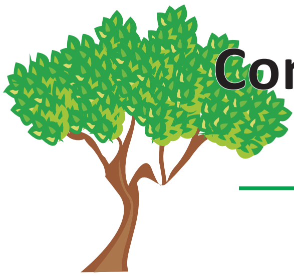


Construction of trees with parameters in XGBoost

Based on: https://xgboost.readthedocs.io/en/latest/parameter.html

Input: training set

...	Feature_i	...
...	value_1	...
...	value_2	...
...	value_3	...
...	...	...
...	value_N	...

Random subset for tree

...	Feature_i	...
...	value_1	...
...	value_2	...
...	value_3	...
...	...	...
...	value_N	...

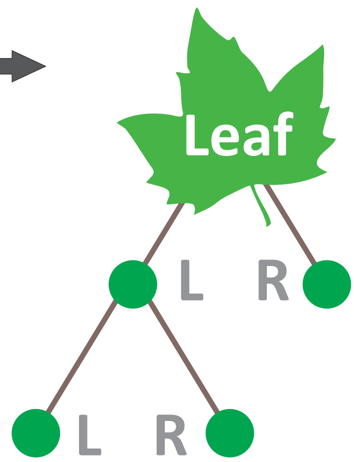
subsample

colsample_bytree
also
colsample_bylevel, colsample_bynode

Initial prediction

y_pred
pred_1
pred_2
pred_3
...
pred_N

base_score



Parameters for one tree

booster

- *gblinear*: Generalized Linear Model
- *gbtree*: Gradient Boosted Trees. Default
- *dart*: Dropout Additive Regression Trees

tree_method

- *exact / gpu_exact*: Exact greedy algorithm. No bins. Default for small & medium sets
- *approx*: Approximate algorithm. With bins, uses **sketch_eps**. Default for big sets
- *hist / gpu_hist*: Fast approximate algorithm. With bins, uses **max_bin**.
  Uses **grow_policy** for adding new leaves:
  - *depthwise*: split at nodes closest to the root. Default
  - *lossguide*: split at nodes with highest loss change, like LGBM.
    Uses **max_leaves**

updater - an advanced parameter that is usually set automatically.

- *grow_histmaker*: Row-based data splitting. Global histogram counting for bins
- *grow_local_histmaker*: Row-based data splitting. Local histogram counting for bins
- *grow_colmaker*: Column-based construction of trees. Default
- *prune*: Prunes the splits where the best gain < **gamma**. Default

max_depth - maximum depth of a tree. 0 indicates no limit

scale_pos_weight - control the balance of positive and negative weights (binary classification)
Useful for unbalanced classes

Next tree if possible

Parameters for boosting

train(

- **num_boost_round** - number of boosting iterations
- **early_stopping_rounds** - validation error needs to decrease at least every such number of rounds to continue training.

)

y_pred
pred_1
pred_2
pred_3
...
pred_N

= **learning_rate** *

weigths
w_1
w_2
w_3
...
w_N

w_i - weight for the leaf, corresponding i-th row of data

One-leaf calculation with parameters in XGBoost

Based on: https://arxiv.org/pdf/1603.02754.pdf
https://github.com/dmlc/xgboost/blob/master/src/tree/param.h

Input: prediction from previous boosting iteration → **The first and second order gradient statistics on the loss(y_iter, y_pred)**

y_pred
pred_1
pred_2
pred_3
…
pred_N'

Gradients	Hessians
grad_1	hess_1
grad_2	hess_2
grad_3	hess_3
…	…
grad_N'	hess_N'
sum_grad	**sum_hess**

Input: subset of features → **Binned feature by percentiles**

…	Feature_i	…
…	value_1	…
…	value_2	…
…	value_3	…
…	…	…
…	value_N'	…

Feature_i
bin_1
…
bin_M

max_bin
sketch_eps

If local binarization is used, binning is done here.
If global binarization is used, each leaf gets binned features as input.
If binarization is not used (Exact greedy algorithm), the features themselves will be splitted.

Weight of leaf

$$\text{new sum\_grad} = \begin{cases} \text{sum\_grad} - \textbf{alpha}, & \text{sum\_grad} > \textbf{alpha} \\ \text{sum\_grad} + \textbf{alpha}, & \text{sum\_grad} < -\textbf{alpha} \\ 0, & \text{else} \end{cases}$$

$$W = \begin{cases} -\dfrac{\text{new sum\_grad}}{\text{sum\_hess} + \textbf{lambda}}, & \text{sum\_hess} >= \textbf{min\_child\_weight} \\ 0, & \text{else} \end{cases}$$

$$\text{final\_W} = \begin{cases} \textbf{max\_delta\_step}, & W > \textbf{max\_delta\_step} > 0 \\ -\textbf{max\_delta\_step}, & W < -\textbf{max\_delta\_step} < 0 \\ W, & \text{else} \end{cases}$$

Sorted bins with sums of statistics

	Feature_i	Gradients	Hessians
↑ L subset	bin_i1	grads_i1	hess_i1
R subset ↓	bin_i2	grads_i2	hess_i2
	…	…	…

Best split

$$\text{Gain} = \begin{cases} \text{new sum\_grad} * \text{final\_W} + \\ 0.5 * (\text{sum\_hess} + \textbf{lambda}) * \text{final\_W}^\wedge 2 + , & \text{sum\_hess} >= \textbf{min\_child\_weight} \\ \textbf{alpha} * |\text{final\_W}| & \\ 0, & \text{else} \end{cases}$$

Gain split = Gain_L + Gain_R - Gain_All

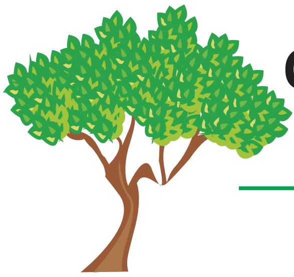
Best gain = Max { Gain split : for each split on L/R subsets, for each feature **}**

Best split = argMax { Gain split : for each split on L/R subsets, for each feature **}**

Outputs:

Weight of leaf
Best gain

Feature with best gain
L/R subsets for best gain



Construction of trees with parameters in LGBM

Based on: https://media.readthedocs.org/pdf/lightgbm/latest/lightgbm.pdf
https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

Input: training set → **Exclusive Feature Bundling (EFB)** → **Random subset for tree** → **Initial prediction**

Input: training set

...	Feature_i	...
...	value_1	...
...	value_2	...
...	value_3	...
...	...	...
...	value_N	...

Exclusive Feature Bundling (EFB)

Feature_i	Feature_k
0	value_1
value_2	0
0	0
...	...
0	value_N

→

Bundle_j
value_1
value_2
0
...
value_N

enable_bundle
max_conflict_rate

Random subset for tree

...	Bundle_j	...
...	value_1	...
...	value_2	...
...	value_3	...
...	...	...
...	value_N	...

bagging_freq, bagging_fraction
(not for "goss")

feature_fraction

Initial prediction

y_pred
pred_1
pred_2
pred_3
...
pred_N

init_score
initscore_filename
valid_data_initscores

Parameters for one tree

boosting

- *rf*: Random Forest. Not boosting
- *gbdt*: Gradient Boosted Decision Trees. Default
- *dart*: Dropout Additive Regression Trees
- *goss*: Gradient-based One-Side Sampling. Using "smart" subsampling with parameters:
  - **top_rate** - sampling ratio of large gradient data. If 0, other_rate ~ bagging_fraction
  - **other_rate** - sampling ratio of small gradient data

max_depth - maximum depth of a tree. <=0 indicates no limit

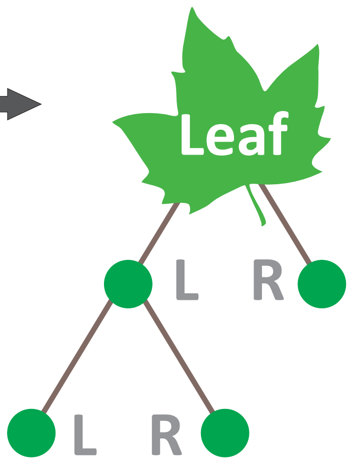
min_gain_to_split - prune by minimum loss requirement (post training regularization)

min_data_in_leaf - prune by minimum number of observations requirement

num_leaves - maximum number of leaves in one tree

scale_pos_weight - control the balance of positive and negative weights (binary classification).
Useful for unbalanced classes.
Cannot be used at the same time with is_unbalance

is_unbalance - set this to true if training data are unbalanced (binary classification).
Useful for unbalanced classes
Cannot be used at the same time with scale_pos_weight



Next tree if possible

Parameters for boosting

train(

num_iterations - number of boosting iterations

early_stopping_rounds - validation error needs to decrease at least every such number of rounds to continue training.

)

y_pred
pred_1
pred_2
pred_3
...
pred_N

= **learning_rate** *

weigths
w_1
w_2
w_3
...
w_N

w_i - weight for the leaf, corresponding i-th row of data

One-leaf calculation with parameters in LGBM

Based on: https://media.readthedocs.org/pdf/lightgbm/latest/lightgbm.pdf
https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

Input: prediction from previous boosting iteration

y_pred
pred_1
pred_2
pred_3
…
pred_N'

The first and second order gradient statistics on the loss(y_iter, y_pred)

Gradients	Hessians
grad_1	hess_1
grad_2	hess_2
grad_3	hess_3
…	…
grad_N'	hess_N'
sum_grad	**sum_hess**

Input: subset of bundles

…	Bundle_j	…
…	value_1	…
…	value_2	…
…	value_3	…
…	…	…
…	value_N'	…

Binned bundles by percentiles

Bundle_j
bin_1
…
bin_M

max_bin
min_data_in_bin
bin_construct_sample_cnt
max_cat_threshold
max_cat_to_onehot
cat_l2

Weight of leaf

$$\text{new sum\_grad} = \begin{cases} \text{sum\_grad} - \text{lambda\_l1}, & \text{sum\_grad} > \text{lambda\_l1} \\ \text{sum\_grad} + \text{lambda\_l1}, & \text{sum\_grad} < -\text{lambda\_l1} \\ 0, & \text{else} \end{cases}$$

$$W = \begin{cases} -\dfrac{\text{new sum\_grad}}{\text{sum\_hess} + \text{lambda\_l2}}, & \text{sum\_hess} >= \text{min\_sum\_hessian\_in\_leaf} \\ 0, & \text{else} \end{cases}$$

$$\text{final\_W} = \begin{cases} \text{max\_delta\_step}, & W > \text{max\_delta\_step} > 0 \\ -\text{max\_delta\_step}, & W < -\text{max\_delta\_step} < 0 \\ W, & \text{else} \end{cases}$$

Sorted bins with sums of statistics

	Bundle_j	Gradients	Hessians
↑ **L** subset	bin_i1	grads_i1	hess_i1
R subset ↓	bin_i2	grads_i2	hess_i2
	…	…	…

Best split

$$\text{Gain} = \begin{cases} \text{new sum\_grad} * \text{final\_W} + 0.5 * (\text{sum\_hess} + \text{lambda\_l2}) * \text{final\_W\^{}2} + \text{lambda\_l1} * |\text{final\_W}|, & \text{sum\_hess} >= \text{min\_sum\_hessian\_in\_leaf} \\ 0, & \text{else} \end{cases}$$

Gain split = Gain_L + Gain_R

Best gain = Max { Gain split : for each split on L/R subsets, for each bundle **}**

Best split = argMax { Gain split : for each split on L/R subsets, for each bundle **}**

Outputs:

Weight of leaf
Best gain

Bundle with best gain
L/R subsets for best gain