regression tree.
- Without two regularization parameters, you can think of (6) as the sum of squared error function with the predicted y as the average of y of examples of that leaf.
- (6) is also similar to the impurity/entropy function that you are familiar with.
- So the tree splitting is decided by the best split that most reduces loss function and regularization terms
- Equation (7) is the important function implemented for deciding how to split a tree.

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{7}$$

This term is because adding one more leaf incurs γ penalty.

Two more Regularization Parameters

1. **Shrinkage** scales newly added weights (w_i) by a factor after each step of tree boosting. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.
2. The second technique is column (feature) subsampling. This is similar to Random Forest's main idea, in each iteration, we use a randomly selected subset to build a regression tree.
3. Row sub-sampling in Random Forest is also supported but features sampling is more effective in preventing overfitting.

More Remarks

1. XGBoost supports “Approximate Algorithm” for finding the best split. XGBoost is design to be scalable (fast) even when we have huge number of features. The approximate algorithm can speed up each regression tree building. (Section 3.2-3.3 in the original paper)

2. XGBoost supports “Sparsity-aware Split Finding”. When your feature is sparse (e.g., tf-idf, many dummy variables, many missing values imputed by 0, …etc.), you can use this feature to speed up the computation. (Section 3.4 in the original paper)

Summary of the idea of the algorithm

1. We iteratively build CART regression tree $f_i(x)$ to predict y.
2. The number of iteration is a parameter.
3. Each regression tree is decided by the scoring function (7) to decide the best split.
4. The depth of the tree is constrained by a depth parameter, which is also one important stopping condition of each tree (pre-pruning).
5. After we have a predicted value $f_i(x)$, the residual of y (unpredicted part of y) is used as the dependent variable for prediction in the next iteration (this is weighted down by a learning rate).

XGBoost?

1. Overview
    1. What is XGBoost?
    2. How good?
    3. Why good?
2. The Algorithm: gradient boosting regression tree with regularization and several settings to improve speed and scalability
3. Parameters Tuning in R ⇦
4. Brief Overview of LightGBM

XGBoost Tuning Parameters

- XGBoost parameters can be divided into three categories according to the authors: https://xgboost.readthedocs.io/en/latest/parameter.html
- **General Parameters (skipped):**
  - Controls the booster type in the model
  - Decide computational resources
- **Booster Parameters:** several main parameters for tuning
- **Learning Task Parameters:** objective function and performance metrics

XGBoost Booster Parameters

Caret Supports the following 7 parameters

1. Nrounds: # Boosting Iterations
2. max_depth: Max Tree Depth
3. Eta: Shrinkage
4. Gamma: Minimum Loss Reduction
5. colsample_bytree: Subsample Ratio of Columns
6. min_child_weight: Minimum Sum of Instance Weight
7. Subsample: Subsample Percentage of rows

XGBoost Parameters Tuning

1. Nrounds: # Boosting Iterations

- **[default=100]**
- It controls the maximum number of iterations. For classification, it is similar to the number of trees to grow.

2. max_depth: Max Tree Depth

- **[default=6]**
- It controls the depth of the tree. Larger the depth, more complex the model; higher chances of overfitting.
- There is no standard value for max_depth.
- Data sets with more (useful) features require deep trees to learn the rules from data.

XGBoost Parameters Tuning

3. eta: Shrinkage

- **[default=0.3]**
- It controls the learning rate across rounds, i.e., the rate at which our model learns patterns in data. After every round, it shrinks the feature weights to reach the best optimum.
- **Lower eta leads to slower computation. It must be supported by increase in nrounds.**
- Typically, it lies between 0.01 - 0.3

XGBoost Parameters Tuning

4. gamma: Minimum Loss Reduction

- **[default=0]**
- It controls regularization (or prevents overfitting). The optimal value of gamma depends on the data set and other parameter values.
- Higher the value, higher the regularization. Regularization means penalizing large coefficients which don't improve the model's performance.
- default = 0 means no regularization.
- *Tune trick:* Start with 0 and check CV error rate. If you see train error << test error, bring gamma into action. Higher the gamma, lower the difference in train and test CV. Start with gamma=5 and see the performance.

XGBoost Parameters Tuning

5. colsample_bytree: Subsample Ratio of Columns

- **[default=1]**
- It control the number of features (variables) supplied to a tree

7. subsample: Subsample Percentage

- **[default=1]**
- It controls the number of samples (observations) supplied to a tree.

XGBoost Parameters Tuning

6. min_child_weight: Minimum Sum of Instance Weight

- **[default=1]**
- In regression, it refers to the minimum number of instances required in a child node.
- In classification, if the leaf node has a minimum sum of instance weight lower than min_child_weight, the tree splitting stops.
- In simple words, it is another parameter that prevents overfitting.

XGBoost Tuning Parameters

Two advanced, optional parameters not supported by Caret.

- **lambda[default=0]** It controls L2 regularization (equivalent to Ridge regression) on weights. It is used to avoid overfitting.
- **alpha[default=1]** It controls L1 regularization (equivalent to Lasso regression) on weights. In addition to shrinkage, enabling alpha also results in feature selection. Hence, it's more useful on high dimensional data sets.

Bias-Variance Tradeoff

- When we allow the model to become more complicated (e.g. more depth), the model has better ability to fit the training data, resulting in a less biased model. However, such complicated model requires more data to fit.

- Most of parameters in xgboost are about bias variance tradeoff. Parameters Documentation will tell you whether each parameter will make the model more conservative or not. This can be used to help you turn the knob between complicated model and simple model.

Control Overfitting

- When you observe high training accuracy, but much lower tests accuracy, it is likely that you encounter overfitting problem.
- Solution 1: directly control model complexity, this include max_depth, min_child_weight and gamma
- Solution 2: add randomness to make training robust to noise
  - This include subsample, colsample_bytree
  - You can also reduce stepsize eta, but needs to remember to increase num_round when you do so.

XGBoost Outputs

- **Objective[default=reg:linear]**
  - reg:linear - for linear regression
  - binary:logistic - logistic regression for binary classification. It returns class probabilities
  - multi:softmax - multiclassification using softmax objective. It returns predicted class labels. It requires setting num_class parameter denoting number of unique prediction classes.
  - multi:softprob - multiclassification using softmax objective. It returns predicted class probabilities.

XGBoost Metrics

- **eval_metric [no default]**
  - These metrics are used to evaluate a model's accuracy on validation data. For regression, default metric is RMSE. For classification, default metric is error.
  - Available error functions are as follows:
    - mae - Mean Absolute Error (used in regression)
    - Logloss - Negative loglikelihood (used in classification)
    - AUC - Area under curve (used in classification)
    - RMSE - Root mean square error (used in regression)
    - error - Binary classification error rate [#wrong cases/#all cases]
    - mlogloss - multiclass logloss (used in classification)

XGBoost?

1. Overview
    1. What is XGBoost?
    2. How good?
    3. Why good?
2. The Algorithm: gradient boosting regression tree with regularization and several settings to improve speed and scalability
3. Parameters Tuning in R

4. Brief Overview of LightGBM

Evolution of Boosting

1. AdaBoost: now outdated a bit because performance is not as good as GBM.
2. GBM (in R and in Python): the first GBM, although called boosting, the main idea is quite different from AdaBoost.
   – Problem => GBM package is accurate but slow.
3. XGBoost: the core algorithm is the same as GBM.
   – This is a better implementation in terms of speed and scalability.
   – It also coded more flexibility for users to do tuning by various regularization functions.
4. LightGBM (only in Python, difficult to use in R):
   – The main idea is to sacrifice accuracy (or keep the same accuracy) while further speed up XGBoost in the Big data era.
   – There are two parts of algorithms that are different from XGBoost.
   – Quite a number of users claim the accuracy performance is better than XGBoost?

Light GBM

- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems* (pp. 3146-3154).
- Light GBM and XGBoost seem to be the two most popular and powerful packages now.
- NIPS is the #2 conference in Machine Learning.
- The authors are from Microsoft. So sometimes people say Microsoft's LightGBM

Overview (from the authors)

- The efficiency and scalability of XGBoost are still unsatisfactory when the feature dimension is high and data size is large.
- This paper propose two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB).
- According to this paper’s 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.**
- This depends on your dataset… If your dataset is small in the number of rows and columns, then speed is not much faster. The power of XGBoost and LightGBM wont be obvious, relative to old algorithms such as random forest, NN, SVM…etc.

New Improvement 1. GOSS (Optional)

- In GBDT, if an instance is associated with a small gradient, the training error for this instance is small and it is already well-trained.
- A straightforward idea is to discard those data instances with small gradients.
- However, the data distribution will be changed by doing so, which might hurt the accuracy of the learned model.
- To avoid this problem, this paper proposes a new method called Gradient-based One-Side Sampling (GOSS).

GOSS

- GOSS keeps all the instances with large gradients and performs random sampling on the instances with small gradients.
- In order to compensate the influence to the data distribution, when computing the information gain, GOSS introduces a constant multiplier for the data instances with small gradients (see Alg. 2).
- Specifically, GOSS firstly sorts the data instances according to the absolute value of their gradients and selects the top a * 100% instances.
- Then it randomly samples b* 100% instances from the rest of the data.
- After that, GOSS amplifies the sampled data with small gradients by a constant (1-a)/b when calculating the information gain.

Histogram-Based vs GOSS

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.
    ...
```

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 $\leftarrow \{\}$, fact $\leftarrow \frac{1-a}{b}$
topN $\leftarrow$ a $\times$ len($I$) , randN $\leftarrow$ b $\times$ len($I$)
for $i = 1$ to $d$ do
    preds $\leftarrow$ models.predict($I$)
    g $\leftarrow loss(I$, preds), w $\leftarrow$ {1,1,...}
    sorted $\leftarrow$ GetSortedIndices(abs(g))
    topSet $\leftarrow$ sorted[1:topN]
    randSet $\leftarrow$ RandomPick(sorted[topN:len(I)],
    randN)
    usedSet $\leftarrow$ topSet + randSet
    w[randSet] $\times=$ fact ▷ Assign weight $fact$ to the
    small gradient data.
    newModel $\leftarrow$ L($I$[usedSet], $-$ g[usedSet],
    w[usedSet])
    models.append(newModel)
```

Improvement 2: Exclusive Feature Bundling (Optional)

- Usually in real applications, although there are a large number of features, the feature space is quite sparse, which provides us a possibility of designing a nearly lossless approach to reduce the number of effective features.
- Specifically, in a sparse feature space, many features are (almost) exclusive, i.e., they rarely take nonzero values simultaneously (e.g., OneHot output).
- This paper designs an efficient algorithm by reducing the optimal bundling problem to a graph coloring problem (by taking features as vertices and adding edges for every two features if they are not mutually exclusive), and solving it by a greedy algorithm with a constant approximation ratio.

EFB

- 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.
- In this way, the complexity of histogram building changes from O(#data * #feature) to O(#data * #bundle), while #bundle << #feature. Then we can significantly speed up the training of GBDT without hurting the accuracy.
- There are two issues to be addressed.
  - The first one is to determine which features should be bundled together (difficult and skipped in this lecture).
  - The second is how to construct the bundle (easy).

EFB Issue 2 (Algorithm 4)

- 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.

EFB Algorithms

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: $bundles$
```

Algorithm 4: Merge Exclusive Features

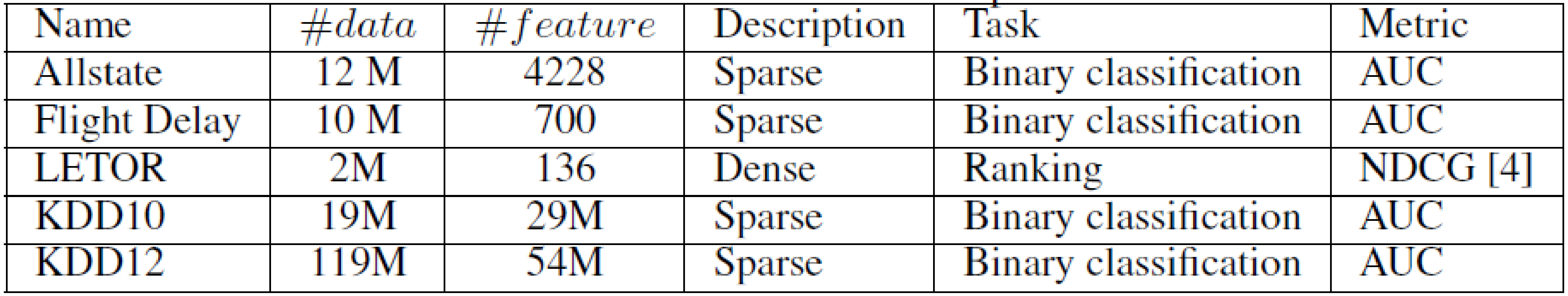
```
Input: $numData$: number of data
Input: $F$: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for $f$ in $F$ do
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new  Bin(numData)
for $i = 1$ to $numData$ do
    newBin[i] ← 0
    for $j = 1$ to len($F$) do
        if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

5. Experimentation

- Five different datasets which are all publicly available.
- We directly use the features used by the winning solution from NTU (Taiwan one)
- These datasets are large, including both sparse and dense features, and cover many real-world tasks.
- Our experimental environment is a Linux server with two E5-2670 v3 CPUs (in total 24 cores) and 256GB memories. All experiments run with multi-threading and the number of threads is fixed to 16.
- One caveat of this experimentation is we don’t know when the number of features and/or number of rows are small (or not large enough), what will happen?

Experimentation

Table 1: Datasets used in the experiments.



Name	$\#data$	$\#feature$	Description	Task	Metric
Allstate	12 M	4228	Sparse	Binary classification	AUC
Flight Delay	10 M	700	Sparse	Binary classification	AUC
LETOR	2M	136	Dense	Ranking	NDCG [4]
KDD10	19M	29M	Sparse	Binary classification	AUC
KDD12	119M	54M	Sparse	Binary classification	AUC

The authors check 5 cases of algorithms.

1. XGBoost with pre-ordering
2. XGBoost with histogram-based
3. LightGBM without both methods
4. LightGBM with only EFB
5. LightGBM with both

3-4 can help us see the effect of EFB,

4-5 can help us see the effect of GOSS

Results

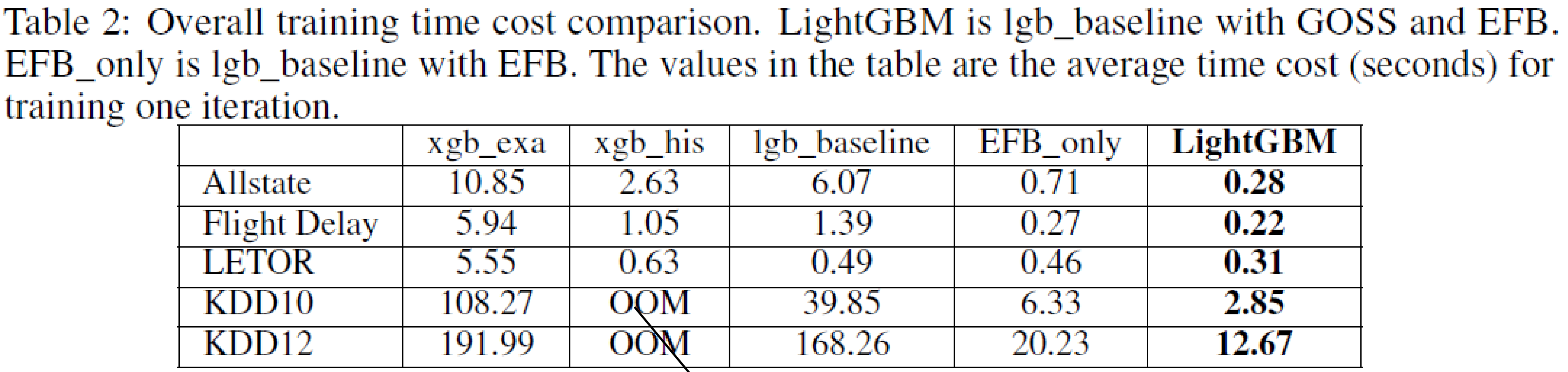


Table 2: Overall training time cost comparison. LightGBM is lgb_baseline with GOSS and EFB. EFB_only is lgb_baseline with EFB. The values in the table are the average time cost (seconds) for training one iteration.

	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**

Out of Memory: you cannot even run XGBoost on this dataset

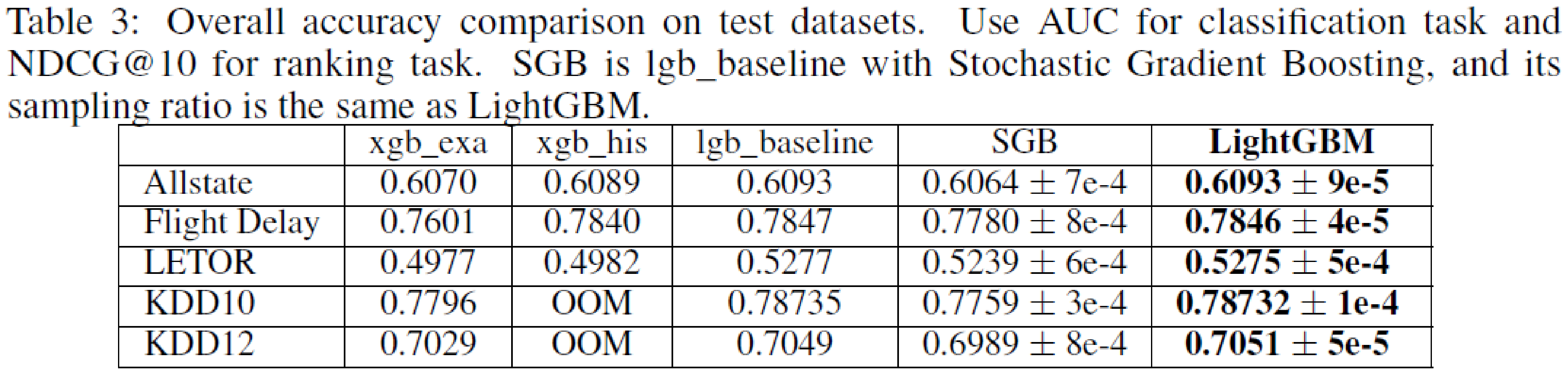


Table 3: Overall accuracy comparison on test datasets. Use AUC for classification task and NDCG@10 for ranking task. SGB is lgb_baseline with Stochastic Gradient Boosting, and its sampling ratio is the same as LightGBM.

	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**

Analysis on GOSS

- First, we study the speed-up ability of GOSS. From the comparison of LightGBM and EFB_only in Table 2, we can see that GOSS can bring nearly 2x speed-up by its own with using 10% - 20% data.
- GOSS can learn trees by only using the sampled data. However, it retains some computations on the full dataset, such as conducting the predictions and computing the gradients. Thus, we can find that the overall speed-up is not linearly correlated with the percentage of sampled data. However, the speed-up brought by GOSS is still very significant and the technique is universally applicable to different datasets.

Analysis on EFB

- We check the contribution of EFB to the speed-up by comparing lgb_baseline with EFB_only.

- Please note lgb_baseline has been optimized for the sparse features, and EFB can still speed up the training by a large factor. It is because EFB merges many sparse features (both the one-hot coding features and implicitly exclusive features) into much fewer features.

- With above analysis, EFB is a very effective algorithm to leverage sparse property in the histogram-based algorithm, and it can bring a significant speed-up for GBDT training process.