# pygbm Documentation

*Release 0.1.0.dev0*

Olivier Grisel, Nicolas Hug

Dec 15, 2018

# Estimator API

**Warning:** Pygbm's API and default values are likely to be changed in future version, without any deprecation cycle.

**Warning:** Pygbm's API and default values are likely to be changed in future version, without any deprecation cycle.

# CHAPTER 1

# Gradient Boosting Estimators

Gradient Boosting decision trees for classification and regression.

**class** pygbm.gradient_boosting.**GradientBoostingClassifier**(*loss='auto'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=256*, *scoring='neg_log_loss'*, *validation_split=0.1*, *n_iter_no_change=5*, *tol=1e-07*, *verbose=0*, *random_state=None*)

Scikit-learn compatible Gradient Boosting Tree for classification.

**Parameters**

- **loss** (*{'auto', 'binary_crossentropy', 'categorical_crossentropy'}, optional(default='auto')*) – The loss function to use in the boosting process. 'binary_crossentropy' (also known as logistic loss) is used for binary classification and generalizes to 'categorical_crossentropy' for multiclass classification. 'auto' will automatically choose either loss depending on the nature of the problem.
- **learning_rate** (*float, optional(default=1)*) – The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.
- **max_iter** (*int, optional(default=100)*) – The maximum number of iterations of the boosting process, i.e. the maximum number of trees for binary classification. For multiclass classification, *n_classes* trees per iteration are built.
- **max_leaf_nodes** (*int or None, optional(default=None)*) – The maximum number of leaves for each tree. If None, there is no maximum limit.

- **max_depth** (*int or None, optional (default=None)*) – The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf.
- **min_samples_leaf** (*int, optional (default=20)*) – The minimum number of samples per leaf.
- **l2_regularization** (*float, optional (default=0)*) – The L2 regularization parameter. Use 0 for no regularization.
- **max_bins** (*int, optional (default=256)*) – The maximum number of bins to use. Before training, each feature of the input array `X` is binned into at most `max_bins` bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. Must be no larger than 256.
- **scoring** (*str or callable or None, optional (default='accuracy')*) – Scoring parameter to use for early stopping (see sklearn.metrics for available options). If None, no early stopping is done.
- **validation_split** (*int or float or None, optional (default=0.1)*) – Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the whole training data.
- **n_iter_no_change** (*int, optional (default=5)*) – Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the ``n_iter_no_change - 1``th-to-last one, up to some tolerance.
- **tol** (*float or None optional (default=1e-7)*) – The absolute tolerance to use when comparing scores. The higher the tolerance, the more likely we are to early stop: higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.
- **verbose** (*int, optional (default=0)*) – The verbosity level. If not zero, print some information about the fitting process.
- **random_state** (*int, np.random.RandomStateInstance or None, optional (default=None)*) – Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See scikit-learn glossary.

## Examples

```
>>> from sklearn.datasets import load_iris
>>> from pygbm import GradientBoostingClassifier
>>> X, y = load_iris(return_X_y=True)
>>> clf = GradientBoostingClassifier().fit(X, y)
>>> clf.score(X, y)
0.97...
```

**fit** (*X, y*)
Fit the gradient boosting model.

**Parameters**

- **X** (*array-like, shape=(n_samples, n_features)*) – The input samples.
- **y** (*array-like, shape=(n_samples,)*) – Target values.

**Returns** self

**Return type** object

**get_params**(*deep=True*)

Get parameters for this estimator.

**Parameters** **deep** (*boolean, optional*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** **params** – Parameter names mapped to their values.

**Return type** mapping of string to any

**predict**(*X*)

Predict classes for X.

**Parameters** **X** (*array-like, shape=(n_samples, n_features)*) – The input samples. If `X.dtype == np.uint8`, the data is assumed to be pre-binned.

**Returns** **y** – The predicted classes.

**Return type** array, shape (n_samples,)

**predict_proba**(*X*)

Predict class probabilities for X.

**Parameters** **X** (*array-like, shape=(n_samples, n_features)*) – The input samples. If `X.dtype == np.uint8`, the data is assumed to be pre-binned.

**Returns** **p** – The class probabilities of the input samples.

**Return type** array, shape (n_samples, n_classes)

**score**(*X, y, sample_weight=None*)

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**

- **X** (*array-like, shape = (n_samples, n_features)*) – Test samples.
- **y** (*array-like, shape = (n_samples) or (n_samples, n_outputs)*) – True labels for X.
- **sample_weight** (*array-like, shape = [n_samples], optional*) – Sample weights.

**Returns** **score** – Mean accuracy of self.predict(X) wrt. y.

**Return type** float

**set_params**(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Returns**

**Return type** self

**class** `pygbm.gradient_boosting.`**`GradientBoostingRegressor`**(*loss='least_squares', learning_rate=0.1, max_iter=100, max_leaf_nodes=31, max_depth=None, min_samples_leaf=20, l2_regularization=0.0, max_bins=256, scoring='neg_mean_squared_error', validation_split=0.1, n_iter_no_change=5, tol=1e-07, verbose=0, random_state=None*)

Scikit-learn compatible Gradient Boosting Tree for regression.

**Parameters**

- **loss** (*{'least_squares'}, optional(default='least_squares')*) – The loss function to use in the boosting process.
- **learning_rate** (*float, optional(default=0.1)*) – The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.
- **max_iter** (*int, optional(default=100)*) – The maximum number of iterations of the boosting process, i.e. the maximum number of trees.
- **max_leaf_nodes** (*int or None, optional(default=None)*) – The maximum number of leaves for each tree. If None, there is no maximum limit.
- **max_depth** (*int or None, optional(default=None)*) – The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf.
- **min_samples_leaf** (*int, optional(default=20)*) – The minimum number of samples per leaf.
- **l2_regularization** (*float, optional(default=0)*) – The L2 regularization parameter. Use 0 for no regularization.
- **max_bins** (*int, optional(default=256)*) – The maximum number of bins to use. Before training, each feature of the input array `X` is binned into at most `max_bins` bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. Must be no larger than 256.
- **scoring** (*str or callable or None, optional (default='neg_mean_squared_error')*) – Scoring parameter to use for early stopping (see sklearn.metrics for available options). If None, no early stopping is done.
- **validation_split** (*int or float or None, optional(default=0.1)*) – Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the whole training data.
- **n_iter_no_change** (*int, optional (default=5)*) – Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the ‘‘n_iter_no_change - 1‘‘th-to-last one, up to some tolerance.
- **tol** (*float or None optional (default=1e-7)*) – The absolute tolerance to use when comparing scores. The higher the tolerance, the more likely we are to early stop:

higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.

- **verbose** (*int, optional (default=0)*) – The verbosity level. If not zero, print some information about the fitting process.
- **random_state** (*int, np.random.RandomStateInstance or None, optional (default=None)*) – Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See scikit-learn glossary.

### Examples

```
>>> from sklearn.datasets import load_boston
>>> from pygbm import GradientBoostingRegressor
>>> X, y = load_boston(return_X_y=True)
>>> est = GradientBoostingRegressor().fit(X, y)
>>> est.score(X, y)
0.92...
```

**fit** (*X*, *y*)

Fit the gradient boosting model.

**Parameters**

- **X** (*array-like, shape=(n_samples, n_features)*) – The input samples.
- **y** (*array-like, shape=(n_samples,)*) – Target values.

**Returns** self

**Return type** object

**get_params** (*deep=True*)

Get parameters for this estimator.

**Parameters** **deep** (*boolean, optional*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** **params** – Parameter names mapped to their values.

**Return type** mapping of string to any

**predict** (*X*)

Predict values for X.

**Parameters** **X** (*array-like, shape=(n_samples, n_features)*) – The input samples. If `X.dtype == np.uint8`, the data is assumed to be pre-binned.

**Returns** **y** – The predicted values.

**Return type** array, shape (n_samples,)

**score** (*X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

**Parameters**

- **X** (*array-like, shape = (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.
- **y** (*array-like, shape = (n_samples) or (n_samples, n_outputs)*) – True values for X.
- **sample_weight** (*array-like, shape = [n_samples], optional*) – Sample weights.

**Returns** **score** – R^2 of self.predict(X) wrt. y.

**Return type** float

**set_params**(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Returns**

**Return type** self

**Warning:** Pygbm's API and default values are likely to be changed in future version, without any deprecation cycle.

# CHAPTER 2

# Losses

This module contains the loss classes.

Specific losses are used for regression, binary classification or multiclass classification.

**class** pygbm.loss.**BinaryCrossEntropy**

Binary cross-entropy loss, for binary classification.

For a given sample x_i, the binary cross-entropy loss is defined as the negative log-likelihood of the model which can be expressed as:

```
loss(x_i) = log(1 + exp(raw_pred_i)) - y_true_i * raw_pred_i
```

See The Elements of Statistical Learning, by Hastie, Tibshirani, Friedman.

**get_baseline_prediction**(*y_train*, *prediction_dim*)

Return initial predictions (before the first iteration).

**Parameters**

- **y_train** (*array-like, shape=(n_samples,)*) – The target training values.
- **prediction_dim** (*int*) – The dimension of one prediction: 1 for binary classification and regression, n_classes for multiclass classification.

**Returns** **baseline_prediction** – The baseline prediction.

**Return type** float or array of shape (1, prediction_dim)

**init_gradients_and_hessians**(*n_samples*, *prediction_dim*)

Return initial gradients and hessians.

Unless hessians are constant, arrays are initialized with undefined values.

**Parameters**

- **n_samples** (*int*) – The number of samples passed to *fit()*

- **prediction_dim** (*int*) – The dimension of a raw prediction, i.e. the number of trees built at each iteration. Equals 1 for regression and binary classification, or K where K is the number of classes for multiclass classification.

**Returns**

- **gradients** (*array-like, shape=(n_samples * prediction_dim)*)
- **hessians** (*array-like, shape=(n_samples * prediction_dim).*) – If hessians are constant (e.g. for `LeastSquares` loss, shape is (1,) and the array is initialized to `1`.

**inverse_link_function = <ufunc 'expit'>**

**update_gradients_and_hessians**(*gradients*, *hessians*, *y_true*, *raw_predictions*)

Update gradients and hessians arrays, inplace.

The gradients (resp. hessians) are the first (resp. second) order derivatives of the loss for each sample with respect to the predictions of model, evaluated at iteration `i - 1`.

**Parameters**

- **gradients** (*array-like, shape=(n_samples * prediction_dim)*) – The gradients (treated as OUT array).
- **hessians** (*array-like, shape=(n_samples * prediction_dim) or (1,)*) – The hessians (treated as OUT array).
- **y_true** (*array-like, shape=(n_samples,)*) – The true target values or each training sample.
- **raw_predictions** (*array-like, shape=(n_samples, prediction_dim)*) – The raw_predictions (i.e. values from the trees) of the tree ensemble at iteration `i - 1`.

**class** pygbm.loss.**CategoricalCrossEntropy**

Categorical cross-entropy loss, for multiclass classification.

For a given sample x_i, the categorical cross-entropy loss is defined as the negative log-likelihood of the model and generalizes the binary cross-entropy to more than 2 classes.

**get_baseline_prediction**(*y_train*, *prediction_dim*)

Return initial predictions (before the first iteration).

**Parameters**

- **y_train** (*array-like, shape=(n_samples,)*) – The target training values.
- **prediction_dim** (*int*) – The dimension of one prediction: 1 for binary classification and regression, n_classes for multiclass classification.

**Returns** **baseline_prediction** – The baseline prediction.

**Return type** float or array of shape (1, prediction_dim)

**init_gradients_and_hessians**(*n_samples*, *prediction_dim*)

Return initial gradients and hessians.

Unless hessians are constant, arrays are initialized with undefined values.

**Parameters**

- **n_samples** (*int*) – The number of samples passed to *fit()*
- **prediction_dim** (*int*) – The dimension of a raw prediction, i.e. the number of trees built at each iteration. Equals 1 for regression and binary classification, or K where K is the number of classes for multiclass classification.

**Returns**

- **gradients** (*array-like, shape=(n_samples * prediction_dim)*)
- **hessians** (*array-like, shape=(n_samples * prediction_dim).*) – If hessians are constant (e.g. for `LeastSquares` loss, shape is (1,) and the array is initialized to `1`.

**update_gradients_and_hessians**(*gradients*, *hessians*, *y_true*, *raw_predictions*)
Update gradients and hessians arrays, inplace.

The gradients (resp. hessians) are the first (resp. second) order derivatives of the loss for each sample with respect to the predictions of model, evaluated at iteration `i - 1`.

**Parameters**

- **gradients** (*array-like, shape=(n_samples * prediction_dim)*) – The gradients (treated as OUT array).
- **hessians** (*array-like, shape=(n_samples * prediction_dim) or (1,)*) – The hessians (treated as OUT array).
- **y_true** (*array-like, shape=(n_samples,)*) – The true target values or each training sample.
- **raw_predictions** (*array-like, shape=(n_samples, prediction_dim)*) – The raw_predictions (i.e. values from the trees) of the tree ensemble at iteration `i - 1`.

**class** pygbm.loss.**LeastSquares**
Least squares loss, for regression.

For a given sample x_i, least squares loss is defined as:

```
loss(x_i) = (y_true_i - raw_pred_i)**2
```

**get_baseline_prediction**(*y_train*, *prediction_dim*)
Return initial predictions (before the first iteration).

**Parameters**

- **y_train** (*array-like, shape=(n_samples,)*) – The target training values.
- **prediction_dim** (*int*) – The dimension of one prediction: 1 for binary classification and regression, n_classes for multiclass classification.

**Returns** **baseline_prediction** – The baseline prediction.

**Return type** float or array of shape (1, prediction_dim)

**init_gradients_and_hessians**(*n_samples*, *prediction_dim*)
Return initial gradients and hessians.

Unless hessians are constant, arrays are initialized with undefined values.

**Parameters**

- **n_samples** (*int*) – The number of samples passed to *fit()*
- **prediction_dim** (*int*) – The dimension of a raw prediction, i.e. the number of trees built at each iteration. Equals 1 for regression and binary classification, or K where K is the number of classes for multiclass classification.

**Returns**

- **gradients** (*array-like, shape=(n_samples * prediction_dim)*)

- **hessians** (*array-like, shape=(n_samples * prediction_dim).*) – If hessians are constant (e.g. for `LeastSquares` loss, shape is (1,) and the array is initialized to `1`.

**`update_gradients_and_hessians`**(*gradients*, *hessians*, *y_true*, *raw_predictions*)

Update gradients and hessians arrays, inplace.

The gradients (resp. hessians) are the first (resp. second) order derivatives of the loss for each sample with respect to the predictions of model, evaluated at iteration `i - 1`.

**Parameters**

- **`gradients`** (*`array-like, shape=(n_samples * prediction_dim)`*) – The gradients (treated as OUT array).
- **`hessians`** (*`array-like, shape=(n_samples * prediction_dim) or (1,)`*) – The hessians (treated as OUT array).
- **`y_true`** (*`array-like, shape=(n_samples,)`*) – The true target values or each training sample.
- **`raw_predictions`** (*`array-like, shape=(n_samples, prediction_dim)`*) – The raw_predictions (i.e. values from the trees) of the tree ensemble at iteration `i - 1`.

# CHAPTER 3

# Binning

This module contains the BinMapper class.

BinMapper is used for mapping a real-valued dataset into integer-valued bins with equally-spaced thresholds.

**class** pygbm.binning.**BinMapper**(*max_bins=256*, *subsample=100000*, *random_state=None*)

Transformer that maps a dataset into integer-valued bins.

The bins are created in a feature-wise fashion, with equally-spaced quantiles.

Large datasets are subsampled, but the feature-wise quantiles should remain stable.

If the number of unique values for a given feature is less than `max_bins`, then the unique values of this feature are used instead of the quantiles.

**Parameters**

- **max_bins** (*int, optional (default=256)*) – The maximum number of bins to use. If for a given feature the number of unique values is less than `max_bins`, then those unique values will be used to compute the bin thresholds, instead of the quantiles.
- **subsample** (*int or None, optional (default=1e5)*) – If `n_samples > subsample`, then `sub_samples` samples will be randomly choosen to compute the quantiles. If `None`, the whole data is used.
- **random_state** (*int or numpy.random.RandomState or None, optional (default=None)*) – Pseudo-random number generator to control the random sub-sampling. See scikit-learn glossary.

**fit**(*X*, *y=None*)

Fit data X by computing the binning thresholds.

**Parameters** **X** (*array-like*) – The data to bin

**Returns** self

**Return type** object

**transform**(*X*)

Bin data X.

**Parameters** **X** (*array-like*) – The data to bin

**Returns** **X_binned** – The binned data

**Return type** array-like

# CHAPTER 4

# Grower

This module contains the TreeGrower class.

TreeGrowee builds a regression tree fitting a Newton-Raphson step, based on the gradients and hessians of the training data.

**class** pygbm.grower.**TreeGrower**(*X_binned*, *gradients*, *hessians*, *max_leaf_nodes=None*, *max_depth=None*, *min_samples_leaf=20*, *min_gain_to_split=0.0*, *max_bins=256*, *n_bins_per_feature=None*, *l2_regularization=0.0*, *min_hessian_to_split=0.001*, *shrinkage=1.0*)

Tree grower class used to build a tree.

The tree is fitted to predict the values of a Newton-Raphson step. The splits are considered in a best-first fashion, and the quality of a split is defined in splitting._split_gain.

**Parameters**

- **X_binned** (*array-like of int, shape=(n_samples, n_features)*) – The binned input samples. Must be Fortran-aligned.
- **gradients** (*array-like, shape=(n_samples,)*) – The gradients of each training sample. Those are the gradients of the loss w.r.t the predictions, evaluated at iteration `i - 1`.
- **hessians** (*array-like, shape=(n_samples,)*) – The hessians of each training sample. Those are the hessians of the loss w.r.t the predictions, evaluated at iteration `i - 1`.
- **max_leaf_nodes** (*int or None, optional(default=None)*) – The maximum number of leaves for each tree. If None, there is no maximum limit.
- **max_depth** (*int or None, optional(default=None)*) – The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf.
- **min_samples_leaf** (*int, optional(default=20)*) – The minimum number of samples per leaf.

- **min_gain_to_split** (*float, optional(default=0.)*) – The minimum gain needed to split a node. Splits with lower gain will be ignored.
- **max_bins** (*int, optional(default=256)*) – The maximum number of bins. Used to define the shape of the histograms.
- **n_bins_per_feature** (*array-like of int or int, optional(default=None)*) – The actual number of bins needed for each feature, which is lower or equal to `max_bins`. If it's an int, all features are considered to have the same number of bins. If None, all features are considered to have `max_bins` bins.
- **l2_regularization** (*float, optional(default=0)*) – The L2 regularization parameter.
- **min_hessian_to_split** (*float, optional(default=1e-3)*) – The minimum sum of hessians needed in each node. Splits that result in at least one child having a sum of hessians less than min_hessian_to_split are discarded.
- **shrinkage** (*float, optional(default=1)*) – The shrinkage parameter to apply to the leaves values, also known as learning rate.

**can_split_further**()

Return True if there are still nodes to split.

**grow**()

Grow the tree, from root to leaves.

**make_predictor**(*bin_thresholds=None*)

Make a TreePredictor object out of the current tree.

**Parameters** **bin_thresholds** (*array-like of floats, optional (default=None)*) – The actual thresholds values of each bin.

**Returns**

**Return type** A TreePredictor object.

**split_next**()

Split the node with highest potential gain.

**Returns**

- **left** (*TreeNode*) – The resulting left child.
- **right** (*TreeNode*) – The resulting right child.

**class** pygbm.grower.**TreeNode**(*depth*, *sample_indices*, *sum_gradients*, *sum_hessians*, *parent=None*)

Tree Node class used in TreeGrower.

This isn't used for prediction purposes, only for training (see TreePredictor).

**Parameters**

- **depth** (*int*) – The depth of the node, i.e. its distance from the root
- **samples_indices** (*array of int*) – The indices of the samples at the node
- **sum_gradients** (*float*) – The sum of the gradients of the samples at the node
- **sum_hessians** (*float*) – The sum of the hessians of the samples at the node
- **parent** (*TreeNode or None, optional(default=None)*) – The parent of the node. None for root.

**depth**

*int* – The depth of the node, i.e. its distance from the root

**`samples_indices`**
*array of int* – The indices of the samples at the node

**`sum_gradients`**
*float* – The sum of the gradients of the samples at the node

**`sum_hessians`**
*float* – The sum of the hessians of the samples at the node

**`parent`**
*TreeNode or None, optional(default=None)* – The parent of the node. None for root.

**`split_info`**
*SplitInfo or None* – The result of the split evaluation

**`left_child`**
*TreeNode or None* – The left child of the node. None for leaves.

**`right_child`**
*TreeNode or None* – The right child of the node. None for leaves.

**`value`**
*float or None* – The value of the leaf, as computed in finalize_leaf(). None for non-leaf nodes

**`find_split_time`**
*float* – The total time spent computing the histogram and finding the best split at the node.

**`construction_speed`**
*float* – The Number of samples at the node divided find_split_time.

**`apply_split_time`**
*float* – The total time spent actually splitting the node, e.g. splitting samples_indices into left and right child.

**`hist_subtraction`**
*bool* – Wheter the subtraction method was used for computing the histograms.

# CHAPTER 5

# Splitting

This module contains njitted routines and data structures to:

- Find the best possible split of a node. For a given node, a split is characterized by a feature and a bin.
- Apply a split to a node, i.e. split the indices of the samples at the node into the newly created left and right childs.

`pygbm.splitting.`**`find_node_split`**

For each feature, find the best bin to split on at a given node.

Returns the best split info among all features, and the histograms of all the features. The histograms are computed by scanning the whole data.

**Parameters**

- **`context`** (*`SplittingContext`*) – The splitting context
- **`sample_indices`** (*`array of int`*) – The indices of the samples at the node to split.

**Returns**

- **best_split_info** (*SplitInfo*) – The info about the best possible split among all features.
- **histograms** (*array of HISTOGRAM_DTYPE, shape=(n_features, max_bins)*) – The histograms of each feature. A histogram is an array of HISTOGRAM_DTYPE of size `max_bins` (only `n_bins_per_features[feature]` entries are relevant).

`pygbm.splitting.`**`find_node_split_subtraction`**

For each feature, find the best bin to split on at a given node.

Returns the best split info among all features, and the histograms of all the features.

This does the same job as `find_node_split()` but uses the histograms of the parent and sibling of the node to split. This allows to use the identity: `histogram(parent) = histogram(node) - histogram(sibling)`, which is significantly faster than computing the histograms from data.

Returns the best SplitInfo among all features, along with all the feature histograms that can be latter used to compute the sibling or children histograms by substraction.

**Parameters**

- **`context`** (*`SplittingContext`*) – The splitting context
- **`sample_indices`** (*`array of int`*) – The indices of the samples at the node to split.
- **`parent_histograms`** (*`array of HISTOGRAM_DTYPE of shape(n_features, max_bins)`*) – The histograms of the parent
- **`sibling_histograms`** (*`array of HISTOGRAM_DTYPE of shape(n_features, max_bins)`*) – The histograms of the sibling

**Returns**

- **best_split_info** (*SplitInfo*) – The info about the best possible split among all features.
- **histograms** (*array of HISTOGRAM_DTYPE, shape=(n_features, max_bins)*) – The histograms of each feature. A histogram is an array of HISTOGRAM_DTYPE of size `max_bins` (only `n_bins_per_features[feature]` entries are relevant).

`pygbm.splitting.`**`split_indices`**

Split samples into left and right arrays.

**Parameters**

- **`context`** (*`SplittingContext`*) – The splitting context
- **`split_ingo`** (*`SplitInfo`*) – The SplitInfo of the node to split
- **`sample_indices`** (*`array of int`*) – The indices of the samples at the node to split. This is a view on context.partition, and it is modified inplace by placing the indices of the left child at the beginning, and the indices of the right child at the end.

**Returns**

- **left_indices** (*array of int*) – The indices of the samples in the left child. This is a view on context.partition.
- **right_indices** (*array of int*) – The indices of the samples in the right child. This is a view on context.partition.

# CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index