高等机器学习

梯度提升树

柯国霖
微软亚洲研究院

Microsoft

清華大學
Tsinghua University

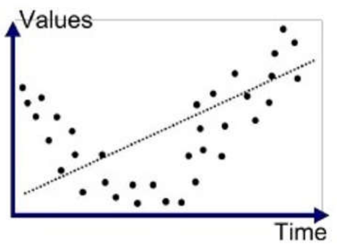
Outline

- Overview
- Decision Tree
- Boosting
- GBDT (Gradient Boosting Decision Tree)

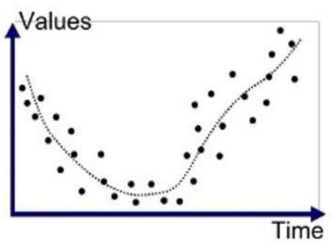
Overview

No Free Lunch

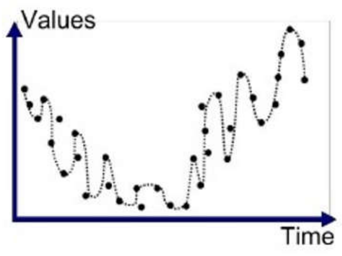
- It is hard to pre-define a universal model/function for all kinds of tasks/data
  - The distributions of data are various and unknown
  - Simple function -> underfitting
  - Complex function -> overfitting
- Therefore, human-efforts is required in model design, for bias-variance tradeoff.



Underfitted



Good Fit/Robust

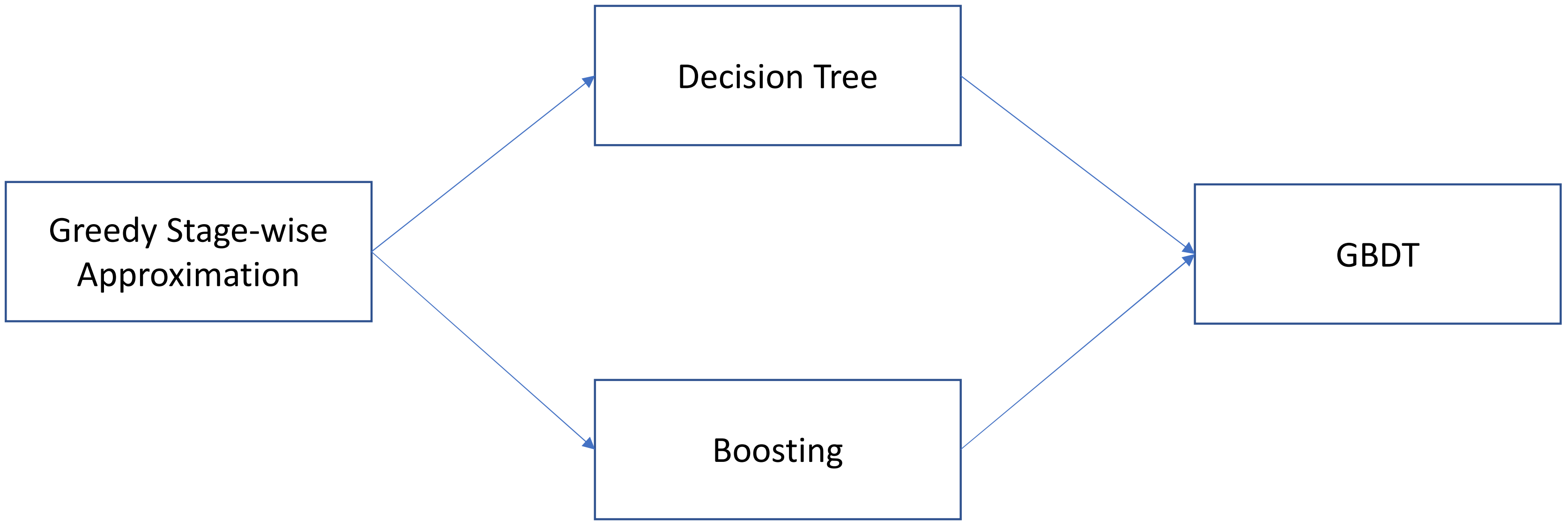


Overfitted

"Cheap Launch": Greedy Stage-wise Approximation

- Dynamic model space: approximate the data and increase model complexity step by step, greedily
    - Can stop approximation when "good fit", by early-stopping on validation
- Also called Greedy Additive Approximation
    - $F_m(X) = F_{m-1}(X) + f_m(X)$, where $L(F_m(X), Y) < L(F_{m-1}(X), Y)$
- Both Boosting and Decision Tree are in this category

Greedy Stage-wise Approximation



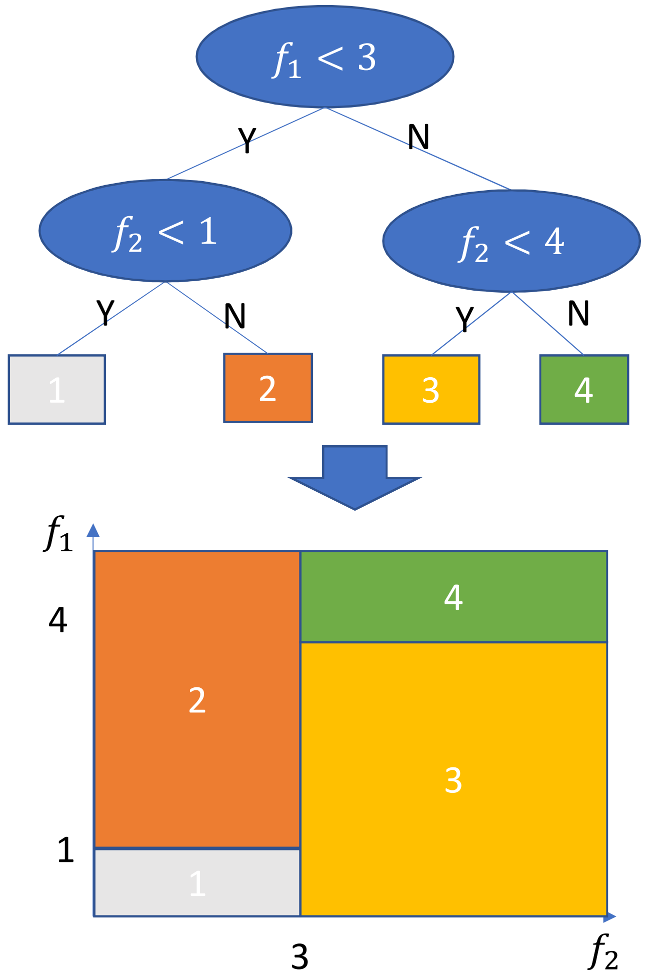
Decision Tree

Recall: (Discriminative) Supervised Learning

- Components of supervised learning
  - Data: $[X, Y]$
    - $X = [x_1, x_2, \dots, x_n]^T, Y = [y_1, y_2, \dots, y_n]^T, x_i = [x_{i1}, x_{i2}, \dots, x_{im}]$
      - x_i is i-th training record, its label is y_i
      - x_{ij} is the j-th feature value of i-th training record.
  - Model/Function with learnable parameters $\boldsymbol{\theta}$: $F(x;\ \boldsymbol{\theta})$
    - E.g. Linear model $F(x_i\ ;\ \boldsymbol{w}) = \sum_j w_j x_{ij}$
  - Objective Loss Function: $\sum_i l(F(x_i; \boldsymbol{\theta}), y_i)$
    - E.g. L2 loss: $l(F(x_i; \boldsymbol{\theta}), y_i) = (F(x_i) - y_i)^2$
- Goal of supervised learning: learn the parameters $\boldsymbol{\theta}^*$ with (almost) the lowest losses, over data $[X, Y]$
  - $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}}(\sum_i l(F(x_i; \boldsymbol{\theta}), y_i))$

Decision Tree: Structure View

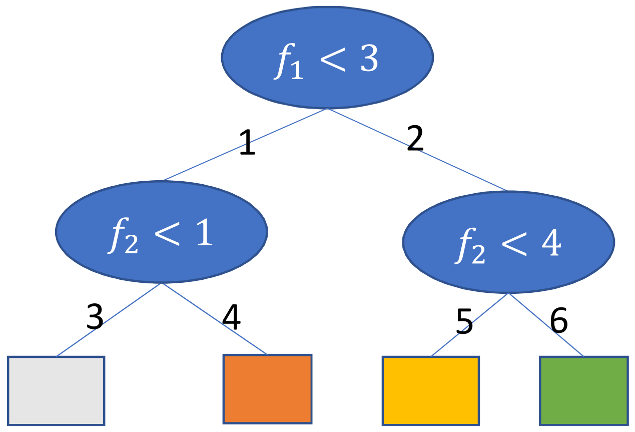
- Decision tree partition data into many non-overlapping regions
- Components
  - Non-leaf node, decision node
    - Contain a decision rule(split), $\{feature, threshold\}$
    - Partition current region into two regions
  - Leaf node,
    - Each x_i belongs one leaf
    - Each leaf r_j has an output value



Decision Tree Definition

- Define a tree with m leaves as $T_m = (S_{m-1},\ R_m)$, where
  - S_{m-1} contains $m-1$ internal nodes
    - The decision rule of j-th node $S_j = (f^j,\ t^j)$
  - R_m contains m leaf nodes, each of them contains an output value
    - $R_j = average(\{y_i \mid x_i \in j\ th\ leaf\})$, for regression tree
- $T_m(x_i) = R_m\big(I(S_{m-1}, x_i)\big)$, which returns x_i's prediction, where
  - I is a decision/test function, and returns the x_i's leaf index based on S_{m-1}
    - The test in j-th node
      - Go to left node if $x_{i,f^j} \le t^j$, otherwise right node (numerical features)
      - Go to left node if $x_{i,f^j} \in t^j$, otherwise right node (categorical features)
    - From root to leaf, ~ $\log_2 m$ tests
  - $R_m(j)$ returns the leaf output of j-th leaf

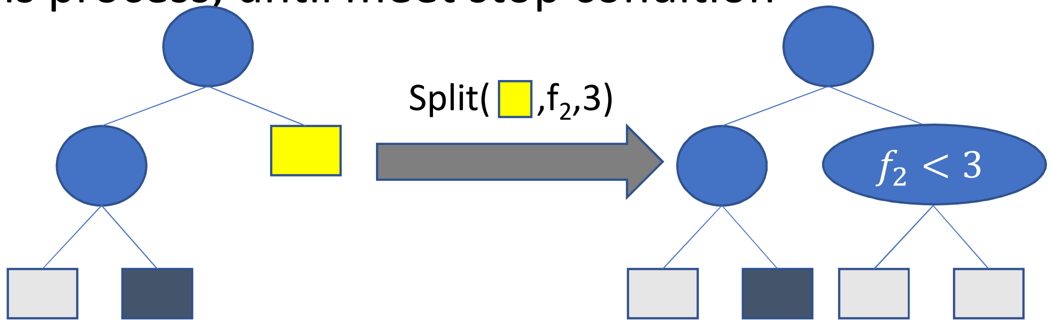
Decision Tree: Inference Example



- x = [1, 0]. decision path 1->3
- x = [1, 2]. decision path 1->4
- x = [4, 3]. decision path 2->5
- x = [4, 5]. decision path 2->6

Decision Tree Learning

- Greedy process
- Choose a leaf and split it into two leaf, with maximum loss reduction
  - $(p_m, f_m, t_j) = \arg\max_{(p,f,v)} (L(T_{m-1}, Y) - L(T_{m-1}.split(p,f,t), Y))$
    - p is the spilt node, f is the feature and t is the threshold
  - $T_m = T_{m-1}.split\,(p_m, f_m, t_m)$
    - Greedy additive process: remove one leaf, and add two new leaves
  - Repeat this process, until meet stop condition



Decision Tree Learning Algorithm

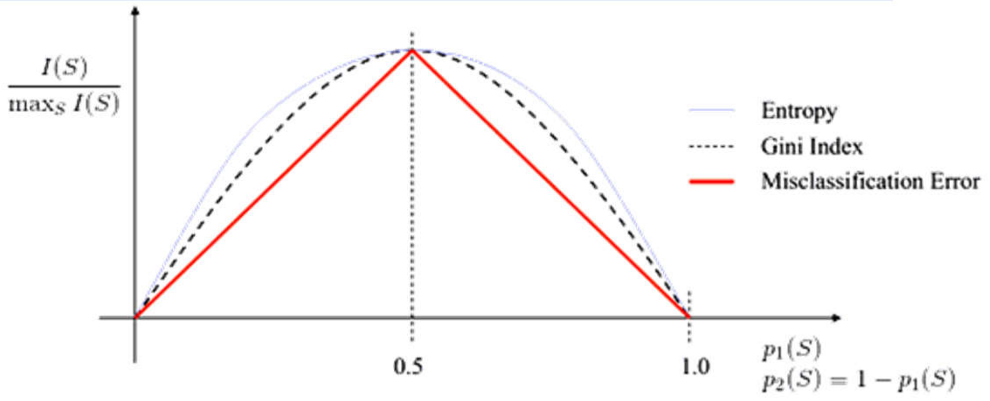
```
Algorithm: **DecisionTree**
**Input: Training data $(X, Y)$ , number of leaf $C$,**
          **Loss function $l$**
  ▻ put all data on root
  $T_1(X) = X$
  **For** m in (2,C):
     ▻ find best split
     $\boxed{(p_m, f_m, v_m) = FindBestSplit(X, Y, T_{m-1}, l)}$
     ▻ perform split
     $T_m(X) = T_{m-1}(X).split(p_m, f_m, v_m)$
```

```
Algorithm: **FindBestSplit**
  **Input: Training data $(X, Y)$ , Loss function $l$, Current Model $T_{m-1}(X)$**
  **For all** Leaf p in $T_{m-1}(X)$:
    $X' = \text{data\_in\_cur\_leaf}(X, p)$
    **For all** $f$ in $X'.features$:
      **For all** $v$ in $f.thresholds$:
        $(left, right) = partition(p, f, v)$
        $\Delta loss = L(X_p, Y_p) - L(X_{left}, Y_{left}) - L(X_{right}, Y_{right})$
        if $\Delta loss > \Delta loss(p_m, f_m, v_m)$:
          $(p_m, f_m, v_m) = (p, f, v)$
```

Decision Tree Learning: Well-known Loss

ID3	Information Gain	Classification	$L(X,Y) = -\sum_{j=1}^{J} p_j \log p_j, where\ p_j = \frac{cnt(y_i = j)}{cnt(x)}$ Total J classes
CART	Gini impurity	Classification	$L(X,Y) = 1 - \sum_{j=1}^{J} p_j^2, where\ p_j = \frac{cnt(\{y_i = j\})}{cnt(Y)}$ Total J classes
CART	Variance reduction	Regression	$L(X,Y) = \sum_{i=1}^{n} (y_i - \bar{y})^2, where\ \bar{y} = \frac{sum(Y)}{cnt(Y)}$

CART = Classification And Regression Tree



Missing Value Handle in Tree

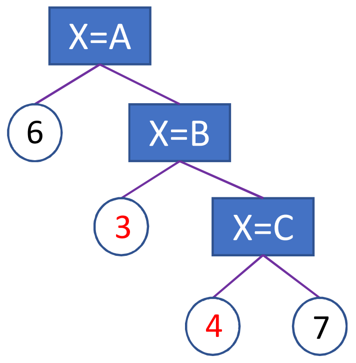
- In most models, the missing values need to be filled before training
- However, in tree, the missing values could be directly handled
- Simply test which child (left or right) is the best for the missing values
    - Two passes of sorted data / histogram
    - All missing values are in the same child

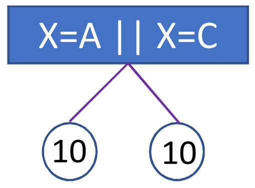
Categorical Feature in Tree

- Learning tree from numerical values is easier, due to they could be ordered
  - The decision/split is, left child if value <= threshold, else right child
  - There are #distinct_value possible split results
- However, there are not order relations in categorical (nominal) values
  - The decision/split is, left child if value in (c1, c2, …,ck), else right child
  - There are 2^#distinct_value possible split results

Categorical Feature in Tree

- The common solution one-hot encoding is not good for tree(equal to use one category in one node, one-vs-rest split)
    - It needs deep tree when uses categorical feature
    - It will partition data in many small regions, which is not good for tree learning
    - It cannot be used for the high cardinality categorial features due to the highly imbalance in one-vs-rest split





Note: Numbers in circles represent to the #data in that node

Categorical Feature in Tree: Encoding

- Convert to numerical values
- Unsupervised encoding
  - Ordinal encoding
  - Binary encoding / Base k encoding
    - Produce log(#) features
  - Frequency encoding
- Supervised encoding
  - Target encoding (k-folds)

Efficient Tree Learning

- The most time-consuming part in decision tree is the split finding
- The time complexity is $O(feature \times \#threshold \times \#data)$ for a leaf
  - Time cost for partition and $\Delta loss$ are both O(#data)
  - The partition could be O(1), if the v is sorted
  - The $\Delta loss$ could be O(1) as well, if the loss could be accumulated

```
Algorithm: FindBestSplit
  Input: Training data $(X, Y)$ , Loss function $l$, Current Model $T_{m-1}(X)$
  For all Leaf p in $T_{m-1}(X)$:
    $X' = \text{data\_in\_cur\_leaf}(X, p)$
    For all $f$ in $X'.features$:
      For all $v$ in $f.thresholds$:
        $(left, right) = partition(p, f, v)$
        $\Delta loss = L(X_p, Y_p) - L(X_{left}, Y_{left}) - L(X_{right}, Y_{right})$
        if $\Delta loss > \Delta loss(p_m, f_m, v_m)$:
          $(p_m, f_m, v_m) = (p, f, v)$
```

Efficient Tree Learning: $\Delta loss$ Simplification

- Denote L2 loss for a leaf r_j as L_j , as we only use regression tree in GBDT
  - $\mathrm{L}_j = \sum_{x_i \in r_j} \left(y_i - R(r_j)\right)^2, R(r_j) = (\sum_{x_i \in r_j} y_i)/n_j$
  - Denote $S_j = \sum_{x_i \in r_j} y_i, SQ_j = \sum_{x_i \in r_j} \left((y_i)^2\right)$, Then
  - $\mathrm{L}_j = n_j \, E_{x_i \in r_j}\left[\left(y_i - E_{x_i \in r_j}(y_i)\right)^2\right] = n_j \left(\frac{SQ_j}{n_j} - \left(\frac{S_j}{n_j}\right)^2\right) = -\frac{S_j^2}{n_j} + SQ_j$

$\boxed{E\left[\left(x - E(x)\right)^2\right] = E[x^2] - E[x]^2}$

- And we choose a split with maximal delta loss:
  - $\Delta loss = L_P - L_{left} - L_{right} = \frac{S_{Left}^2}{n_{left}} + \frac{S_{right}^2}{n_{right}} - \frac{S_P^2}{n_P}$
- After simplification, $\Delta loss$ could be accumulated

Efficient Tree Learning: Sorted Split Finding

```
Algorithm: **FindBestSplit**
  **Input: Training data** $(\boldsymbol{X}, \boldsymbol{Y})$ **, Current Model** $\boldsymbol{T_{c-1}}(\boldsymbol{X})$
  **For all** Leaf p in $T_{c-1}(X)$:
    $X' = \text{data\_in\_cur\_leaf}(X, p)$
    **For all** f in $X'.features$:
      $sorted\_idx$ = get_sorted_indices($f.values$)
      $S_L = n_L = 0$
      **For** $i$ **in** (0,len($f.values$) - 1):
        $j = sorted\_idx[i]$
        $S_L \mathrel{+}= Y[j]$ ; $n_L \mathrel{+}= 1$
        $S_R = S_P - \mathrm{S_L};\ \mathrm{n_R} = \mathrm{n_P} - \mathrm{n_L}$
        $\Delta loss = \frac{S_L^2}{n_L} + \frac{S_R^2}{n_R} - \frac{S_P^2}{n_P}$
        if $\Delta loss > \Delta loss(p_m, f_m, v_m)$:
          $(p_m, f_m, v_m) = (p, f, f.values[j])$
```

Could be cached, to avoid re-sort

From $O(feature \times \#threshold \times \#data)$ to $O(feature \times \#data)$

Efficient Tree Learning: Pre-sorted Solution

Pre-sorted solution often be used with level-wise tree growing, Growing one level nodes by one-pass of data
Preparation: generate sorted_idx for each feature (pre-sorted)

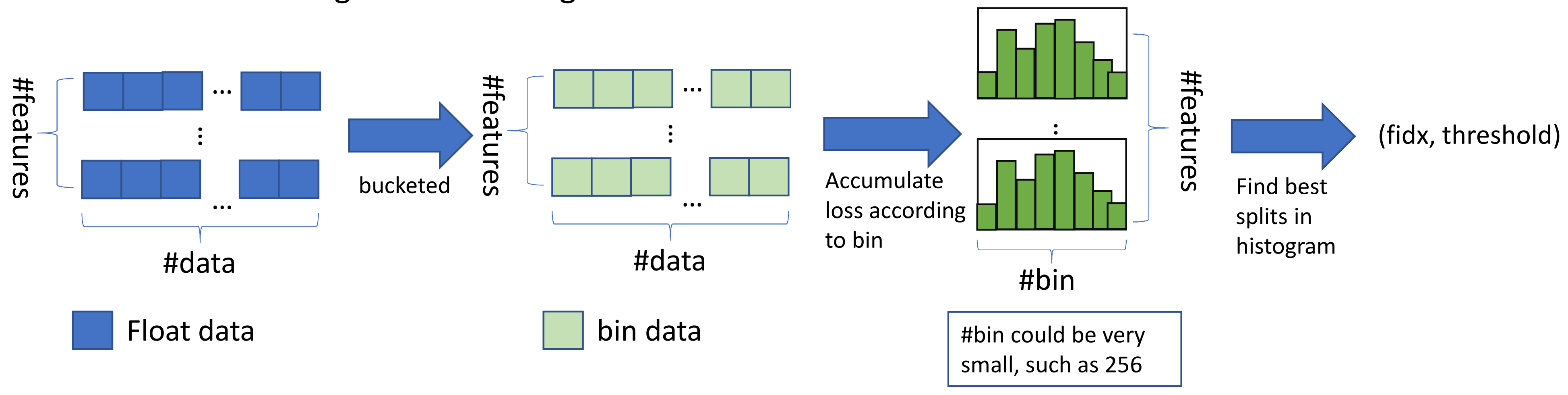
```
Algorithm: **PreSortedTreeLearning**
Input: Training data $(X, Y)$ , max depth D
  ▻ put all data on root
  row_to_node = {1, 1, 1, 1, ….}
  split_info_per_node = {}  # best splits for each node
  for m in (2, D):
    ▻ find best split per node
    for fidx in (1, X.num_feature):
      for row_idx in (1, X.num_data):
        sorted_idx = X[fidx].sorted_idx[row_idx]
        y = Y[sorted_idx] # random access
        node = row_to_node[sorted_idx] # random access
        cur_split = split_info_per_node[node]
        $\Delta loss$ = cur_split.accumulate(y).loss
        if $\Delta loss$ > cur_split.best_loss:
            split_info_per_node[node].best_split = (fidx, X[fidx].val[sorted_idx])
    ▻ perform split (update row_to_node)
    for row_idx in (1, X.num_data):
      node = row_to_node[row_idx]
      fidx, threshold = split_info_per_node[node].best_split
      if X[fidx].val[row_idx] <= threshold: # left child
        row_to_node[row_idx] *= 2
      else: # right child
        row_to_node[row_idx] = row_to_node[row_idx] * 2 + 1
```

Memory access cost for one feature:
#row * 4 Bytes continued access
2 * #row * 4 Bytes random access

Very inefficient due to many random accesses, which will cause serious cache-miss problem

Efficient Tree Learning: Histogram Optimization

- What if the sort could be avoided?
  - bucket the feature values, and accumulate the loss in the same bin
- Bucket continues values to discrete values(“bin”)
  - E.g. [0,0.1) -> 0, [0.1,0.3)->1, …
- Improve generalization ability
  - Avoid overfitting from too fine-grained threshold


#features
#data
Float data
bucketed
#features
#data
bin data
Accumulate loss according to bin
#bin
#bin could be very small, such as 256
#features
Find best splits in histogram
(fidx, threshold)

Efficient Tree Learning: Histogram Optimization

Preparation: generate bucked bin for each feature

```
Algorithm: **FindBestSplitWithHistogram**
  Input: Training data $(X, Y)$ , Current Model $T_{c-1}(X)$
  For all Leaf p in $T_{c-1}(X)$:
    used_indices = data_in_current_leaf(p)
    ordered_y  = []
    for row_idx in used_row:
       ordered_y.append(Y[row_idx])
    for fidx in (1, X.num_feature):
        H = new Histogram()
        for i in (1, len(used_row)):
          row_idx = used_row[i]
          bin = X[fidx].bin[row_idx]
          H[bin].y += ordered_y[i]
          H[bin].n += 1
        best_split_per_feature[fidx] = H.FindBest()
```

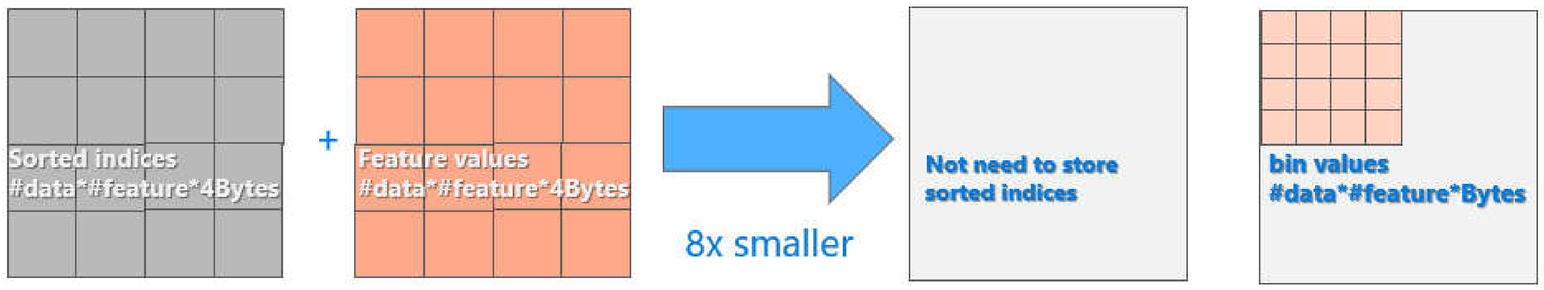
Cache one time, and all features can access it continued

Memory access cost for one feature:
#row * Bytes ordered (but non-continued) access
2 * #row * 4 Bytes continued access

Improve cache hit chance, leverage the cache performance

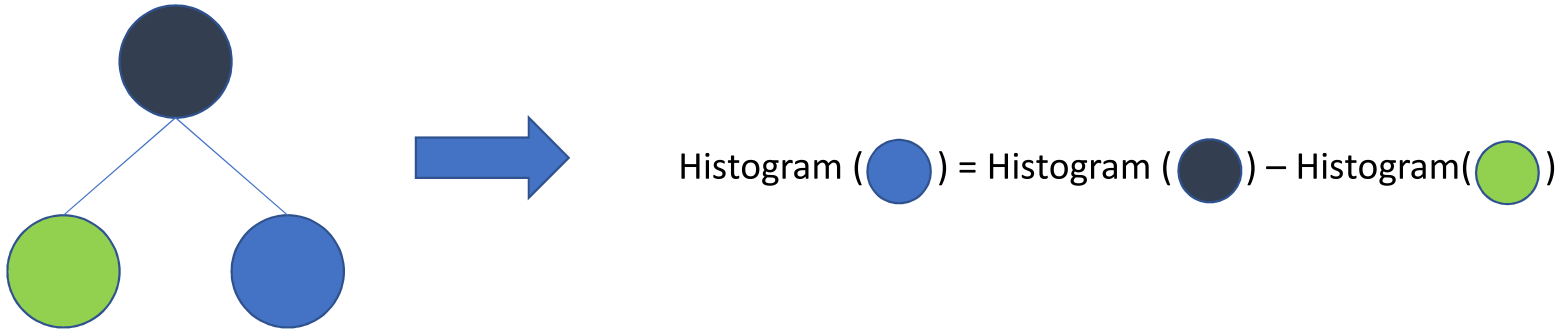
Efficient Tree Learning: Histogram Optimization

- Histogram optimization also reduce the memory cost
- Only need to save bin values.
- If #bins is small, can use small data type, e.g. uint8_t, to store training data



Efficient Tree Learning: Histogram Subtraction

- To get one leaf's histograms in a binary tree, we can use the histogram subtraction of its parent and its neighbor
  - Reduce the cost from #row to #bin
- More than 2x speed-up



Ensemble of Decision Trees

- Cannot always increase the complexity of a single tree
  - Too few data in the deep nodes, and cause the biased splits
- Too deep tree -> overfitting; too shallow tree -> underfitting
- Therefore, a single tree often cannot perform well. And ensemble of shallow trees is widely-used, such as Random Forest and GBDT

Boosting

Boosting

- Boosting is an additive model
  - $F_m(X) = F_{m-1}(X) + f_m(X) = f_1(X) + \ldots + f_m(X)$
  - $f_i(\cdot)$ is the weak learner
- Like bagging, but with different learning strategies
  - Bagging: learn parallel, independently, fixed model complexity
  - Boosting: learn sequentially, dynamic model complexity

Bagging:

$$\begin{tikzcd}
{} \arrow[r, Rightarrow] & \boxed{f_1(X)} \\
{} \arrow[r, Rightarrow] & \boxed{f_2(X)} \\
{} \arrow[r, Rightarrow] & \boxed{f_3(X)}
\end{tikzcd}$$

Boosting:

$$\begin{tikzcd}
{} \arrow[r, Rightarrow] & \boxed{f_1(X)} \arrow[r, Rightarrow] & \boxed{f_2(X)} \arrow[r, Rightarrow] & \boxed{f_3(X)}
\end{tikzcd}$$

Boosting

- Greedy stage-wise approximation
  - Learn F_1, then $F_2, F_3, \dots$
  - Emphases error on each iteration
  - $L(F_m(X), Y) < L(F_{m-1}(X), Y)$
- AdaBoost
  - Emphases error by changing the distribution of samples
- Gradient Boosting
  - Emphases error by changing training targets

Gradient Boosting

- We want to get $f_m(X)$ that satisfies
  - $L(F_{m-1}(X) + f_m(X), Y) < L(F_{m-1}(X), Y)$
- Calculate the negative gradients
  - $\hat{y}_i = -\partial_{F_{m-1}(x_i)} l(F_{m-1}(x_i), y_i)$

L2 loss, $\hat{y}_i = y_i - F_{m-1}(x_i)$

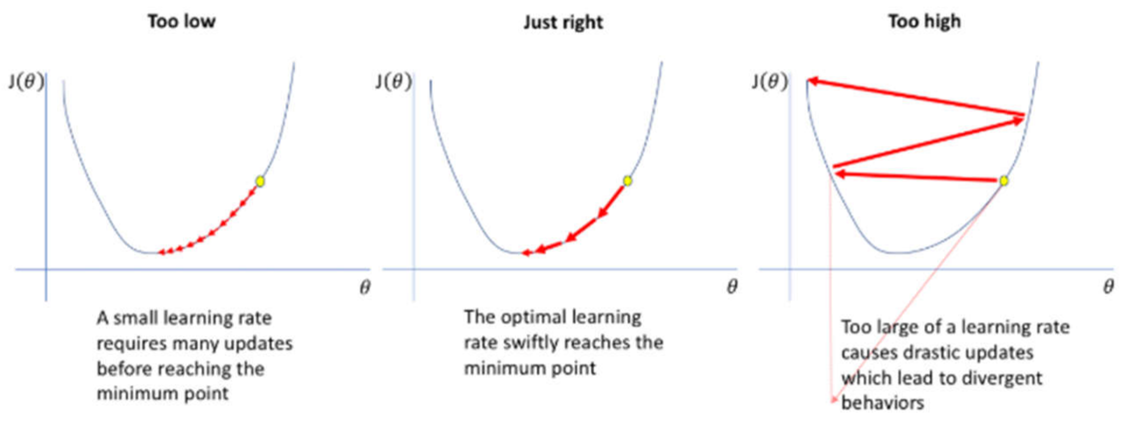
- Learn $f_m(X)$ to fit $\hat{Y}$ by using L2 loss
  - $f_m(X) = \arg\min_{f(X)} \sum_{i=1}^{n} (f(x_i) - \hat{y}_i)^2$
- Prove by first order Taylor Expansion
  - $l(y_i, F_{m-1}(x_i) + f_m(x_i)) = l(y_i, F_{m-1}(x_i)) + \partial_{F_{m-1}(x_i)} l(F_{m-1}(x_i), y_i) f_m(x_i)$
  - And $f_m(x_i) \approx \hat{y}_i = -\partial_{F_{m-1}(x_i)} l(F_{m-1}(x_i), y_i)$
  - Then $l(y_i, F_{m-1}(x_i) + f_m(x_i)) \approx l(y_i, F_{m-1}(x_i)) - \hat{y}_i^2 < l(y_i, F_{m-1}(x_i))$

Stochastic Boosting

- Use a random subset in each iteration
  - Sub-rows: could be used when #data is relatively large
  - Sub-features: could be used at the most time
- Leverage the Bagging into Boosting framework
- Speed up the learning, as only use subset in training
- Better generalization ability, benefit from bagging

Shrinkage

- Shrinkage $f_m(X)$ on each iteration
    - $F_m(X) = F_{m-1}(X) + \gamma f_m(X),$ where γ is shrinkage rate
- Avoid too large optimization steps
    - Like the learning rate in stochastic optimization


Too low
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
Just right
J(θ)
θ
The optimal learning rate swiftly reaches the minimum point
Too high
J(θ)
θ
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Compared with Decision Tree

- Boosting adds new models, tree partitions data
- Boosting can use the full dataset on all stages, while tree can only use the data in that node
- Boosting can always increase model complexity, while tree cannot
- Boosting needs the weak model as its base learner, while tree doesn’t need

GBDT

GBDT

- GBDT = Gradient Boosting + Decision Tree
  - The combination of two greedy stage-wise approximation models
  - Solve the problems in both boosting and tree:
    - Boosting needs a weak learner
    - Tree cannot always increase its complexity
- For different task/application, the main difference is the loss function

```
Algorithm: **GBDT**
  **Input: Training data** $(\boldsymbol{X}, \boldsymbol{Y})$ **, Iteration** $\boldsymbol{M}$**,**
        **Loss function** $\boldsymbol{l}$**, number of leaf C**
  $F_0(X) = 0$
  **For** m in (1,M):
     ⊳ get the training targets
     **For all** $(x_i, y_i) \in (X, Y)$:
        $\boxed{y_i^m = -\partial_{F_{m-1}(x_i)} l(y_i, F_{m-1}(x_i))}$
     ⊳ use decision tree to fit targets
     $f_m(X) = DecisionTree(X, Y^m, C, L2Loss)$
     $F_m(X) = F_{m-1}(X) + \gamma f_m(X)$
```

Regression: $l(y_i, F_{m-1}(x_i)) = (y - F_{m-1}(x_i))^2$

Binary Classification: $\bar{y}_i = \frac{1}{1+e^{-F_{m-1}(x_i)}}$

$$l(y_i, \bar{y}_i) = y_i \log \bar{y}_i + (1 - y_i)\log(1 - \bar{y}_i)$$

Lambdarank: Using $y_i^m = \lambda_i = \sum_{i<j} \lambda_{ij} - \sum_{j<i} \lambda_{ji}$

$$\lambda_{ij} = \frac{-\sigma|\Delta Ndcg_{ij}|}{1 + e^{\sigma(F_{m-1}(x_i) - F_{m-1}(x_j))}},$$

GBDT Hyper-parameter Tuning

- Most important hyper-parameters
  - Number of iterations M, shrinkage rate γ, number of leaves C
- Leverage early-stopping to reduce the tuning efforts
  - Fix M to a large value, e.g. 1000, and γ to a small value, e.g. 0.05
  - Use early-stopping to find a near-optimal combination of M and γ
  - With some different C
  - May need to try different γ

GBDT vs. Deep Learning

- Deep Learning (NN) is the model with human prior knowledge
  - Special structures designed by human to automatically extract useful information from data
    - CNN: “local receptive fields” from human vision
    - RNN: context in text/speech
  - Therefore, DL works very well for image, text and speech
  - However, need to design a new structure when applied in new tasks/data
- GBDT is a powerful function approximator, with excellent trade-off between bias and variance
  - No special design in models, can approximate all kinds of distribution
  - Therefore, GBDT works well for the tabular data in online tasks, such as click prediction, recommendation, etc.
  - However, GBDT doesn’t have prior knowledge to extract useful information, therefore, feature engineering is often needed for better performance

GBDT vs. Deep Learning

	GBDT	NN
Tasks	For all kinds of tabular data	Image, Text, Speech
Human efforts	Feature engineering	Architecture Design; Hyper-parameter tuning
Resource consumption	CPU	GPU

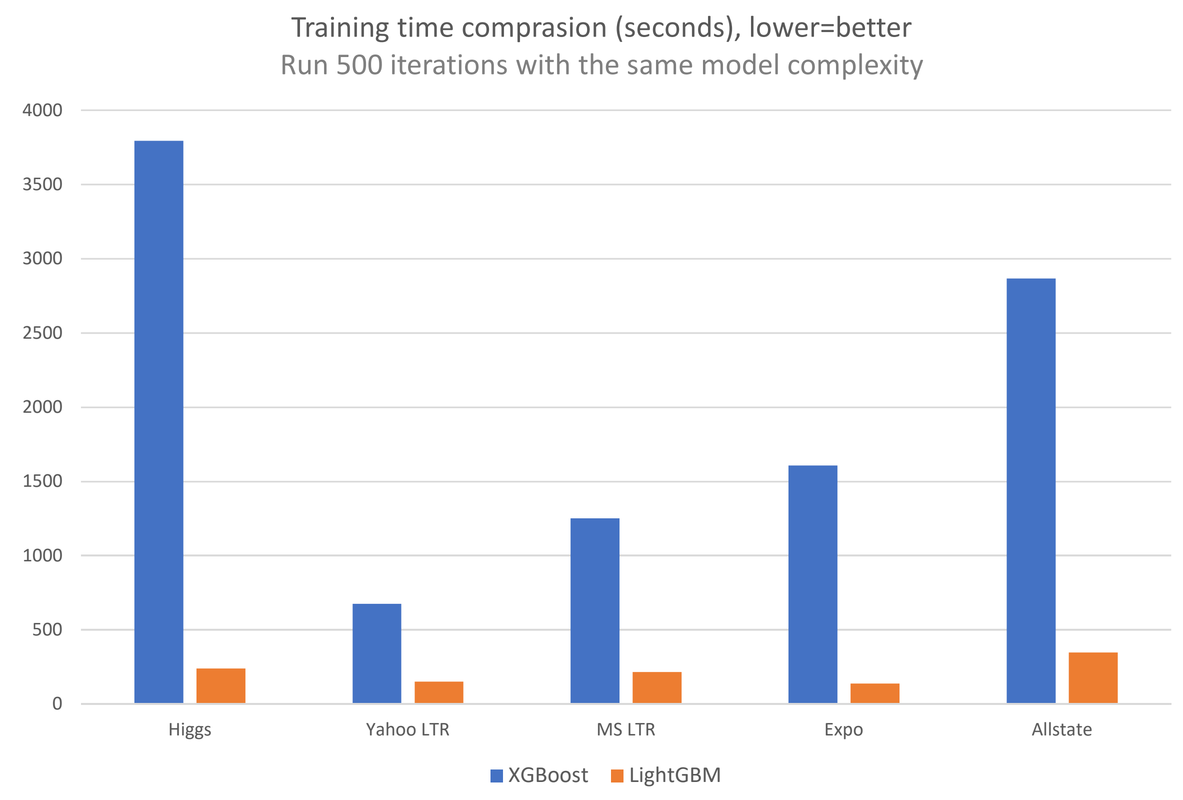
GBDT vs. Random Forest

- Both GBDT and RF are the ensemble of trees
- Both of them are widely-used in tabular data
- GBDT is based on boosting, RF is based on Bagging
    - Bagging could be used in GBDT as well
- In most cases, GBDT is better than RF

GBDT Tools

- XGBoost, 2014, https://github.com/dmlc/xgboost
    - Pre-sorted with level wise algorithm
    - The first high performance GBDT tool and remaining its popularity
- LightGBM, 2016, https://github.com/Microsoft/LightGBM
    - Histogram with leaf wise algorithm
    - The fastest GBDT tool and becoming more and more popular

GBDT Tools



Speed Comparison:

Data	XGBoost	LightGBM	speed up
Higgs	3794.34	238.5055	16x
Yahoo LTR	674.322	150.1864	4.5x
MS LTR	1251.27	215.3203	6x
Expo	1607.35	138.5042	11.6x
Allstate	2867.22	348.0845	8.2x

Memory consumption:

Data	XGBoost	LightGBM
Higgs	4.853GB	0.868GB
Yahoo LTR	1.907GB	0.831GB
MS LTR	5.469GB	0.886GB
Expo	1.553GB	0.543GB
Allstate	6.237GB	1.027GB

Course Project

- Incremental learning algorithm for GBDT
  - The traditional GBDT often needs to train from full dataset to achieve the good performance. Therefore, using GBDT in online applications is inefficient due to the frequently retraining. If GBDT model could be efficiently updated by the new data, like neural networks, GBDT could be used in more scenarios.
- Dataset: Criteo
  - Download link: https://drive.google.com/file/d/1Ytnf-2F_SBem-ydxeFB_O1NKhHH8ccXS/view?usp=sharing
- Evolution
  - Simulate the online update, by partition data into 6 batches, chronologically. ratios: [0.5, 0.1, 0.1, 0.1, 0.1, 0.1]. Then at i-th batch, only can use the data in that batch to update the model, and use the data in i+1 th batch for the evaluation, by AUC score. The final score is the average of 5 evaluations.

Course Project

- Baseline:
  - `refit` in LightGBM: https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.Booster.html#lightgbm.Booster.refit
- Submission:
  - Write a detail report about your algorithm and the experiment results
  - Write your code based on LightGBM, provide the GitHub link of your code
- Contact guolin.ke@Microsoft.com for more questions

Reference

- [1] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. Annals of statistics, pages 1189–1232, 2001.
- [2] Ruoming Jin and Gagan Agrawal. Communication and memory efficient parallel decision tree construction. In Proceedings of the 2003.
- [3] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22Nd ACM SIGKDD International. Conference on Knowledge Discovery and Data Mining, pages 785–794. ACM, 2016.
- [4] Stephen Tyree, Kilian Q Weinberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In Proceedings of the 20th international conference on World wide web, pages 387–396. ACM, 2011.
- [5] Jerome H Friedman. Stochastic gradient boosting. Computational Statistics & Data Analysis, 38(4):367–378, 2002.
- [6] Zhi-Hua Zhou. Ensemble methods: foundations and algorithms. CRC press, 2012.
- [7] Haijian Shi. Best-first decision tree learning. PhD thesis, The University of Waikato, 2007.
- [8] Ke, et al. "Lightgbm: A highly efficient gradient boosting decision tree." Advances in Neural Information Processing Systems. 2017.

Newton Optimization for GBDT

- Denote
  - $\bar{y}_i^{m-1} = F_{m-1}(x_i), \mathrm{g_i} = \partial_{\bar{y}_i^{m-1}} l\left(\bar{y}_i^{m-1}, y_i\right), \mathrm{h_i} = \partial^2_{\bar{y}_i^{m-1}} l\left(\bar{y}_i^{m-1}, y_i\right)$
- Using second-order Taylor expansion
  - $L(F_{m-1}(X) \ + f(X), Y) = \sum_{i=1}^{n} l(\bar{y}_i^{m-1} + f(x_i), y_i)$
    $\approx \ \sum_{i=1}^{n}(l\left(\bar{y}_i^{m-1}, y_i\right) + g_i f(x_i) + \frac{1}{2} h_i f^2(x_i))$
    $\equiv \sum_{i=1}^{n}(g_i f(x_i) + \frac{1}{2} h_i f^2(x_i))$
- For the samples in leaf q, with output r, the loss is
  - $\sum_{i \in q}(g_i r + \frac{1}{2} h_i \mathrm{r}^2) = \left(\sum_{i \in q} g_i\right) r + \frac{1}{2}\left(\sum_{i \in q} h_i\right) r^2$
- For quadratic function, the minimal loss is
  - $\min_r Gr + \frac{1}{2} H r^2 = -\frac{G^2}{2H}, \ \ where\ G = \sum_{i \in q} g_i\, , H = \sum_{i \in q} h_i\, , H > 0$
  - When $\mathrm{r} = -\frac{G}{H}$
- Therefore, we can use $-\frac{G^2}{H}$ to guide the tree learning greedily

Newton Optimization for GBDT

- Split leaf P into leaf L and leaf R
  - Delta loss: $\Delta loss = loss_P \ - loss_{left} - loss_{right} = \frac{G_{left}^2}{H_{left}} + \frac{G_{right}^2}{H_{right}} - \frac{G_P^2}{H_P} > 0$
  - The constant factor $\frac{1}{2}$ is removed for it doesn't change $\Delta loss$
  - The best split will have maximal $\Delta loss$
- We can use $\Delta loss$ to find best split and grow trees
- Revisit solutions
  - Solution 1, use L2 loss (towards gradients) to learn decision tree
    - $\Delta loss = \frac{G_L^2}{n_L} + \frac{G_R^2}{n_R} - \frac{G_P^2}{n_P}$
  - Solution 2, newton optimization
    - $\Delta loss = \frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_P^2}{H_P}$
- They have same structure, We can construct histogram for G, H

GBDT in Practice: Feature Engineering

- Basic preprocessing
  - Categorical feature handle
- Human/domain Knowledge
  - Holiday in demand forecasting
- Feature Selection
  - Often not needed, more is better
    - GBDT can automatically select features by itself
- Feature Combination
  - Generate new features by existing features
  - Leverage domain Knowledge, e.g. combine sex with age

GBDT in Practice: Hyper-parameter Tuning

- Try different tree sizes (depth and leaves), learning rates, with early stopping to determine the best number of trees
- Always enable sub-row and sub features
  - Faster learning speed and better generalization ability
- Regularization
  - min number of data per leaf
  - l1 and l2 for leaf outputs

GBDT in Practice: Auto Hyper-parameter Tuning

- Define the search range of each hyper-parameter
- Search Methods
  - Grid Search: search all combination, sequentially
  - Random Search: like grid search, but search randomly
  - Bayesian optimization: based on Bayesian theory, reduce the search space when searching
- Open-source solutions
  - https://github.com/microsoft/nni
  - https://github.com/automl/auto-sklearn
  - https://github.com/pfnet/optuna
- Example:
  - https://github.com/microsoft/nni/tree/master/examples/trials/auto-gbdt