XGBoost Parameters

Before running XGBoost, we must set three types of parameters: general parameters, booster parameters and task parameters.

- **General parameters** relate to which booster we are using to do boosting, commonly tree or linear model
- **Booster parameters** depend on which booster you have chosen
- **Learning task parameters** decide on the learning scenario. For example, regression tasks may use different parameters with ranking tasks.
- **Command line parameters** relate to behavior of CLI version of XGBoost.

Note

Parameters in R package

In R-package, you can use `.` (dot) to replace underscore in the parameters, for example, you can use `max.depth` to indicate `max_depth`. The underscore parameters are also valid in R.

- Global Configuration
- General Parameters
  - Parameters for Tree Booster
  - Additional parameters for `hist`, `gpu_hist` and `approx` tree method
  - Additional parameters for Dart Booster (`booster=dart`)
  - Parameters for Linear Booster (`booster=gblinear`)
- Learning Task Parameters
  - Parameters for Tweedie Regression (`objective=reg:tweedie`)
  - Parameter for using Pseudo-Huber (`reg:pseudohubererror`)
- Command Line Parameters

Global Configuration

The following parameters can be set in the global scope, using `xgboost.config_context()` (Python) or `xgb.set.config()` (R).

- `verbosity`: Verbosity of printing messages. Valid values of 0 (silent), 1 (warning), 2 (info), and 3 (debug).
- `use_rmm`: Whether to use RAPIDS Memory Manager (RMM) to allocate GPU memory. This option is only applicable when XGBoost is built (compiled) with the RMM plugin enabled. Valid values are `true` and `false`.

General Parameters

- `booster` [default= `gbtree`]
  - Which booster to use. Can be `gbtree`, `gblinear` or `dart`; `gbtree` and `dart` use tree based models while `gblinear` uses linear functions.
- `verbosity` [default=1]
  - Verbosity of printing messages. Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug). Sometimes XGBoost tries to change configurations based on heuristics, which is displayed as warning message. If there's unexpected behaviour, please try to increase value of verbosity.
- `validate_parameters` [default to `false`, except for Python, R and CLI interface]
  - When set to True, XGBoost will perform validation of input parameters to check whether a parameter is used or not. The feature is still experimental. It's expected to have some false positives.
- `nthread` [default to maximum number of threads available if not set]
  - Number of parallel threads used to run XGBoost. When choosing it, please keep thread contention and hyperthreading in mind.
- `disable_default_eval_metric` [default= `false`]
  - Flag to disable default metric. Set to 1 or `true` to disable.
- `num_feature` [set automatically by XGBoost, no need to be set by user]
  - Feature dimension used in boosting, set to maximum dimension of the feature

Parameters for Tree Booster

- `eta` [default=0.3, alias: `learning_rate`]
  - Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- `gamma` [default=0, alias: `min_split_loss`]
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
  - range: [0,∞]

- `max_depth` [default=6]
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. `exact` tree method requires non-zero value.
  - range: [0,∞]
- `min_child_weight` [default=1]
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than `min_child_weight`, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger `min_child_weight` is, the more conservative the algorithm will be.
  - range: [0,∞]
- `max_delta_step` [default=0]
  - Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.
  - range: [0,∞]
- `subsample` [default=1]
  - Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.
  - range: (0,1]
- `sampling_method` [default= `uniform`]
  - The method to use to sample the training instances.
  - `uniform`: each training instance has an equal probability of being selected. Typically set `subsample` >= 0.5 for good results.
  - `gradient_based`: the selection probability for each training instance is proportional to the *regularized absolute value* of gradients (more specifically, $\sqrt{g^2+\lambda h^2}$). `subsample` may be set to as low as 0.1 without loss of model accuracy. Note that this sampling method is only supported when `tree_method` is set to `gpu_hist`; other tree methods only support `uniform` sampling.
- `colsample_bytree`, `colsample_bylevel`, `colsample_bynode` [default=1]

    - This is a family of parameters for subsampling of columns.
    - All `colsample_by*` parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
    - `colsample_bytree` is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
    - `colsample_bylevel` is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.
    - `colsample_bynode` is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
    - `colsample_by*` parameters work cumulatively. For instance, the combination `{'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5}` with 64 features will leave 8 features to choose from at each split.

      Using the Python or the R package, one can set the `feature_weights` for DMatrix to define the probability of each feature being selected when using column sampling. There's a similar parameter for `fit` method in sklearn interface.

- `lambda` [default=1, alias: `reg_lambda`]
    - L2 regularization term on weights. Increasing this value will make model more conservative.
- `alpha` [default=0, alias: `reg_alpha`]
    - L1 regularization term on weights. Increasing this value will make model more conservative.
- `tree_method` string [default= `auto`]
    - The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.
    - XGBoost supports `approx`, `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist`.
    - Choices: `auto`, `exact`, `approx`, `hist`, `gpu_hist`, this is a combination of commonly used updaters. For other updaters like `refresh`, set the parameter `updater` directly.
        - `auto`: Use heuristic to choose the fastest method.
            - For small dataset, exact greedy (`exact`) will be used.
            - For larger dataset, approximate algorithm (`approx`) will be chosen. It's recommended to try `hist` and `gpu_hist` for higher performance with large dataset. (`gpu_hist`)has support for `external memory`.
            - Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.

    - `exact`: Exact greedy algorithm. Enumerates all split candidates.
    - `approx`: Approximate greedy algorithm using quantile sketch and gradient histogram.
    - `hist`: Faster histogram optimized approximate greedy algorithm.
    - `gpu_hist`: GPU implementation of `hist` algorithm.

- `sketch_eps` [default=0.03]
    - Only used for `updater=grow_local_histmaker`.
    - This roughly translates into `O(1 / sketch_eps)` number of bins. Compared to directly select number of bins, this comes with theoretical guarantee with sketch accuracy.
    - Usually user does not have to tune this. But consider setting to a lower number for more accurate enumeration of split candidates.
    - range: (0, 1)

- `scale_pos_weight` [default=1]
    - Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: `sum(negative instances) / sum(positive instances)`. See Parameters Tuning for more discussion. Also, see Higgs Kaggle competition demo for examples: R, py1, py2, py3.

- `updater`
    - A comma separated string defining the sequence of tree updaters to run, providing a modular way to construct and to modify the trees. This is an advanced parameter that is usually set automatically, depending on some other parameters. However, it could be also set explicitly by a user. The following updaters exist:
        - `grow_colmaker`: non-distributed column-based construction of trees.
        - `grow_histmaker`: distributed tree construction with row-based data splitting based on global proposal of histogram counting.
        - `grow_local_histmaker`: based on local histogram counting.
        - `grow_quantile_histmaker`: Grow tree using quantized histogram.
        - `grow_gpu_hist`: Grow tree with GPU.
        - `sync`: synchronizes trees in all distributed nodes.
        - `refresh`: refreshes tree's statistics and/or leaf values based on the current data. Note that no random subsampling of data rows is performed.
        - `prune`: prunes the splits where loss < min_split_loss (or gamma) and nodes that have depth greater than `max_depth`.

- `refresh_leaf` [default=1]
    - This is a parameter of the `refresh` updater. When this flag is 1, tree leafs as well as tree nodes' stats are updated. When it is 0, only node stats are updated.

- `process_type` [default= `default`]

    - A type of boosting process to run.
    - Choices: `default`, `update`
        - `default`: The normal boosting process which creates new trees.
        - `update`: Starts from an existing model and only updates its trees. In each boosting iteration, a tree from the initial model is taken, a specified sequence of updaters is run for that tree, and a modified tree is added to the new model. The new model would have either the same or smaller number of trees, depending on the number of boosting iterations performed. Currently, the following built-in updaters could be meaningfully used with this process type: `refresh`, `prune`. With `process_type=update`, one cannot use updaters that create new trees.
- `grow_policy` [default= `depthwise`]
    - Controls a way new nodes are added to the tree.
    - Currently supported only if `tree_method` is set to `hist`, `approx` or `gpu_hist`.
    - Choices: `depthwise`, `lossguide`
        - `depthwise`: split at nodes closest to the root.
        - `lossguide`: split at nodes with highest loss change.
- `max_leaves` [default=0]
    - Maximum number of nodes to be added. Not used by `exact` tree method.
- `max_bin`, [default=256]
    - Only used if `tree_method` is set to `hist`, `approx` or `gpu_hist`.
    - Maximum number of discrete bins to bucket continuous features.
    - Increasing this number improves the optimality of splits at the cost of higher computation time.
- `predictor`, [default= `auto`]
    - The type of predictor algorithm to use. Provides the same results but allows the use of GPU or CPU.
        - `auto`: Configure predictor based on heuristics.
        - `cpu_predictor`: Multicore CPU prediction algorithm.
        - `gpu_predictor`: Prediction using GPU. Used when `tree_method` is `gpu_hist`. When `predictor` is set to default value `auto`, the `gpu_hist` tree method is able to provide GPU based prediction without copying training data to GPU memory. If `gpu_predictor` is explicitly specified, then all data is copied into GPU, only recommended for performing prediction tasks.
- `num_parallel_tree`, [default=1]

- Number of parallel trees constructed during each iteration. This option is used to support boosted random forest.

- `monotone_constraints`
  - Constraint of variable monotonicity. See Monotonic Constraints for more information.
- `interaction_constraints`
  - Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nest list, e.g. `[[0, 1], [2, 3, 4]]`, where each inner list is a group of indices of features that are allowed to interact with each other. See Feature Interaction Constraints for more information.

Additional parameters for `hist`, `gpu_hist` and `approx` tree method

- `single_precision_histogram`, [default= `false`]
  - Use single precision to build histograms instead of double precision.
- `max_cat_to_onehot`

  New in version 1.6.

  > **Note**
  >
  > The support for this parameter is experimental.

  - A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Only relevant for regression and binary classification. Also, `exact` tree method is not supported

Additional parameters for Dart Booster (`booster=dart`)

> **Note**
>
> Using `predict()` with DART booster

If the booster object is DART type, `predict()` will perform dropouts, i.e. only some of the trees will be evaluated. This will produce incorrect results if `data` is not the training data. To obtain correct results on test sets, set `iteration_range` to a nonzero value, e.g.

```
preds = bst.predict(dtest, iteration_range=(0, num_round))
```

- `sample_type` [default= `uniform`]
  - Type of sampling algorithm.
    - `uniform` : dropped trees are selected uniformly.
    - `weighted` : dropped trees are selected in proportion to weight.
- `normalize_type` [default= `tree`]
  - Type of normalization algorithm.
    - `tree` : new trees have the same weight of each of dropped trees.
      - Weight of new trees are `1 / (k + learning_rate)` .
      - Dropped trees are scaled by a factor of `k / (k + learning_rate)` .
    - `forest` : new trees have the same weight of sum of dropped trees (forest).
      - Weight of new trees are `1 / (1 + learning_rate)` .
      - Dropped trees are scaled by a factor of `1 / (1 + learning_rate)` .
- `rate_drop` [default=0.0]
  - Dropout rate (a fraction of previous trees to drop during the dropout).
  - range: [0.0, 1.0]
- `one_drop` [default=0]
  - When this flag is enabled, at least one tree is always dropped during the dropout (allows Binomial-plus-one or epsilon-dropout from the original DART paper).
- `skip_drop` [default=0.0]
  - Probability of skipping the dropout procedure during a boosting iteration.
    - If a dropout is skipped, new trees are added in the same manner as `gbtree` .
    - Note that non-zero `skip_drop` has higher priority than `rate_drop` or `one_drop` .
  - range: [0.0, 1.0]

Parameters for Linear Booster (`booster=gblinear`)

- `lambda` [default=0, alias: `reg_lambda`]
  - L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
- `alpha` [default=0, alias: `reg_alpha`]
  - L1 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
- `updater` [default= `shotgun`]
  - Choice of algorithm to fit linear model
    - `shotgun` : Parallel coordinate descent algorithm based on shotgun algorithm. Uses ‘hogwild’ parallelism and therefore produces a nondeterministic solution on each run.

        - `coord_descent`: Ordinary coordinate descent algorithm. Also multithreaded but still produces a deterministic solution.
- `feature_selector` [default= `cyclic`]
    - Feature selection and ordering method
        - `cyclic`: Deterministic selection by cycling through features one at a time.
        - `shuffle`: Similar to `cyclic` but with random feature shuffling prior to each update.
        - `random`: A random (with replacement) coordinate selector.
        - `greedy`: Select coordinate with the greatest gradient magnitude. It has `O(num_feature^2)` complexity. It is fully deterministic. It allows restricting the selection to `top_k` features per group with the largest magnitude of univariate weight change, by setting the `top_k` parameter. Doing so would reduce the complexity to `O(num_feature*top_k)`.
        - `thrifty`: Thrifty, approximately-greedy feature selector. Prior to cyclic updates, reorders features in descending magnitude of their univariate weight changes. This operation is multithreaded and is a linear complexity approximation of the quadratic greedy selection. It allows restricting the selection to `top_k` features per group with the largest magnitude of univariate weight change, by setting the `top_k` parameter.
- `top_k` [default=0]
    - The number of top features to select in `greedy` and `thrifty` feature selector. The value of 0 means using all the features.

Learning Task Parameters

Specify the learning task and the corresponding learning objective. The objective options are below:

- `objective` [default=reg:squarederror]
    - `reg:squarederror`: regression with squared loss.
    - `reg:squaredlogerror`: regression with squared log loss $\frac{1}{2}[log(pred + 1) - log(label + 1)]^2$. All input labels are required to be greater than -1. Also, see metric `rmsle` for possible issue with this objective.
    - `reg:logistic`: logistic regression.
    - `reg:pseudohubererror`: regression with Pseudo Huber loss, a twice differentiable alternative to absolute loss.
    - `binary:logistic`: logistic regression for binary classification, output probability
    - `binary:logitraw`: logistic regression for binary classification, output score before logistic transformation
    - `binary:hinge`: hinge loss for binary classification. This makes predictions of 0 or 1, rather than producing probabilities.

    - `count:poisson`: Poisson regression for count data, output mean of Poisson distribution.
        - `max_delta_step` is set to 0.7 by default in Poisson regression (used to safeguard optimization)
    - `survival:cox`: Cox regression for right censored survival time data (negative values are considered right censored). Note that predictions are returned on the hazard ratio scale (i.e., as HR = exp(marginal_prediction) in the proportional hazard function `h(t) = h0(t) * HR`).
    - `survival:aft`: Accelerated failure time model for censored survival time data. See Survival Analysis with Accelerated Failure Time for details.
    - `aft_loss_distribution`: Probability Density Function used by `survival:aft` objective and `aft-nloglik` metric.
    - `multi:softmax`: set XGBoost to do multiclass classification using the softmax objective, you also need to set num_class(number of classes)
    - `multi:softprob`: same as softmax, but output a vector of `ndata * nclass`, which can be further reshaped to `ndata * nclass` matrix. The result contains predicted probability of each data point belonging to each class.
    - `rank:pairwise`: Use LambdaMART to perform pairwise ranking where the pairwise loss is minimized
    - `rank:ndcg`: Use LambdaMART to perform list-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized
    - `rank:map`: Use LambdaMART to perform list-wise ranking where Mean Average Precision (MAP) is maximized
    - `reg:gamma`: gamma regression with log-link. Output is a mean of gamma distribution. It might be useful, e.g., for modeling insurance claims severity, or for any outcome that might be gamma-distributed.
    - `reg:tweedie`: Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any outcome that might be Tweedie-distributed.

- `base_score` [default=0.5]
    - The initial prediction score of all instances, global bias
    - For sufficient number of iterations, changing this value will not have too much effect.

- `eval_metric` [default according to objective]
    - Evaluation metrics for validation data, a default metric will be assigned according to objective (rmse for regression, and logloss for classification, mean average precision for ranking)
    - User can add multiple evaluation metrics. Python users: remember to pass the metrics in as list of parameters pairs instead of map, so that latter `eval_metric` won't override previous one
    - The choices are listed below:

- `rmse`: root mean square error
- `rmsle`: root mean square log error: $\sqrt{\frac{1}{N}[log(pred+1)-log(label+1)]^2}$. Default metric of `reg:squaredlogerror` objective. This metric reduces errors generated by outliers in dataset. But because `log` function is employed, `rmsle` might output `nan` when prediction value is less than -1. See `reg:squaredlogerror` for other requirements.
- `mae`: mean absolute error
- `mape`: mean absolute percentage error
- `mphe`: mean Pseudo Huber error. Default metric of `reg:pseudohubererror` objective.
- `logloss`: negative log-likelihood
- `error`: Binary classification error rate. It is calculated as `#(wrong cases)/#(all cases)`. For the predictions, the evaluation will regard the instances with prediction value larger than 0.5 as positive instances, and the others as negative instances.
- `error@t`: a different than 0.5 binary classification threshold value could be specified by providing a numerical value through 't'.
- `merror`: Multiclass classification error rate. It is calculated as `#(wrong cases)/#(all cases)`.
- `mlogloss`: Multiclass logloss.
- `auc`: Receiver Operating Characteristic Area under the Curve. Available for classification and learning-to-rank tasks.
    - When used with binary classification, the objective should be `binary:logistic` or similar functions that work on probability.
    - When used with multi-class classification, objective should be `multi:softprob` instead of `multi:softmax`, as the latter doesn't output probability. Also the AUC is calculated by 1-vs-rest with reference class weighted by class prevalence.
    - When used with LTR task, the AUC is computed by comparing pairs of documents to count correctly sorted pairs. This corresponds to pairwise learning to rank. The implementation has some issues with average AUC around groups and distributed workers not being well-defined.
    - On a single machine the AUC calculation is exact. In a distributed environment the AUC is a weighted average over the AUC of training rows on each node - therefore, distributed AUC is an approximation sensitive to the distribution of data across workers. Use another metric in distributed environments if precision and reproducibility are important.
    - When input dataset contains only negative or positive samples, the output is *NaN*. The behavior is implementation defined, for instance, `scikit-learn` returns 0.5 instead.
- `aucpr`: Area under the PR curve. Available for classification and learning-to-rank tasks.

After XGBoost 1.6, both of the requirements and restrictions for using `aucpr` in classification problem are similar to `auc`. For ranking task, only binary relevance label $y \in [0, 1]$ is supported. Different from `map (mean average precision)`, `aucpr` calculates the *interpolated* area under precision recall curve using continuous interpolation.

  - `ndcg`: Normalized Discounted Cumulative Gain
  - `map`: Mean Average Precision
  - `ndcg@n`, `map@n`: 'n' can be assigned as an integer to cut off the top positions in the lists for evaluation.
  - `ndcg-`, `map-`, `ndcg@n-`, `map@n-`: In XGBoost, NDCG and MAP will evaluate the score of a list without any positive samples as 1. By adding "-" in the evaluation metric XGBoost will evaluate these score as 0 to be consistent under some conditions.
  - `poisson-nloglik`: negative log-likelihood for Poisson regression
  - `gamma-nloglik`: negative log-likelihood for gamma regression
  - `cox-nloglik`: negative partial log-likelihood for Cox proportional hazards regression
  - `gamma-deviance`: residual deviance for gamma regression
  - `tweedie-nloglik`: negative log-likelihood for Tweedie regression (at a specified value of the `tweedie_variance_power` parameter)
  - `aft-nloglik`: Negative log likelihood of Accelerated Failure Time model. See Survival Analysis with Accelerated Failure Time for details.
  - `interval-regression-accuracy`: Fraction of data points whose predicted labels fall in the interval-censored labels. Only applicable for interval-censored data. See Survival Analysis with Accelerated Failure Time for details.

- `seed` [default=0]
  - Random number seed. This parameter is ignored in R package, use *set.seed()* instead.
- `seed_per_iteration` [default= `false`]
  - Seed PRNG determnisticly via iterator number.

Parameters for Tweedie Regression (`objective=reg:tweedie`)

- `tweedie_variance_power` [default=1.5]
  - Parameter that controls the variance of the Tweedie distribution `var(y) ~ E(y)^tweedie_variance_power`
  - range: (1,2)
  - Set closer to 2 to shift towards a gamma distribution
  - Set closer to 1 to shift towards a Poisson distribution.

Parameter for using Pseudo-Huber (`reg:pseudohubererror`)

- `huber_slope` : A parameter used for Pseudo-Huber loss to define the δ term. [default = 1.0]

Command Line Parameters

The following parameters are only used in the console version of XGBoost

- `num_round`
  - The number of rounds for boosting
- `data`
  - The path of training data
- `test:data`
  - The path of test data to do prediction
- `save_period` [default=0]
  - The period to save the model. Setting `save_period=10` means that for every 10 rounds XGBoost will save the model. Setting it to 0 means not saving any model during the training.
- `task` [default= `train`] options: `train` , `pred` , `eval` , `dump`
  - `train` : training using data
  - `pred` : making prediction for test:data
  - `eval` : for evaluating statistics specified by `eval[name]=filename`
  - `dump` : for dump the learned model into text format
- `model_in` [default=NULL]
  - Path to input model, needed for `test` , `eval` , `dump` tasks. If it is specified in training, XGBoost will continue training from the input model.
- `model_out` [default=NULL]
  - Path to output model after training finishes. If not specified, XGBoost will output files with such names as `0003.model` where `0003` is number of boosting rounds.
- `model_dir` [default= `models/`]
  - The output directory of the saved models during training
- `fmap`
  - Feature map, used for dumping model
- `dump_format` [default= `text`] options: `text` , `json`
  - Format of model dump file
- `name_dump` [default= `dump.txt`]
  - Name of model dump file
- `name_pred` [default= `pred.txt`]
  - Name of prediction file, used in pred mode
- `pred_margin` [default=0]
  - Predict margin instead of transformed probability