Package ‘xgboost’

March 18, 2026

Type Package

Title Extreme Gradient Boosting

Version 3.2.1.1

Date 2026-03-18

Maintainer Jiaming Yuan <`jm.yuan@outlook.com`>

Description Extreme Gradient Boosting, which is an efficient implementation
of the gradient boosting frame-
work from Chen & Guestrin (2016) <doi:10.1145/2939672.2939785>.
This package is its R interface. The package includes efficient linear
model solver and tree learning algorithms. The package can automatically
do parallel computation on a single machine which could be more than 10
times faster than existing gradient boosting packages. It supports
various objective functions, including regression, classification and ranking.
The package is made to be extensible, so that users are also allowed to define
their own objectives easily.

License Apache License (== 2.0) | file LICENSE

URL `https://github.com/dmlc/xgboost`

BugReports `https://github.com/dmlc/xgboost/issues`

NeedsCompilation yes

VignetteBuilder knitr

Suggests knitr, rmarkdown, ggplot2 (>= 1.0.1), DiagrammeR (>= 0.9.0),
DiagrammeRsvg, rsvg, htmlwidgets, Ckmeans.1d.dp (>= 3.3.1), vcd
(>= 1.3), testthat, igraph (>= 1.0.1), float, titanic,
RhpcBLASctl, survival

Depends R (>= 4.3.0)

Imports Matrix (>= 1.1-0), methods, data.table (>= 1.9.6), jsonlite
(>= 1.0)

RoxygenNote 7.3.3

Encoding UTF-8

SystemRequirements GNU make, C++17

Author Tianqi Chen [aut],
Tong He [aut],
Michael Benesty [aut],
Vadim Khotilovich [aut],
Yuan Tang [aut] (ORCID: <https://orcid.org/0000-0001-5243-233X>),
Hyunsu Cho [aut],
Kailong Chen [aut],
Rory Mitchell [aut],
Ignacio Cano [aut],
Tianyi Zhou [aut],
Mu Li [aut],
Junyuan Xie [aut],
Min Lin [aut],
Yifeng Geng [aut],
Yutian Li [aut],
Jiaming Yuan [aut, cre],
David Cortes [aut],
XGBoost contributors [cph] (base XGBoost implementation)

Repository CRAN

Date/Publication 2026-03-18 06:10:03 UTC

Contents

`a-compatibility-note-for-saveRDS-save`

Model Serialization and Compatibility

Description

When it comes to serializing XGBoost models, it's possible to use R serializers such as `save()` or `saveRDS()` to serialize an XGBoost model object, but XGBoost also provides its own serializers with better compatibility guarantees, which allow loading said models in other language bindings of XGBoost.

Note that an `xgb.Booster` object (**as produced by** `xgb.train()`, see rest of the doc for objects produced by `xgboost()`), outside of its core components, might also keep:

- Additional model configuration (accessible through `xgb.config()`), which includes model fitting parameters like `max_depth` and runtime parameters like `nthread`. These are not necessarily useful for prediction/importance/plotting.
- Additional R specific attributes - e.g. results of callbacks, such as evaluation logs, which are kept as a `data.table` object, accessible through `attributes(model)$evaluation_log` if present.

The first one (configurations) does not have the same compatibility guarantees as the model itself, including attributes that are set and accessed through `xgb.attributes()` - that is, such configuration might be lost after loading the booster in a different XGBoost version, regardless of the serializer that was used. These are saved when using `saveRDS()`, but will be discarded if loaded into an incompatible XGBoost version. They are not saved when using XGBoost's serializers from its public interface including `xgb.save()` and `xgb.save.raw()`.

The second ones (R attributes) are not part of the standard XGBoost model structure, and thus are not saved when using XGBoost's own serializers. These attributes are only used for informational purposes, such as keeping track of evaluation metrics as the model was fit, or saving the R call that produced the model, but are otherwise not used for prediction / importance / plotting / etc. These R attributes are only preserved when using R's serializers.

In addition to the regular `xgb.Booster` objects produced by `xgb.train()`, the function `xgboost()` produces objects with a different subclass `xgboost` (which inherits from `xgb.Booster`), which keeps other additional metadata as R attributes such as class names in classification problems, and which has a dedicated `predict` method that uses different defaults and takes different argument names. XGBoost's own serializers can work with this `xgboost` class, but as they do not keep R attributes, the resulting object, when deserialized, is downcasted to the regular `xgb.Booster` class (i.e. it loses the metadata, and the resulting object will use `predict.xgb.Booster()` instead of `predict.xgboost()`) - for these `xgboost` objects, `saveRDS` might thus be a better option if the extra functionalities are needed.

Note that XGBoost models in R starting from version `2.1.0` and onwards, and XGBoost models before version `2.1.0`; have a very different R object structure and are incompatible with each other. Hence, models that were saved with R serializers like `saveRDS()` or `save()` before version `2.1.0` will not work with latter `xgboost` versions and vice versa. Be aware that the structure of R model objects could in theory change again in the future, so XGBoost's serializers should be preferred for long-term storage.

Furthermore, note that model objects from XGBoost might not be serializable with third-party R packages like `qs` or `qs2`.

Details

Use `xgb.save()` to save the XGBoost model as a stand-alone file. You may opt into the JSON format by specifying the JSON extension. To read the model back, use `xgb.load()`.

Use `xgb.save.raw()` to save the XGBoost model as a sequence (vector) of raw bytes in a future-proof manner. Future releases of XGBoost will be able to read the raw bytes and re-construct the corresponding model. To read the model back, use `xgb.load.raw()`. The `xgb.save.raw()` function is useful if you would like to persist the XGBoost model as part of another R object.

Use `saveRDS()` if you require the R-specific attributes that a booster might have, such as evaluation logs or the model class `xgboost` instead of `xgb.Booster`, but note that future compatibility of such objects is outside XGBoost's control as it relies on R's serialization format (see e.g. the details section in serialize and `save()` from base R).

For more details and explanation about model persistence and archival, consult the page `https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html`.

Examples

```
data(agaricus.train, package = "xgboost")

bst <- xgb.train(
  data = xgb.DMatrix(agaricus.train$data, label = agaricus.train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = 2,
    objective = "binary:logistic"
  )
)

# Save as a stand-alone file; load it with xgb.load()
fname <- file.path(tempdir(), "xgb_model.ubj")
xgb.save(bst, fname)
bst2 <- xgb.load(fname)

# Save as a stand-alone file (JSON); load it with xgb.load()
fname <- file.path(tempdir(), "xgb_model.json")
xgb.save(bst, fname)
bst2 <- xgb.load(fname)

# Save as a raw byte vector; load it with xgb.load.raw()
xgb_bytes <- xgb.save.raw(bst)
bst2 <- xgb.load.raw(xgb_bytes)

# Persist XGBoost model as part of another R object
obj <- list(xgb_model_bytes = xgb.save.raw(bst), description = "My first XGBoost model")
# Persist the R object. Here, saveRDS() is okay, since it doesn't persist
# xgb.Booster directly. What's being persisted is the future-proof byte representation
# as given by xgb.save.raw().
fname <- file.path(tempdir(), "my_object.Rds")
saveRDS(obj, fname)
# Read back the R object
obj2 <- readRDS(fname)
# Re-construct xgb.Booster object from the bytes
bst2 <- xgb.load.raw(obj2$xgb_model_bytes)
```

`agaricus.test` *Test part from Mushroom Data Set*

Description

This data set is originally from the Mushroom data set, UCI Machine Learning Repository.

Usage

```
data(agaricus.test)
```

Format

A list containing a label vector, and a dgCMatrix object with 1611 rows and 126 variables

Details

It includes the following fields:

- `label`: The label for each record.
- `data`: A sparse Matrix of 'dgCMatrix' class with 126 columns.

References

https://archive.ics.uci.edu/ml/datasets/Mushroom

Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository http://archive.ics.uci.edu/ml. Irvine, CA: University of California, School of Information and Computer Science.

`agaricus.train` *Training part from Mushroom Data Set*

Description

This data set is originally from the Mushroom data set, UCI Machine Learning Repository.

Usage

```
data(agaricus.train)
```

Format

A list containing a label vector, and a dgCMatrix object with 6513 rows and 127 variables

Details

It includes the following fields:

- `label`: The label for each record.
- `data`: A sparse Matrix of 'dgCMatrix' class with 126 columns.

References

`https://archive.ics.uci.edu/ml/datasets/Mushroom`

Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository `http://archive.ics.uci.edu/ml`. Irvine, CA: University of California, School of Information and Computer Science.

`coef.xgb.Booster` *Extract coefficients from linear booster*

Description

Extracts the coefficients from a 'gblinear' booster object, as produced by `xgb.train()` when using parameter `booster="gblinear"`.

Note: this function will error out if passing a booster model which is not of "gblinear" type.

Usage

```
## S3 method for class 'xgb.Booster'
coef(object, ...)
```

Arguments

`object`	A fitted booster of 'gblinear' type.
`...`	Not used.

Value

The extracted coefficients:

- If there is only one coefficient per column in the data, will be returned as a vector, potentially containing the feature names if available, with the intercept as first column.
- If there is more than one coefficient per column in the data (e.g. when using `objective="multi:softmax"`), will be returned as a matrix with dimensions equal to `[num_features, num_cols]`, with the intercepts as first row. Note that the column (classes in multi-class classification) dimension will not be named.

The intercept returned here will include the 'base_score' parameter (unlike the 'bias' or the last coefficient in the model dump, which doesn't have 'base_score' added to it), hence one should get the same values from calling `predict(..., outputmargin = TRUE)` and from performing a matrix multiplication with `model.matrix(~., ...)`.

Be aware that the coefficients are obtained by first converting them to strings and back, so there will always be some very small lose of precision compared to the actual coefficients as used by predict.xgb.Booster.

Examples

```
library(xgboost)

data(mtcars)

y <- mtcars[, 1]
x <- as.matrix(mtcars[, -1])

dm <- xgb.DMatrix(data = x, label = y, nthread = 1)
params <- xgb.params(booster = "gblinear", nthread = 1)
model <- xgb.train(data = dm, params = params, nrounds = 2)
coef(model)
```

dim.xgb.DMatrix *Dimensions of xgb.DMatrix*

Description

Returns a vector of numbers of rows and of columns in an `xgb.DMatrix`.

Usage

```
## S3 method for class 'xgb.DMatrix'
dim(x)
```

Arguments

`x`	Object of class `xgb.DMatrix`

Details

Note: since `nrow()` and `ncol()` internally use `dim()`, they can also be directly used with an `xgb.DMatrix` object.

Examples

```
data(agaricus.train, package = "xgboost")

train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label = train$label, nthread = 2)

stopifnot(nrow(dtrain) == nrow(train$data))
stopifnot(ncol(dtrain) == ncol(train$data))
stopifnot(all(dim(dtrain) == dim(train$data)))
```

dimnames.xgb.DMatrix *Handling of column names of* xgb.DMatrix

Description

Only column names are supported for xgb.DMatrix, thus setting of row names would have no effect and returned row names would be NULL.

Usage

```
## S3 method for class 'xgb.DMatrix'
dimnames(x)

## S3 replacement method for class 'xgb.DMatrix'
dimnames(x) <- value
```

Arguments

x	Object of class xgb.DMatrix.
value	A list of two elements: the first one is ignored and the second one is column names

Details

Generic dimnames() methods are used by colnames(). Since row names are irrelevant, it is recommended to use colnames() directly.

Examples

```
data(agaricus.train, package = "xgboost")

train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label = train$label, nthread = 2)
dimnames(dtrain)
colnames(dtrain)
colnames(dtrain) <- make.names(1:ncol(train$data))
print(dtrain, verbose = TRUE)
```

getinfo.xgb.Booster *Get or set information of xgb.DMatrix and xgb.Booster objects*

Description

Get or set information of xgb.DMatrix and xgb.Booster objects

Usage

```
## S3 method for class 'xgb.Booster'
getinfo(object, name)

## S3 method for class 'xgb.Booster'
setinfo(object, name, info)

getinfo(object, name)

## S3 method for class 'xgb.DMatrix'
getinfo(object, name)

setinfo(object, name, info)

## S3 method for class 'xgb.DMatrix'
setinfo(object, name, info)
```

Arguments

`object`	Object of class `xgb.DMatrix` or `xgb.Booster`.
`name`	The name of the information field to get (see details).
`info`	The specific field of information to set.

Details

The `name` field can be one of the following for `xgb.DMatrix`:

- label
- weight
- base_margin
- label_lower_bound
- label_upper_bound
- group
- feature_type
- feature_name
- nrow

See the documentation for `xgb.DMatrix()` for more information about these fields.

For `xgb.Booster`, can be one of the following:

- `feature_type`
- `feature_name`

Note that, while 'qid' cannot be retrieved, it is possible to get the equivalent 'group' for a DMatrix that had 'qid' assigned.

Important: when calling `setinfo()`, the objects are modified in-place. See `xgb.copy.Booster()` for an idea of this in-place assignment works.

See the documentation for `xgb.DMatrix()` for possible fields that can be set (which correspond to arguments in that function).

Note that the following fields are allowed in the construction of an `xgb.DMatrix` but **are not** allowed here:

- data
- missing
- silent
- nthread

Value

For `getinfo()`, will return the requested field. For `setinfo()`, will always return value `TRUE` if it succeeds.

Examples

```
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))

labels <- getinfo(dtrain, "label")
setinfo(dtrain, "label", 1 - labels)

labels2 <- getinfo(dtrain, "label")
stopifnot(all(labels2 == 1 - labels))
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))

labels <- getinfo(dtrain, "label")
setinfo(dtrain, "label", 1 - labels)

labels2 <- getinfo(dtrain, "label")
stopifnot(all.equal(labels2, 1 - labels))
```

predict.xgb.Booster *Predict method for XGBoost model*

Description

Predict values on data based on XGBoost model.

Usage

```
## S3 method for class 'xgb.Booster'
predict(
  object,
  newdata,
  missing = NA,
  outputmargin = FALSE,
  predleaf = FALSE,
  predcontrib = FALSE,
  approxcontrib = FALSE,
  predinteraction = FALSE,
  training = FALSE,
  iterationrange = NULL,
  strict_shape = FALSE,
  avoid_transpose = FALSE,
  validate_features = FALSE,
  base_margin = NULL,
  ...
)
```

Arguments

`object` Object of class `xgb.Booster`.

`newdata` Takes `data.frame`, `matrix`, `dgCMatrix`, `dgRMatrix`, `dsparseVector`, local data file, or `xgb.DMatrix`.

For single-row predictions on sparse data, it is recommended to use CSR format. If passing a sparse vector, it will take it as a row vector.

Note that, for repeated predictions on the same data, one might want to create a DMatrix to pass here instead of passing R types like matrices or data frames, as predictions will be faster on DMatrix.

If `newdata` is a `data.frame`, be aware that:

- Columns will be converted to numeric if they aren't already, which could potentially make the operation slower than in an equivalent `matrix` object.
- The order of the columns must match with that of the data from which the model was fitted (i.e. columns will not be referenced by their names, just by their order in the data), unless passing `validate_features = TRUE` (which is not the default).

- If the model was fitted to data with categorical columns, these columns must be of `factor` type here, and must use the same encoding (i.e. have the same levels).
- If `newdata` contains any `factor` columns, they will be converted to base-0 encoding (same as during DMatrix creation) - hence, one should not pass a `factor` under a column which during training had a different type.
- Any columns with type other than `factor` will be interpreted as numeric.

`missing`	Float value that represents missing values in data (e.g., 0 or some other extreme value). This parameter is not used when `newdata` is an `xgb.DMatrix` - in such cases, should pass this as an argument to the DMatrix constructor instead.
`outputmargin`	Whether the prediction should be returned in the form of original untransformed sum of predictions from boosting iterations' results. E.g., setting `outputmargin = TRUE` for logistic regression would return log-odds instead of probabilities.
`predleaf`	Whether to predict per-tree leaf indices.
`predcontrib`	Whether to return feature contributions to individual predictions (see Details).
`approxcontrib`	Whether to use a fast approximation for feature contributions (see Details).
`predinteraction`	Whether to return contributions of feature interactions to individual predictions (see Details).
`training`	Whether the prediction result is used for training. For dart booster, training predicting will perform dropout.
`iterationrange`	Sequence of rounds/iterations from the model to use for prediction, specified by passing a two-dimensional vector with the start and end numbers in the sequence (same format as R's `seq` - i.e. base-1 indexing, and inclusive of both ends). For example, passing `c(1,20)` will predict using the first twenty iterations, while passing `c(1,1)` will predict using only the first one. If passing `NULL`, will either stop at the best iteration if the model used early stopping, or use all of the iterations (rounds) otherwise. If passing "all", will use all of the rounds regardless of whether the model had early stopping or not. Not applicable to `gblinear` booster.
`strict_shape`	Whether to always return an array with the same dimensions for the given prediction mode regardless of the model type - meaning that, for example, both a multi-class and a binary classification model would generate output arrays with the same number of dimensions, with the 'class' dimension having size equal to '1' for the binary model. If passing `FALSE` (the default), dimensions will be simplified according to the model type, so that a binary classification model for example would not have a redundant dimension for 'class'. See documentation for the return type for the exact shape of the output arrays for each prediction mode.
`avoid_transpose`	Whether to output the resulting predictions in the same memory layout in which they are generated by the core XGBoost library, without transposing them to match the expected output shape.

Internally, XGBoost uses row-major order for the predictions it generates, while R arrays use column-major order, hence the result needs to be transposed in order to have the expected shape when represented as an R array or matrix, which might be a slow operation.

If passing `TRUE`, then the result will have dimensions in reverse order - for example, rows will be the last dimensions instead of the first dimension.

`validate_features`

When `TRUE`, validate that the Booster's and newdata's feature_names match (only applicable when both `object` and `newdata` have feature names).

If the column names differ and `newdata` is not an `xgb.DMatrix`, will try to reorder the columns in `newdata` to match with the booster's.

If the booster has feature types and `newdata` is either an `xgb.DMatrix` or `data.frame`, will additionally verify that categorical columns are of the correct type in `newdata`, throwing an error if they do not match.

If passing `FALSE`, it is assumed that the feature names and types are the same, and come in the same order as in the training data.

Note that this check might add some sizable latency to the predictions, so it's recommended to disable it for performance-sensitive applications.

`base_margin` Base margin used for boosting from existing model (raw score that gets added to all observations independently of the trees in the model).

If supplied, should be either a vector with length equal to the number of rows in `newdata` (for objectives which produces a single score per observation), or a matrix with number of rows matching to the number rows in `newdata` and number of columns matching to the number of scores estimated by the model (e.g. number of classes for multi-class classification).

Note that, if `newdata` is an `xgb.DMatrix` object, this argument will be ignored as it needs to be added to the DMatrix instead (e.g. by passing it as an argument in its constructor, or by calling `setinfo.xgb.DMatrix()`.

`...` Not used.

Details

Note that `iterationrange` would currently do nothing for predictions from "gblinear", since "gblinear" doesn't keep its boosting history.

One possible practical applications of the `predleaf` option is to use the model as a generator of new features which capture non-linearity and interactions, e.g., as implemented in `xgb.create.features()`.

Setting `predcontrib = TRUE` allows to calculate contributions of each feature to individual predictions. For "gblinear" booster, feature contributions are simply linear terms (feature_beta * feature_value). For "gbtree" booster, feature contributions are SHAP values (Lundberg 2017) that sum to the difference between the expected output of the model and the current prediction (where the hessian weights are used to compute the expectations). Setting `approxcontrib = TRUE` approximates these values following the idea explained in `http://blog.datadive.net/interpreting-random-forests/`.

With `predinteraction = TRUE`, SHAP values of contributions of interaction of each pair of features are computed. Note that this operation might be rather expensive in terms of compute and memory. Since it quadratically depends on the number of features, it is recommended to perform selection of the most important features first. See below about the format of the returned results.

The `predict()` method uses as many threads as defined in `xgb.Booster` object (all by default). If you want to change their number, assign a new number to `nthread` using `xgb.model.parameters<-()`. Note that converting a matrix to `xgb.DMatrix()` uses multiple threads too.

Value

A numeric vector or array, with corresponding dimensions depending on the prediction mode and on parameter `strict_shape` as follows:

If passing `strict_shape=FALSE`:

- For regression or binary classification: a vector of length `nrows`.
- For multi-class and multi-target objectives: a matrix of dimensions `[nrows, ngroups]`.
  Note that objective variant `multi:softmax` defaults towards predicting most likely class (a vector `nrows`) instead of per-class probabilities.
- For `predleaf`: a matrix with one column per tree.
  For multi-class / multi-target, they will be arranged so that columns in the output will have the leafs from one group followed by leafs of the other group (e.g. order will be `group1:feat1`, `group1:feat2`, ..., `group2:feat1`, `group2:feat2`, ...).
  If there is more than one parallel tree (e.g. random forests), the parallel trees will be the last grouping in the resulting order, which will still be 2D.
- For `predcontrib`: when not multi-class / multi-target, a matrix with dimensions `[nrows, nfeats+1]`. The last "+ 1" column corresponds to the baseline value.
  For multi-class and multi-target objectives, will be an array with dimensions `[nrows, ngroups, nfeats+1]`.
  The contribution values are on the scale of untransformed margin (e.g., for binary classification, the values are log-odds deviations from the baseline).
- For `predinteraction`: when not multi-class / multi-target, the output is a 3D array of dimensions `[nrows, nfeats+1, nfeats+1]`. The off-diagonal (in the last two dimensions) elements represent different feature interaction contributions. The array is symmetric w.r.t. the last two dimensions. The "+ 1" columns corresponds to the baselines. Summing this array along the last dimension should produce practically the same result as `predcontrib = TRUE`.
  For multi-class and multi-target, will be a 4D array with dimensions `[nrows, ngroups, nfeats+1, nfeats+1]`

If passing `strict_shape=TRUE`, the result is always a matrix (if 2D) or array (if 3D or higher):

- For normal predictions, the dimension is `[nrows, ngroups]`.
- For `predcontrib=TRUE`, the dimension is `[nrows, ngroups, nfeats+1]`.
- For `predinteraction=TRUE`, the dimension is `[nrows, ngroups, nfeats+1, nfeats+1]`.
- For `predleaf=TRUE`, the dimension is `[nrows, niter, ngroups, num_parallel_tree]`.

If passing `avoid_transpose=TRUE`, then the dimensions in all cases will be in reverse order - for example, for `predinteraction`, they will be `[nfeats+1, nfeats+1, ngroups, nrows]` instead of `[nrows, ngroups, nfeats+1, nfeats+1]`.

References

1. Scott M. Lundberg, Su-In Lee, "A Unified Approach to Interpreting Model Predictions", NIPS Proceedings 2017, https://arxiv.org/abs/1705.07874
2. Scott M. Lundberg, Su-In Lee, "Consistent feature attribution for tree ensembles", https://arxiv.org/abs/1706.06060

See Also

xgb.train()

Examples

```
## binary classification:

data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 2 for examples
nthread <- 2
data.table::setDTthreads(nthread)

train <- agaricus.train
test <- agaricus.test

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 5,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

# use all trees by default
pred <- predict(bst, test$data)
# use only the 1st tree
pred1 <- predict(bst, test$data, iterationrange = c(1, 1))

# Predicting tree leafs:
# the result is an nsamples X ntrees matrix
pred_leaf <- predict(bst, test$data, predleaf = TRUE)
str(pred_leaf)

# Predicting feature contributions to predictions:
# the result is an nsamples X (nfeatures + 1) matrix
pred_contr <- predict(bst, test$data, predcontrib = TRUE)
str(pred_contr)
# verify that contributions' sums are equal to log-odds of predictions (up to float precision):
summary(rowSums(pred_contr) - qlogis(pred))
# for the 1st record, let's inspect its features that had non-zero contribution to prediction:
contr1 <- pred_contr[1,]
contr1 <- contr1[-length(contr1)]    # drop intercept
contr1 <- contr1[contr1 != 0]        # drop non-contributing features
contr1 <- contr1[order(abs(contr1))] # order by contribution magnitude
old_mar <- par("mar")
par(mar = old_mar + c(0,7,0,0))
barplot(contr1, horiz = TRUE, las = 2, xlab = "contribution to prediction in log-odds")
par(mar = old_mar)
```

```
## multiclass classification in iris dataset:

lb <- as.numeric(iris$Species) - 1
num_class <- 3

set.seed(11)

bst <- xgb.train(
  data = xgb.DMatrix(as.matrix(iris[, -5], nthread = 1), label = lb),
  nrounds = 10,
  params = xgb.params(
    max_depth = 4,
    nthread = 2,
    subsample = 0.5,
    objective = "multi:softprob",
    num_class = num_class
  )
)

# predict for softmax returns num_class probability numbers per case:
pred <- predict(bst, as.matrix(iris[, -5]))
str(pred)
# convert the probabilities to softmax labels
pred_labels <- max.col(pred) - 1
# the following should result in the same error as seen in the last iteration
sum(pred_labels != lb) / length(lb)

# compare with predictions from softmax:
set.seed(11)

bst <- xgb.train(
  data = xgb.DMatrix(as.matrix(iris[, -5], nthread = 1), label = lb),
  nrounds = 10,
  params = xgb.params(
    max_depth = 4,
    nthread = 2,
    subsample = 0.5,
    objective = "multi:softmax",
    num_class = num_class
  )
)

pred <- predict(bst, as.matrix(iris[, -5]))
str(pred)
all.equal(pred, pred_labels)
# prediction from using only 5 iterations should result
# in the same error as seen in iteration 5:
pred5 <- predict(bst, as.matrix(iris[, -5]), iterationrange = c(1, 5))
sum(pred5 != lb) / length(lb)
```

`predict.xgboost` *Compute predictions from XGBoost model on new data*

Description

Predict values on data based on XGBoost model.

Usage

```
## S3 method for class 'xgboost'
predict(
  object,
  newdata,
  type = "response",
  base_margin = NULL,
  iteration_range = NULL,
  validate_features = TRUE,
  ...
)
```

Arguments

`object`	An XGBoost model object of class `xgboost`, as produced by function `xgboost()`. Note that there is also a lower-level `predict.xgb.Booster()` method for models of class `xgb.Booster` as produced by `xgb.train()`, which can also be used for `xgboost` class models as an alternative that performs fewer validations and post-processings.
`newdata`	Data on which to compute predictions from the model passed in `object`. Supported input classes are: • Data Frames (class `data.frame` from base R and subclasses like `data.table`). • Matrices (class `matrix` from base R). • Sparse matrices from package `Matrix`, either as class `dgRMatrix` (CSR) or `dgCMatrix` (CSC). • Sparse vectors from package `Matrix`, which will be interpreted as containing a single observation. In the case of data frames, if there are any categorical features, they should be of class `factor` and should have the same levels as the `factor` columns of the data from which the model was constructed. Any columns with type other than `factor` will be interpreted as numeric. If there are named columns and the model was fitted to data with named columns, they will be matched by name by default (see `validate_features`).
`type`	Type of prediction to make. Supported options are: • `"response"`: will output model predictions on the scale of the response variable (e.g. probabilities of belonging to the last class in the case of binary classification). Result will be either a numeric vector with length matching

to rows in `newdata`, or a numeric matrix with shape `[nrows(newdata), nscores]` (for objectives that produce more than one score per observation such as multi-class classification or multi-quantile regression).

- `"raw"`: will output the unprocessed boosting scores (e.g. log-odds in the case of objective `binary:logistic`). Same output shape and type as for `"response"`.
- `"class"`: will output the class with the highest predicted probability, returned as a `factor` (only applicable to classification objectives) with length matching to rows in `newdata`.
- `"leaf"`: will output the terminal node indices of each observation across each tree, as an integer matrix of shape `[nrows(newdata), ntrees]`, or as an integer array with an extra one or two dimensions, up to `[nrows(newdata), ntrees, nscores, r` for models that produce more than one score per tree and/or which have more than one parallel tree (e.g. random forests).
  Only applicable to tree-based boosters (not `gblinear`).
- `"contrib"`: will produce per-feature contribution estimates towards the model score for a given observation, based on SHAP values. The contribution values are on the scale of untransformed margin (e.g., for binary classification, the values are log-odds deviations from the baseline).
  Output will be a numeric matrix with shape `[nrows, nfeatures+1]`, with the intercept being the last feature, or a numeric array with shape `[nrows, nscores, nfeatures+1]` if the model produces more than one score per observation.
- `"interaction"`: similar to `"contrib"`, but computing SHAP values of contributions of interaction of each pair of features. Note that this operation might be rather expensive in terms of compute and memory.
  Since it quadratically depends on the number of features, it is recommended to perform selection of the most important features first.
  Output will be a numeric array of shape `[nrows, nfeatures+1, nfeatures+1]`, or shape `[nrows, nscores, nfeatures+1, nfeatures+1]` (for objectives that produce more than one score per observation).

`base_margin` Base margin used for boosting from existing model (raw score that gets added to all observations independently of the trees in the model).

If supplied, should be either a vector with length equal to the number of rows in `newdata` (for objectives which produces a single score per observation), or a matrix with number of rows matching to the number rows in `newdata` and number of columns matching to the number of scores estimated by the model (e.g. number of classes for multi-class classification).

`iteration_range`

Sequence of rounds/iterations from the model to use for prediction, specified by passing a two-dimensional vector with the start and end numbers in the sequence (same format as R's `seq` - i.e. base-1 indexing, and inclusive of both ends).

For example, passing `c(1,20)` will predict using the first twenty iterations, while passing `c(1,1)` will predict using only the first one.

If passing `NULL`, will either stop at the best iteration if the model used early stopping, or use all of the iterations (rounds) otherwise.

If passing "all", will use all of the rounds regardless of whether the model had early stopping or not.

Not applicable to gblinear booster.

validate_features
: Validate that the feature names in the data match to the feature names in the column, and reorder them in the data otherwise.

  If passing FALSE, it is assumed that the feature names and types are the same, and come in the same order as in the training data.

  Be aware that this only applies to column names and not to factor levels in categorical columns.

  Note that this check might add some sizable latency to the predictions, so it's recommended to disable it for performance-sensitive applications.

... Not used.

Value

Either a numeric vector (for 1D outputs), numeric matrix (for 2D outputs), numeric array (for 3D and higher), or factor (for class predictions). See documentation for parameter type for details about what the output type and shape will be.

Examples

```
data("ToothGrowth")
y <- ToothGrowth$supp
x <- ToothGrowth[, -2L]
model <- xgboost(x, y, nthreads = 1L, nrounds = 3L, max_depth = 2L)
pred_prob <- predict(model, x[1:5, ], type = "response")
pred_raw <- predict(model, x[1:5, ], type = "raw")
pred_class <- predict(model, x[1:5, ], type = "class")

# Relationships between these
manual_probs <- 1 / (1 + exp(-pred_raw))
manual_class <- ifelse(manual_probs < 0.5, levels(y)[1], levels(y)[2])

# They should match up to numerical precision
round(pred_prob, 6) == round(manual_probs, 6)
pred_class == manual_class
```

---

print.xgb.Booster *Print xgb.Booster*

Description

Print information about xgb.Booster.

Usage

```
## S3 method for class 'xgb.Booster'
print(x, ...)
```

Arguments

`x`	An `xgb.Booster` object.
`...`	Not used.

Value

The same `x` object, returned invisibly

Examples

```
data(agaricus.train, package = "xgboost")
train <- agaricus.train

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = 2,
    objective = "binary:logistic"
  )
)

attr(bst, "myattr") <- "memo"

print(bst)
```

`print.xgb.cv.synchronous` *Print xgb.cv result*

Description

Prints formatted results of `xgb.cv()`.

Usage

```
## S3 method for class 'xgb.cv.synchronous'
print(x, verbose = FALSE, ...)
```

Arguments

`x`	An `xgb.cv.synchronous` object.
`verbose`	Whether to print detailed data.
`...`	Passed to `data.table.print()`.

Details

When not verbose, it would only print the evaluation results, including the best iteration (when available).

Examples

```
data(agaricus.train, package = "xgboost")

train <- agaricus.train
cv <- xgb.cv(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nfold = 5,
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = 2,
    objective = "binary:logistic"
  )
)
print(cv)
print(cv, verbose = TRUE)
```

`print.xgb.DMatrix` *Print xgb.DMatrix*

Description

Print information about xgb.DMatrix. Currently it displays dimensions and presence of info-fields and colnames.

Usage

```
## S3 method for class 'xgb.DMatrix'
print(x, verbose = FALSE, ...)
```

Arguments

`x`	An xgb.DMatrix object.
`verbose`	Whether to print colnames (when present).
`...`	Not currently used.

Examples

```
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))
dtrain

print(dtrain, verbose = TRUE)
```

`print.xgboost` *Print info from XGBoost model*

Description

Prints basic properties of an XGBoost model object.

Usage

```
## S3 method for class 'xgboost'
print(x, ...)
```

Arguments

`x`	An XGBoost model object of class `xgboost`, as produced by function `xgboost()`.
`...`	Not used.

Value

Same object `x`, after printing its info.

`variable.names.xgb.Booster` *Get Features Names from Booster*

Description

Returns the feature / variable / column names from a fitted booster object, which are set automatically during the call to `xgb.train()` from the DMatrix names, or which can be set manually through `setinfo()`.

If the object doesn't have feature names, will return `NULL`.

It is equivalent to calling `getinfo(object, "feature_name")`.

Usage

```
## S3 method for class 'xgb.Booster'
variable.names(object, ...)
```

Arguments

`object`	An `xgb.Booster` object.
`...`	Not used.

`xgb.attr` *Accessors for serializable attributes of a model*

Description

These methods allow to manipulate the key-value attribute strings of an XGBoost model.

Usage

```
xgb.attr(object, name)

xgb.attr(object, name) <- value

xgb.attributes(object)

xgb.attributes(object) <- value
```

Arguments

`object`	Object of class `xgb.Booster`. **Will be modified in-place** when assigning to it.
`name`	A non-empty character string specifying which attribute is to be accessed.
`value`	For `xgb.attr<-`, a value of an attribute; for `xgb.attributes<-`, it is a list (or an object coercible to a list) with the names of attributes to set and the elements corresponding to attribute values. Non-character values are converted to character. When an attribute value is not a scalar, only the first index is used. Use `NULL` to remove an attribute.

Details

The primary purpose of XGBoost model attributes is to store some meta data about the model. Note that they are a separate concept from the object attributes in R. Specifically, they refer to key-value strings that can be attached to an XGBoost model, stored together with the model's binary representation, and accessed later (from R or any other interface). In contrast, any R attribute assigned to an R object of `xgb.Booster` class would not be saved by `xgb.save()` because an XGBoost model is an external memory object and its serialization is handled externally. Also, setting an attribute that has the same name as one of XGBoost's parameters wouldn't change the value of that parameter for a model. Use `xgb.model.parameters<-()` to set or change model parameters.

The `xgb.attributes<-` setter either updates the existing or adds one or several attributes, but it doesn't delete the other existing attributes.

Important: since this modifies the booster's C object, semantics for assignment here will differ from R's, as any object reference to the same booster will be modified too, while assignment of R attributes through `attributes(model)$<attr> <- <value>` will follow the usual copy-on-write R semantics (see `xgb.copy.Booster()` for an example of these behaviors).

Value

- `xgb.attr()` returns either a string value of an attribute or `NULL` if an attribute wasn't stored in a model.
- `xgb.attributes()` returns a list of all attributes stored in a model or `NULL` if a model has no stored attributes.

Examples

```
data(agaricus.train, package = "xgboost")
train <- agaricus.train

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = 2,
    objective = "binary:logistic"
  )
)

xgb.attr(bst, "my_attribute") <- "my attribute value"
print(xgb.attr(bst, "my_attribute"))
xgb.attributes(bst) <- list(a = 123, b = "abc")

fname <- file.path(tempdir(), "xgb.ubj")
xgb.save(bst, fname)
bst1 <- xgb.load(fname)
print(xgb.attr(bst1, "my_attribute"))
print(xgb.attributes(bst1))

# deletion:
xgb.attr(bst1, "my_attribute") <- NULL
print(xgb.attributes(bst1))
xgb.attributes(bst1) <- list(a = NULL, b = NULL)
print(xgb.attributes(bst1))
```

`xgb.Callback` *XGBoost Callback Constructor*

Description

Constructor for defining the structure of callback functions that can be executed at different stages of model training (before / after training, before / after each boosting iteration).

Usage

```
xgb.Callback(
  cb_name = "custom_callback",
  env = new.env(),
 f_before_training = function(env, model, data, evals, begin_iteration, end_iteration)
    NULL,
  f_before_iter = function(env, model, data, evals, iteration) NULL,
  f_after_iter = function(env, model, data, evals, iteration, iter_feval) NULL,
  f_after_training = function(env, model, data, evals, iteration, final_feval,
    prev_cb_res) NULL
)
```

Arguments

`cb_name` Name for the callback.

If the callback produces some non-NULL result (from executing the function passed under `f_after_training`), that result will be added as an R attribute to the resulting booster (or as a named element in the result of CV), with the attribute name specified here.

Names of callbacks must be unique - i.e. there cannot be two callbacks with the same name.

`env` An environment object that will be passed to the different functions in the callback. Note that this environment will not be shared with other callbacks.

`f_before_training`

A function that will be executed before the training has started.

If passing `NULL` for this or for the other function inputs, then no function will be executed.

If passing a function, it will be called with parameters supplied as non-named arguments matching the function signatures that are shown in the default value for each function argument.

`f_before_iter` A function that will be executed before each boosting round.

This function can signal whether the training should be finalized or not, by outputting a value that evaluates to `TRUE` - i.e. if the output from the function provided here at a given round is `TRUE`, then training will be stopped before the current iteration happens.

Return values of `NULL` will be interpreted as `FALSE`.

`f_after_iter` A function that will be executed after each boosting round.

This function can signal whether the training should be finalized or not, by outputting a value that evaluates to `TRUE` - i.e. if the output from the function provided here at a given round is `TRUE`, then training will be stopped at that round.

Return values of `NULL` will be interpreted as `FALSE`.

`f_after_training`

A function that will be executed after training is finished.

This function can optionally output something non-NULL, which will become part of the R attributes of the booster (assuming one passes `keep_extra_attributes=TRUE` to `xgb.train()`) under the name supplied for parameter `cb_name` imn the case of `xgb.train()`; or a part of the named elements in the result of `xgb.cv()`.

Details

Arguments that will be passed to the supplied functions are as follows:

- env The same environment that is passed under argument `env`.

  It may be modified by the functions in order to e.g. keep tracking of what happens across iterations or similar.

  This environment is only used by the functions supplied to the callback, and will not be kept after the model fitting function terminates (see parameter `f_after_training`).

- model The booster object when using `xgb.train()`, or the folds when using `xgb.cv()`.

  For `xgb.cv()`, folds are a list with a structure as follows:

  - `dtrain`: The training data for the fold (as an `xgb.DMatrix` object).
  - `bst`: Rhe `xgb.Booster` object for the fold.
  - `evals`: A list containing two DMatrices, with names `train` and `test` (`test` is the held-out data for the fold).
  - `index`: The indices of the hold-out data for that fold (base-1 indexing), from which the `test` entry in `evals` was obtained.

  This object should **not** be in-place modified in ways that conflict with the training (e.g. resetting the parameters for a training update in a way that resets the number of rounds to zero in order to overwrite rounds).

  Note that any R attributes that are assigned to the booster during the callback functions, will not be kept thereafter as the booster object variable is not re-assigned during training. It is however possible to set C-level attributes of the booster through `xgb.attr()` or `xgb.attributes()`, which should remain available for the rest of the iterations and after the training is done.

  For keeping variables across iterations, it's recommended to use `env` instead.

- data The data to which the model is being fit, as an `xgb.DMatrix` object.

  Note that, for `xgb.cv()`, this will be the full data, while data for the specific folds can be found in the `model` object.

- evals The evaluation data, as passed under argument `evals` to `xgb.train()`.

  For `xgb.cv()`, this will always be `NULL`.

- begin_iteration Index of the first boosting iteration that will be executed (base-1 indexing).

  This will typically be '1', but when using training continuation, depending on the parameters for updates, boosting rounds will be continued from where the previous model ended, in which case this will be larger than 1.

- end_iteration Index of the last boostign iteration that will be executed (base-1 indexing, inclusive of this end).

  It should match with argument `nrounds` passed to `xgb.train()` or `xgb.cv()`.

  Note that boosting might be interrupted before reaching this last iteration, for example by using the early stopping callback `xgb.cb.early.stop()`.

- iteration Index of the iteration number that is being executed (first iteration will be the same as parameter `begin_iteration`, then next one will add +1, and so on).

- iter_feval Evaluation metrics for `evals` that were supplied, either determined by the objective, or by parameter `custom_metric`.

For `xgb.train()`, this will be a named vector with one entry per element in `evals`, where the names are determined as 'evals name' + '-' + 'metric name' - for example, if `evals` contains an entry named "tr" and the metric is "rmse", this will be a one-element vector with name "tr-rmse".

For `xgb.cv()`, this will be a 2d matrix with dimensions `[length(evals), nfolds]`, where the row names will follow the same naming logic as the one-dimensional vector that is passed in `xgb.train()`.

Note that, internally, the built-in callbacks such as xgb.cb.print.evaluation summarize this table by calculating the row-wise means and standard deviations.

- final_feval The evaluation results after the last boosting round is executed (same format as `iter_feval`, and will be the exact same input as passed under `iter_feval` to the last round that is executed during model fitting).
- prev_cb_res Result from a previous run of a callback sharing the same name (as given by parameter `cb_name`) when conducting training continuation, if there was any in the booster R attributes.

  Sometimes, one might want to append the new results to the previous one, and this will be done automatically by the built-in callbacks such as xgb.cb.evaluation.log, which will append the new rows to the previous table.

  If no such previous callback result is available (which it never will when fitting a model from start instead of updating an existing model), this will be `NULL`.

  For `xgb.cv()`, which doesn't support training continuation, this will always be `NULL`.

The following names (`cb_name` values) are reserved for internal callbacks:

- print_evaluation
- evaluation_log
- reset_parameters
- early_stop
- save_model
- cv_predict
- gblinear_history

The following names are reserved for other non-callback attributes:

- names
- class
- call
- params
- niter
- nfeatures
- folds

When using the built-in early stopping callback (xgb.cb.early.stop), said callback will always be executed before the others, as it sets some booster C-level attributes that other callbacks might also use. Otherwise, the order of execution will match with the order in which the callbacks are passed to the model fitting function.

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

See Also

Built-in callbacks:

- xgb.cb.print.evaluation
- xgb.cb.evaluation.log
- xgb.cb.reset.parameters
- xgb.cb.early.stop
- xgb.cb.save.model
- xgb.cb.cv.predict
- xgb.cb.gblinear.history

Examples

```
# Example constructing a custom callback that calculates
# squared error on the training data (no separate test set),
# and outputs the per-iteration results.
ssq_callback <- xgb.Callback(
  cb_name = "ssq",
  f_before_training = function(env, model, data, evals,
                               begin_iteration, end_iteration) {
    # A vector to keep track of a number at each iteration
    env$logs <- rep(NA_real_, end_iteration - begin_iteration + 1)
  },
  f_after_iter = function(env, model, data, evals, iteration, iter_feval) {
    # This calculates the sum of squared errors on the training data.
    # Note that this can be better done by passing an 'evals' entry,
    # but this demonstrates a way in which callbacks can be structured.
    pred <- predict(model, data)
    err <- pred - getinfo(data, "label")
    sq_err <- sum(err^2)
    env$logs[iteration] <- sq_err
    cat(
      sprintf(
        "Squared error at iteration %d: %.2f\n",
        iteration, sq_err
      )
    )

    # A return value of 'TRUE' here would signal to finalize the training
    return(FALSE)
  },
  f_after_training = function(env, model, data, evals, iteration,
                              final_feval, prev_cb_res) {
    return(env$logs)
  }
)
```

```
data(mtcars)

y <- mtcars$mpg
x <- as.matrix(mtcars[, -1])

dm <- xgb.DMatrix(x, label = y, nthread = 1)
model <- xgb.train(
  data = dm,
  params = xgb.params(objective = "reg:squarederror", nthread = 1),
  nrounds = 5,
  callbacks = list(ssq_callback)
)

# Result from 'f_after_iter' will be available as an attribute
attributes(model)$ssq
```

`xgb.cb.cv.predict` *Callback for returning cross-validation based predictions*

Description

This callback function saves predictions for all of the test folds, and also allows to save the folds' models.

Usage

```
xgb.cb.cv.predict(save_models = FALSE, outputmargin = FALSE)
```

Arguments

`save_models`	A flag for whether to save the folds' models.
`outputmargin`	Whether to save margin predictions (same effect as passing this parameter to predict.xgb.Booster).

Details

Predictions are saved inside of the `pred` element, which is either a vector or a matrix, depending on the number of prediction outputs per data row. The order of predictions corresponds to the order of rows in the original dataset. Note that when a custom `folds` list is provided in `xgb.cv()`, the predictions would only be returned properly when this list is a non-overlapping list of k sets of indices, as in a standard k-fold CV. The predictions would not be meaningful when user-provided folds have overlapping indices as in, e.g., random sampling splits. When some of the indices in the training dataset are not included into user-provided `folds`, their prediction value would be `NA`.

Value

An `xgb.Callback` object, which can be passed to `xgb.cv()`, but **not** to `xgb.train()`.

`xgb.cb.early.stop` *Callback to activate early stopping*

Description

This callback function determines the condition for early stopping.

The following attributes are assigned to the booster's object:

- `best_score` the evaluation score at the best iteration
- `best_iteration` at which boosting iteration the best score has occurred (0-based index for interoperability of binary models)

The same values are also stored as R attributes as a result of the callback, plus an additional attribute `stopped_by_max_rounds` which indicates whether an early stopping by the `stopping_rounds` condition occurred. Note that the `best_iteration` that is stored under R attributes will follow base-1 indexing, so it will be larger by '1' than the C-level 'best_iteration' that is accessed through `xgb.attr()` or `xgb.attributes()`.

At least one dataset is required in `evals` for early stopping to work.

Usage

```
xgb.cb.early.stop(
  stopping_rounds,
  maximize = FALSE,
  metric_name = NULL,
  verbose = TRUE,
  save_best = FALSE
)
```

Arguments

`stopping_rounds`	The number of rounds with no improvement in the evaluation metric in order to stop the training.
`maximize`	Whether to maximize the evaluation metric.
`metric_name`	The name of an evaluation column to use as a criteria for early stopping. If not set, the last column would be used. Let's say the test data in `evals` was labelled as `dtest`, and one wants to use the AUC in test data for early stopping regardless of where it is in the `evals`, then one of the following would need to be set: `metric_name = 'dtest-auc'` or `metric_name = 'dtest_auc'`. All dash '-' characters in metric names are considered equivalent to '_'.
`verbose`	Whether to print the early stopping information.
`save_best`	Whether training should return the best model or the last model. If set to `TRUE`, it will only keep the boosting rounds up to the detected best iteration, discarding the ones that come after. This parameter is not supported by the `xgb.cv` function and the `gblinear` booster yet.

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

`xgb.cb.evaluation.log` *Callback for logging the evaluation history*

Description

Callback for logging the evaluation history

Usage

```
xgb.cb.evaluation.log()
```

Details

This callback creates a table with per-iteration evaluation metrics (see parameters `evals` and `custom_metric` in `xgb.train()`).

Note: in the column names of the final data.table, the dash '-' character is replaced with the underscore '_' in order to make the column names more like regular R identifiers.

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

See Also

xgb.cb.print.evaluation

`xgb.cb.gblinear.history` *Callback for collecting coefficients history of a gblinear booster*

Description

Callback for collecting coefficients history of a gblinear booster

Usage

```
xgb.cb.gblinear.history(sparse = FALSE)
```

Arguments

`sparse`	When set to `FALSE`/`TRUE`, a dense/sparse matrix is used to store the result. Sparse format is useful when one expects only a subset of coefficients to be non-zero, when using the "thrifty" feature selector with fairly small number of top features selected per iteration.

Details

To keep things fast and simple, gblinear booster does not internally store the history of linear model coefficients at each boosting iteration. This callback provides a workaround for storing the coefficients' path, by extracting them after each training iteration.

This callback will construct a matrix where rows are boosting iterations and columns are feature coefficients (same order as when calling coef.xgb.Booster, with the intercept corresponding to the first column).

When there is more than one coefficient per feature (e.g. multi-class classification), the result will be reshaped into a vector where coefficients are arranged first by features and then by class (e.g. first 1 through N coefficients will be for the first class, then coefficients N+1 through 2N for the second class, and so on).

If the result has only one coefficient per feature in the data, then the resulting matrix will have column names matching with the feature names, otherwise (when there's more than one coefficient per feature) the names will be composed as 'column name' + ':' + 'class index' (so e.g. column 'c1' for class '0' will be named 'c1:0').

With `xgb.train()`, the output is either a dense or a sparse matrix. With with `xgb.cv()`, it is a list (one element per each fold) of such matrices.

Function xgb.gblinear.history provides an easy way to retrieve the outputs from this callback.

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

See Also

xgb.gblinear.history, coef.xgb.Booster.

Examples

```
#### Binary classification:

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)

# In the iris dataset, it is hard to linearly separate Versicolor class from the rest
# without considering the 2nd order interactions:
x <- model.matrix(Species ~ .^2, iris)[, -1]
colnames(x)
dtrain <- xgb.DMatrix(
  scale(x),
  label = 1 * (iris$Species == "versicolor"),
  nthread = nthread
)
param <- xgb.params(
  booster = "gblinear",
  objective = "reg:logistic",
  eval_metric = "auc",
  reg_lambda = 0.0003,
  reg_alpha = 0.0003,
  nthread = nthread
)

# For 'shotgun', which is a default linear updater, using high learning_rate values may result in
# unstable behaviour in some datasets. With this simple dataset, however, the high learning
# rate does not break the convergence, but allows us to illustrate the typical pattern of
# "stochastic explosion" behaviour of this lock-free algorithm at early boosting iterations.
bst <- xgb.train(
  c(param, list(learning_rate = 1.)),
  dtrain,
  evals = list(tr = dtrain),
  nrounds = 200,
  callbacks = list(xgb.cb.gblinear.history())
)

# Extract the coefficients' path and plot them vs boosting iteration number:
coef_path <- xgb.gblinear.history(bst)
matplot(coef_path, type = "l")

# With the deterministic coordinate descent updater, it is safer to use higher learning rates.
# Will try the classical componentwise boosting which selects a single best feature per round:
bst <- xgb.train(
  c(
    param,
    xgb.params(
      learning_rate = 0.8,
      updater = "coord_descent",
      feature_selector = "thrifty",
      top_k = 1
    )
  ),
  dtrain,
  evals = list(tr = dtrain),
  nrounds = 200,
  callbacks = list(xgb.cb.gblinear.history())
)
matplot(xgb.gblinear.history(bst), type = "l")
#  Componentwise boosting is known to have similar effect to Lasso regularization.
# Try experimenting with various values of top_k, learning_rate, nrounds,
# as well as different feature_selectors.

# For xgb.cv:
bst <- xgb.cv(
  c(
    param,
    xgb.params(
      learning_rate = 0.8,
      updater = "coord_descent",
      feature_selector = "thrifty",
      top_k = 1
    )
  ),
  dtrain,
  nfold = 5,
  nrounds = 100,
  callbacks = list(xgb.cb.gblinear.history())
)
# coefficients in the CV fold #3
matplot(xgb.gblinear.history(bst)[[3]], type = "l")


#### Multiclass classification:
dtrain <- xgb.DMatrix(scale(x), label = as.numeric(iris$Species) - 1, nthread = nthread)

param <- xgb.params(
  booster = "gblinear",
  objective = "multi:softprob",
  num_class = 3,
  reg_lambda = 0.0003,
  reg_alpha = 0.0003,
  nthread = nthread
)

# For the default linear updater 'shotgun' it sometimes is helpful
# to use smaller learning_rate to reduce instability
bst <- xgb.train(
  c(param, list(learning_rate = 0.5)),
  dtrain,
  evals = list(tr = dtrain),
  nrounds = 50,
  callbacks = list(xgb.cb.gblinear.history())
)

# Will plot the coefficient paths separately for each class:
matplot(xgb.gblinear.history(bst, class_index = 0), type = "l")
matplot(xgb.gblinear.history(bst, class_index = 1), type = "l")
matplot(xgb.gblinear.history(bst, class_index = 2), type = "l")

# CV:
bst <- xgb.cv(
  c(param, list(learning_rate = 0.5)),
  dtrain,
  nfold = 5,
  nrounds = 70,
  callbacks = list(xgb.cb.gblinear.history(FALSE))
)
# 1st fold of 1st class
matplot(xgb.gblinear.history(bst, class_index = 0)[[1]], type = "l")
```

xgb.cb.print.evaluation

Callback for printing the result of evaluation

Description

The callback function prints the result of evaluation at every `period` iterations. The initial and the last iteration's evaluations are always printed.

Does not leave any attribute in the booster (see xgb.cb.evaluation.log for that).

Usage

```
xgb.cb.print.evaluation(period = 1, showsd = TRUE)
```

Arguments

`period`	Results would be printed every number of periods.
`showsd`	Whether standard deviations should be printed (when available).

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

See Also

xgb.Callback

`xgb.cb.reset.parameters` *Callback for resetting booster parameters at each iteration*

Description

Callback for resetting booster parameters at each iteration

Usage

```
xgb.cb.reset.parameters(new_params)
```

Arguments

`new_params`	List of parameters needed to be reset. Each element's value must be either a vector of values of length `nrounds` to be set at each iteration, or a function of two parameters `learning_rates(iteration, nrounds)` which returns a new parameter value by using the current iteration number and the total number of boosting rounds.

Details

Note that when training is resumed from some previous model, and a function is used to reset a parameter value, the `nrounds` argument in this function would be the the number of boosting rounds in the current training.

Does not leave any attribute in the booster.

Value

An `xgb.Callback` object, which can be passed to `xgb.train()` or `xgb.cv()`.

`xgb.cb.save.model` *Callback for saving a model file*

Description

This callback function allows to save an xgb-model file, either periodically after each `save_period`'s or at the end.

Does not leave any attribute in the booster.

Usage

```
xgb.cb.save.model(save_period = 0, save_name = "xgboost.ubj")
```

Arguments

`save_period`	Save the model to disk after every `save_period` iterations; 0 means save the model at the end.
`save_name`	The name or path for the saved model file. It can contain a `sprintf()` formatting specifier to include the integer iteration number in the file name. E.g., with `save_name = 'xgboost_%04d.model'`, the file saved at iteration 50 would be named "xgboost_0050.model".

Value

An `xgb.Callback` object, which can be passed to `xgb.train()`, but **not** to `xgb.cv()`.

`xgb.config` *Accessors for model parameters as JSON string*

Description

Accessors for model parameters as JSON string

Usage

```
xgb.config(object)

xgb.config(object) <- value
```

Arguments

`object`	Object of class `xgb.Booster`.**Will be modified in-place** when assigning to it.
`value`	A list.

Details

Note that assignment is performed in-place on the booster C object, which unlike assignment of R attributes, doesn't follow typical copy-on-write semantics for assignment - i.e. all references to the same booster will also get updated.

See `xgb.copy.Booster()` for an example of this behavior.

Value

Parameters as a list.

Examples

```
data(agaricus.train, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)
train <- agaricus.train

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

config <- xgb.config(bst)
```

`xgb.copy.Booster` *Deep-copies a Booster Object*

Description

Creates a deep copy of an 'xgb.Booster' object, such that the C object pointer contained will be a different object, and hence functions like `xgb.attr()` will not affect the object from which it was copied.

Usage

```
xgb.copy.Booster(model)
```

Arguments

`model`	An 'xgb.Booster' object.

Value

A deep copy of `model` - it will be identical in every way, but C-level functions called on that copy will not affect the `model` variable.

Examples

```
library(xgboost)

data(mtcars)

y <- mtcars$mpg
x <- mtcars[, -1]

dm <- xgb.DMatrix(x, label = y, nthread = 1)

model <- xgb.train(
  data = dm,
  params = xgb.params(nthread = 1),
  nrounds = 3
)

# Set an arbitrary attribute kept at the C level
xgb.attr(model, "my_attr") <- 100
print(xgb.attr(model, "my_attr"))

# Just assigning to a new variable will not create
# a deep copy - C object pointer is shared, and in-place
# modifications will affect both objects
model_shallow_copy <- model
xgb.attr(model_shallow_copy, "my_attr") <- 333
# 'model' was also affected by this change:
print(xgb.attr(model, "my_attr"))

model_deep_copy <- xgb.copy.Booster(model)
xgb.attr(model_deep_copy, "my_attr") <- 444
# 'model' was NOT affected by this change
# (keeps previous value that was assigned before)
print(xgb.attr(model, "my_attr"))

# Verify that the new object was actually modified
print(xgb.attr(model_deep_copy, "my_attr"))
```

`xgb.create.features` *Create new features from a previously learned model*

Description

May improve the learning by adding new features to the training data based on the decision trees from a previously learned model.

Usage

```
xgb.create.features(model, data)
```

Arguments

`model`	Decision tree boosting model learned on the original data.
`data`	Original data (usually provided as a `dgCMatrix` matrix).

Details

This is the function inspired from the paragraph 3.1 of the paper:

Practical Lessons from Predicting Clicks on Ads at Facebook

(Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yan, xin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, Joaquin Quinonero Candela)

International Workshop on Data Mining for Online Advertising (ADKDD) - August 24, 2014

`https://research.facebook.com/publications/practical-lessons-from-predicting-clicks-on-ads-at-faceb`

Extract explaining the method:

"We found that boosted decision trees are a powerful and very convenient way to implement non-linear and tuple transformations of the kind we just described. We treat each individual tree as a categorical feature that takes as value the index of the leaf an instance ends up falling in. We use 1-of-K coding of this type of features.

For example, consider the boosted tree model in Figure 1 with 2 subtrees, where the first subtree has 3 leafs and the second 2 leafs. If an instance ends up in leaf 2 in the first subtree and leaf 1 in second subtree, the overall input to the linear classifier will be the binary vector `[0, 1, 0, 1, 0]`, where the first 3 entries correspond to the leaves of the first subtree and last 2 to those of the second subtree.

...

We can understand boosted decision tree based transformation as a supervised feature encoding that converts a real-valued vector into a compact binary-valued vector. A traversal from root node to a leaf node represents a rule on certain features."

Value

A `dgCMatrix` matrix including both the original data and the new features.

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))
dtest <- with(agaricus.test, xgb.DMatrix(data, label = label, nthread = 2))

param <- list(max_depth = 2, learning_rate = 1, objective = 'binary:logistic', nthread = 1)
nrounds = 4
```

```
bst <- xgb.train(params = param, data = dtrain, nrounds = nrounds)

# Model accuracy without new features
accuracy.before <- sum((predict(bst, agaricus.test$data) >= 0.5) == agaricus.test$label) /
                   length(agaricus.test$label)

# Convert previous features to one hot encoding
new.features.train <- xgb.create.features(model = bst, agaricus.train$data)
new.features.test <- xgb.create.features(model = bst, agaricus.test$data)

# learning with new features
new.dtrain <- xgb.DMatrix(
  data = new.features.train, label = agaricus.train$label, nthread = 1
)
new.dtest <- xgb.DMatrix(
  data = new.features.test, label = agaricus.test$label, nthread = 1
)
bst <- xgb.train(params = param, data = new.dtrain, nrounds = nrounds)

# Model accuracy with new features
accuracy.after <- sum((predict(bst, new.dtest) >= 0.5) == agaricus.test$label) /
                  length(agaricus.test$label)

# Here the accuracy was already good and is now perfect.
cat(paste("The accuracy was", accuracy.before, "before adding leaf features and it is now",
          accuracy.after, "!\n"))
```

xgb.cv *Cross Validation*

Description

The cross validation function of xgboost.

Usage

```
xgb.cv(
  params = xgb.params(),
  data,
  nrounds,
  nfold,
  prediction = FALSE,
  showsd = TRUE,
  metrics = list(),
  objective = NULL,
  custom_metric = NULL,
  stratified = "auto",
  folds = NULL,
  train_folds = NULL,
  verbose = TRUE,
  print_every_n = 1L,
  early_stopping_rounds = NULL,
  maximize = NULL,
  callbacks = list(),
  ...
)
```

Arguments

`params`	List of XGBoost parameters which control the model building process. See the online documentation and the documentation for `xgb.params()` for details. Should be passed as list with named entries. Parameters that are not specified in this list will use their default values. A list of named parameters can be created through the function `xgb.params()`, which accepts all valid parameters as function arguments.
`data`	An `xgb.DMatrix` object, with corresponding fields like `label` or bounds as required for model training by the objective. Note that only the basic `xgb.DMatrix` class is supported - variants such as `xgb.QuantileDMatrix` or `xgb.ExtMemDMatrix` are not supported here.
`nrounds`	Max number of boosting iterations.
`nfold`	The original dataset is randomly partitioned into `nfold` equal size subsamples.
`prediction`	A logical value indicating whether to return the test fold predictions from each CV model. This parameter engages the `xgb.cb.cv.predict()` callback.
`showsd`	Logical value whether to show standard deviation of cross validation.
`metrics`	List of evaluation metrics to be used in cross validation, when it is not specified, the evaluation metric is chosen according to objective function. Possible options are: • `error`: Binary classification error rate • `rmse`: Root mean square error • `logloss`: Negative log-likelihood function • `mae`: Mean absolute error • `mape`: Mean absolute percentage error • `auc`: Area under curve • `aucpr`: Area under PR curve • `merror`: Exact matching error used to evaluate multi-class classification
`objective`	Customized objective function. Should take two arguments: the first one will be the current predictions (either a numeric vector or matrix depending on the number of targets / classes), and the second one will be the `data` DMatrix object that is used for training. It should return a list with two elements `grad` and `hess` (in that order), as either numeric vectors or numeric matrices depending on the number of targets / classes (same dimension as the predictions that are passed as first argument).

`custom_metric` Customized evaluation function. Just like `objective`, should take two arguments, with the first one being the predictions and the second one the `data` DMatrix.

Should return a list with two elements `metric` (name that will be displayed for this metric, should be a string / character), and `value` (the number that the function calculates, should be a numeric scalar).

Note that even if passing `custom_metric`, objectives also have an associated default metric that will be evaluated in addition to it. In order to disable the built-in metric, one can pass parameter `disable_default_eval_metric = TRUE`.

`stratified` Logical flag indicating whether sampling of folds should be stratified by the values of outcome labels. For real-valued labels in regression objectives, stratification will be done by discretizing the labels into up to 5 buckets beforehand.

If passing "auto", will be set to `TRUE` if the objective in `params` is a classification objective (from XGBoost's built-in objectives, doesn't apply to custom ones), and to `FALSE` otherwise.

This parameter is ignored when `data` has a `group` field - in such case, the splitting will be based on whole groups (note that this might make the folds have different sizes).

Value `TRUE` here is **not** supported for custom objectives.

`folds` List with pre-defined CV folds (each element must be a vector of test fold's indices). When folds are supplied, the `nfold` and `stratified` parameters are ignored.

If `data` has a `group` field and the objective requires this field, each fold (list element) must additionally have two attributes (retrievable through `attributes`) named `group_test` and `group_train`, which should hold the `group` to assign through `setinfo.xgb.DMatrix()` to the resulting DMatrices.

`train_folds` List specifying which indices to use for training. If `NULL` (the default) all indices not specified in `folds` will be used for training.

This is not supported when `data` has `group` field.

`verbose` If 0, xgboost will stay silent. If 1, it will print information about performance. If 2, some additional information will be printed out. Note that setting `verbose > 0` automatically engages the `xgb.cb.print.evaluation(period=1)` callback function.

`print_every_n` When passing `verbose>0`, evaluation logs (metrics calculated on the data passed under `evals`) will be printed every nth iteration according to the value passed here. The first and last iteration are always included regardless of this 'n'.

Only has an effect when passing data under `evals` and when passing `verbose>0`. The parameter is passed to the `xgb.cb.print.evaluation()` callback.

`early_stopping_rounds`

Number of boosting rounds after which training will be stopped if there is no improvement in performance (as measured by the evaluatiation metric that is supplied or selected by default for the objective) on the evaluation data passed under `evals`.

Must pass `evals` in order to use this functionality. Setting this parameter adds the `xgb.cb.early.stop()` callback.

If `NULL`, early stopping will not be used.

`maximize`	If `feval` and `early_stopping_rounds` are set, then this parameter must be set as well. When it is `TRUE`, it means the larger the evaluation score the better. This parameter is passed to the `xgb.cb.early.stop()` callback.
`callbacks`	A list of callback functions to perform various task during boosting. See `xgb.Callback()`. Some of the callbacks are automatically created depending on the parameters' values. User can provide either existing or their own callback methods in order to customize the training process.
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

The original sample is randomly partitioned into `nfold` equal size subsamples.

Of the `nfold` subsamples, a single subsample is retained as the validation data for testing the model, and the remaining `nfold - 1` subsamples are used as training data.

The cross-validation process is then repeated `nrounds` times, with each of the `nfold` subsamples used exactly once as the validation data.

All observations are used for both training and validation.

Adapted from `https://en.wikipedia.org/wiki/Cross-validation_%28statistics%29`

Value

An object of class 'xgb.cv.synchronous' with the following elements:

- `call`: Function call.
- `params`: Parameters that were passed to the xgboost library. Note that it does not capture parameters changed by the `xgb.cb.reset.parameters()` callback.
- `evaluation_log`: Evaluation history stored as a `data.table` with the first column corresponding to iteration number and the rest corresponding to the CV-based evaluation means and standard deviations for the training and test CV-sets. It is created by the `xgb.cb.evaluation.log()` callback.
- `niter`: Number of boosting iterations.
- `nfeatures`: Number of features in training data.
- `folds`: The list of CV folds' indices - either those passed through the `folds` parameter or randomly generated.

Plus other potential elements that are the result of callbacks, such as a list `cv_predict` with a sub-element `pred` when passing `prediction` = `TRUE`, which is added by the `xgb.cb.cv.predict()` callback (note that one can also pass it manually under `callbacks` with different settings, such as saving also the models created during cross validation); or a list `early_stop` which will contain elements such as `best_iteration` when using the early stopping callback (`xgb.cb.early.stop()`).

Examples

```
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))

cv <- xgb.cv(
  data = dtrain,
  nrounds = 20,
  early_stopping_rounds = 1,
  params = xgb.params(
    nthread = 2,
    max_depth = 3,
    objective = "binary:logistic"
  ),
  nfold = 5,
  metrics = list("rmse","auc"),
  prediction = TRUE
)
print(cv)
print(cv, verbose = TRUE)

# Callbacks might add additional attributes, separated by the name of the callback
cv$early_stop$best_iteration
head(cv$cv_predict$pred)
```

`xgb.DataBatch` *Structure for Data Batches*

Description

Helper function to supply data in batches of a data iterator when constructing a DMatrix from external memory through `xgb.ExtMemDMatrix()` or through `xgb.QuantileDMatrix.from_iterator()`.

This function is **only** meant to be called inside of a callback function (which is passed as argument to function `xgb.DataIter()` to construct a data iterator) when constructing a DMatrix through external memory - otherwise, one should call `xgb.DMatrix()` or `xgb.QuantileDMatrix()`.

The object that results from calling this function directly is **not** like an `xgb.DMatrix` - i.e. cannot be used to train a model, nor to get predictions - only possible usage is to supply data to an iterator, from which a DMatrix is then constructed.

For more information and for example usage, see the documentation for `xgb.ExtMemDMatrix()`.

Usage

```
xgb.DataBatch(
  data,
  label = NULL,
  weight = NULL,
  base_margin = NULL,
  feature_names = colnames(data),
  feature_types = NULL,
  group = NULL,
  qid = NULL,
  label_lower_bound = NULL,
  label_upper_bound = NULL,
  feature_weights = NULL
)
```

Arguments

`data`	Batch of data belonging to this batch. Note that not all of the input types supported by `xgb.DMatrix()` are possible to pass here. Supported types are: • `matrix`, with types `numeric`, `integer`, and `logical`. Note that for types `integer` and `logical`, missing values might not be automatically recognized as as such - see the documentation for parameter `missing` in `xgb.ExtMemDMatrix()` for details on this. • `data.frame`, with the same types as supported by 'xgb.DMatrix' and same conversions applied to it. See the documentation for parameter `data` in `xgb.DMatrix()` for details on it. • CSR matrices, as class `dgRMatrix` from package "Matrix".
`label`	Label of the training data. For classification problems, should be passed encoded as integers with numeration starting at zero.
`weight`	Weight for each instance. Note that, for ranking task, weights are per-group. In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.
`base_margin`	Base margin used for boosting from existing model. In the case of multi-output models, one can also pass multi-dimensional base_margin.
`feature_names`	Set names for features. Overrides column names in data frame and matrix. Note: columns are not referenced by name when calling `predict`, so the column order there must be the same as in the DMatrix construction, regardless of the column names.
`feature_types`	Set types for features. If `data` is a `data.frame` and passing `feature_types` is not supplied, feature types will be deduced automatically from the column types. Otherwise, one can pass a character vector with the same length as number of columns in `data`, with the following possible values:

- "c", which represents categorical columns.
- "q", which represents numeric columns.
- "int", which represents integer columns.
- "i", which represents logical (boolean) columns.

Note that, while categorical types are treated differently from the rest for model fitting purposes, the other types do not influence the generated model, but have effects in other functionalities such as feature importances.

Important: Categorical features, if specified manually through `feature_types`, must be encoded as integers with numeration starting at zero, and the same encoding needs to be applied when passing data to `predict()`. Even if passing `factor` types, the encoding will not be saved, so make sure that `factor` columns passed to `predict` have the same `levels`.

`group`	Group size for all ranking group.
`qid`	Query ID for data samples, used for ranking.
`label_lower_bound`	Lower bound for survival training.
`label_upper_bound`	Upper bound for survival training.
`feature_weights`	Set feature weights for column sampling.

Value

An object of class `xgb.DataBatch`, which is just a list containing the data and parameters passed here. It does **not** inherit from `xgb.DMatrix`.

See Also

`xgb.DataIter()`, `xgb.ExtMemDMatrix()`.

`xgb.DataIter` *XGBoost Data Iterator*

Description

Interface to create a custom data iterator in order to construct a DMatrix from external memory.

This function is responsible for generating an R object structure containing callback functions and an environment shared with them.

The output structure from this function is then meant to be passed to `xgb.ExtMemDMatrix()`, which will consume the data and create a DMatrix from it by executing the callback functions.

For more information, and for a usage example, see the documentation for `xgb.ExtMemDMatrix()`.

Usage

```
xgb.DataIter(env = new.env(), f_next, f_reset)
```

Arguments

`env`	An R environment to pass to the callback functions supplied here, which can be used to keep track of variables to determine how to handle the batches. For example, one might want to keep track of an iteration number in this environment in order to know which part of the data to pass next.
`f_next`	`function(env)` which is responsible for: • Accessing or retrieving the next batch of data in the iterator. • Supplying this data by calling function `xgb.DataBatch()` on it and returning the result. • Keeping track of where in the iterator batch it is or will go next, which can for example be done by modifiying variables in the `env` variable that is passed here. • Signaling whether there are more batches to be consumed or not, by returning `NULL` when the stream of data ends (all batches in the iterator have been consumed), or the result from calling `xgb.DataBatch()` when there are more batches in the line to be consumed.
`f_reset`	`function(env)` which is responsible for reseting the data iterator (i.e. taking it back to the first batch, called before and after the sequence of batches has been consumed). Note that, after resetting the iterator, the batches will be accessed again, so the same data (and in the same order) must be passed in subsequent iterations.

Value

An `xgb.DataIter` object, containing the same inputs supplied here, which can then be passed to `xgb.ExtMemDMatrix()`.

See Also

`xgb.ExtMemDMatrix()`, `xgb.DataBatch()`.

`xgb.DMatrix` *Construct xgb.DMatrix object*

Description

Construct an 'xgb.DMatrix' object from a given data source, which can then be passed to functions such as `xgb.train()` or `predict()`.

Usage

```
xgb.DMatrix(
  data,
  label = NULL,
  weight = NULL,
  base_margin = NULL,
  missing = NA,
  silent = FALSE,
  feature_names = colnames(data),
  feature_types = NULL,
  nthread = NULL,
  group = NULL,
  qid = NULL,
  label_lower_bound = NULL,
  label_upper_bound = NULL,
  feature_weights = NULL,
  data_split_mode = "row",
  ...
)

xgb.QuantileDMatrix(
  data,
  label = NULL,
  weight = NULL,
  base_margin = NULL,
  missing = NA,
  feature_names = colnames(data),
  feature_types = NULL,
  nthread = NULL,
  group = NULL,
  qid = NULL,
  label_lower_bound = NULL,
  label_upper_bound = NULL,
  feature_weights = NULL,
  ref = NULL,
  max_bin = NULL
)
```

Arguments

`data` Data from which to create a DMatrix, which can then be used for fitting models or for getting predictions out of a fitted model.

Supported input types are as follows:

- `matrix` objects, with types `numeric`, `integer`, or `logical`.
- `data.frame` objects, with columns of types `numeric`, `integer`, `logical`, or `factor`

Note that xgboost uses base-0 encoding for categorical types, hence `factor` types (which use base-1 encoding') will be converted inside the function call. Be aware that the encoding used for `factor` types is not kept as part of the model, so in subsequent calls to `predict`, it is the user's responsibility to ensure that factor columns have the same levels as the ones from which the DMatrix was constructed.

Other column types are not supported.

- CSR matrices, as class `dgRMatrix` from package `Matrix`.
- CSC matrices, as class `dgCMatrix` from package `Matrix`.

These are **not** supported by `xgb.QuantileDMatrix`.

- XGBoost's own binary format for DMatrices, as produced by `xgb.DMatrix.save()`.
- Single-row CSR matrices, as class `dsparseVector` from package `Matrix`, which is interpreted as a single row (only when making predictions from a fitted model).

`label` Label of the training data. For classification problems, should be passed encoded as integers with numeration starting at zero.

`weight` Weight for each instance.

Note that, for ranking task, weights are per-group. In ranking task, one weight is assigned to each group (not each data point). This is because we only care about the relative ordering of data points within each group, so it doesn't make sense to assign weights to individual data points.

`base_margin` Base margin used for boosting from existing model.

In the case of multi-output models, one can also pass multi-dimensional base_margin.

`missing` A float value to represents missing values in data (not used when creating DMatrix from text files). It is useful to change when a zero, infinite, or some other extreme value represents missing values in data.

`silent` whether to suppress printing an informational message after loading from a file.

`feature_names` Set names for features. Overrides column names in data frame and matrix.

Note: columns are not referenced by name when calling `predict`, so the column order there must be the same as in the DMatrix construction, regardless of the column names.

`feature_types` Set types for features.

If `data` is a `data.frame` and passing `feature_types` is not supplied, feature types will be deduced automatically from the column types.

Otherwise, one can pass a character vector with the same length as number of columns in `data`, with the following possible values:

- "c", which represents categorical columns.
- "q", which represents numeric columns.
- "int", which represents integer columns.
- "i", which represents logical (boolean) columns.

Note that, while categorical types are treated differently from the rest for model fitting purposes, the other types do not influence the generated model, but have effects in other functionalities such as feature importances.

Important: Categorical features, if specified manually through `feature_types`, must be encoded as integers with numeration starting at zero, and the same encoding needs to be applied when passing data to `predict()`. Even if passing `factor` types, the encoding will not be saved, so make sure that `factor` columns passed to `predict` have the same `levels`.

`nthread` Number of threads used for creating DMatrix.

`group` Group size for all ranking group.

`qid`	Query ID for data samples, used for ranking.
`label_lower_bound`	Lower bound for survival training.
`label_upper_bound`	Upper bound for survival training.
`feature_weights`	Set feature weights for column sampling.
`data_split_mode`	Not used yet. This parameter is for distributed training, which is not yet available for the R package.
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.
`ref`	The training dataset that provides quantile information, needed when creating validation/test dataset with `xgb.QuantileDMatrix()`. Supplying the training DMatrix as a reference means that the same quantisation applied to the training data is applied to the validation/test data
`max_bin`	The number of histogram bin, should be consistent with the training parameter `max_bin`. This is only supported when constructing a QuantileDMatrix.

Details

Function `xgb.QuantileDMatrix()` will construct a DMatrix with quantization for the histogram method already applied to it, which can be used to reduce memory usage (compared to using a a regular DMatrix first and then creating a quantization out of it) when using the histogram method (`tree_method = "hist"`, which is the default algorithm), but is not usable for the sorted-indices method (`tree_method = "exact"`), nor for the approximate method (`tree_method = "approx"`).

Note that DMatrix objects are not serializable through R functions such as `saveRDS()` or `save()`. If a DMatrix gets serialized and then de-serialized (for example, when saving data in an R session or caching chunks in an Rmd file), the resulting object will not be usable anymore and will need to be reconstructed from the original source of data.

Value

An 'xgb.DMatrix' object. If calling `xgb.QuantileDMatrix`, it will have additional subclass `xgb.QuantileDMatrix`.

Examples

```
data(agaricus.train, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)
dtrain <- with(
  agaricus.train, xgb.DMatrix(data, label = label, nthread = nthread)
)
fname <- file.path(tempdir(), "xgb.DMatrix.data")
xgb.DMatrix.save(dtrain, fname)
dtrain <- xgb.DMatrix(fname, nthread = 1)
```

`xgb.DMatrix.hasinfo` *Check whether DMatrix object has a field*

Description

Checks whether an xgb.DMatrix object has a given field assigned to it, such as weights, labels, etc.

Usage

```
xgb.DMatrix.hasinfo(object, info)
```

Arguments

`object`	The DMatrix object to check for the given `info` field.
`info`	The field to check for presence or absence in `object`.

See Also

`xgb.DMatrix()`, `getinfo.xgb.DMatrix()`, `setinfo.xgb.DMatrix()`

Examples

```
x <- matrix(1:10, nrow = 5)
dm <- xgb.DMatrix(x, nthread = 1)

# 'dm' so far does not have any fields set
xgb.DMatrix.hasinfo(dm, "label")

# Fields can be added after construction
setinfo(dm, "label", 1:5)
xgb.DMatrix.hasinfo(dm, "label")
```

`xgb.DMatrix.save` *Save xgb.DMatrix object to binary file*

Description

Save xgb.DMatrix object to binary file

Usage

```
xgb.DMatrix.save(dmatrix, fname)
```

Arguments

`dmatrix`	the `xgb.DMatrix` object
`fname`	the name of the file to write.

Examples

```
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))
fname <- file.path(tempdir(), "xgb.DMatrix.data")
xgb.DMatrix.save(dtrain, fname)
dtrain <- xgb.DMatrix(fname, nthread = 1)
```

`xgb.dump` *Dump an XGBoost model in text format.*

Description

Dump an XGBoost model in text format.

Usage

```
xgb.dump(
  model,
  fname = NULL,
  fmap = "",
  with_stats = FALSE,
  dump_format = c("text", "json", "dot"),
  ...
)
```

Arguments

`model`	The model object.
`fname`	The name of the text file where to save the model text dump. If not provided or set to `NULL`, the model is returned as a character vector.
`fmap`	Feature map file representing feature types. See demo/ for a walkthrough example in R, and `https://github.com/dmlc/xgboost/blob/master/demo/data/featmap.txt` to see an example of the value.
`with_stats`	Whether to dump some additional statistics about the splits. When this option is on, the model dump contains two additional values: gain is the approximate loss function gain we get in each split; cover is the sum of second order gradient in each node.
`dump_format`	Either 'text', 'json', or 'dot' (graphviz) format could be specified. Format 'dot' for a single tree can be passed directly to packages that consume this format for graph visualization, such as function `DiagrammeR::grViz()`
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Value

If fname is not provided or set to `NULL` the function will return the model as a character vector. Otherwise it will return `TRUE`.

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

train <- agaricus.train
test <- agaricus.test

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = 2,
    objective = "binary:logistic"
  )
)

# save the model in file 'xgb.model.dump'
dump_path = file.path(tempdir(), 'model.dump')
xgb.dump(bst, dump_path, with_stats = TRUE)

# print the model without saving it to a file
print(xgb.dump(bst, with_stats = TRUE))

# print in JSON format:
cat(xgb.dump(bst, with_stats = TRUE, dump_format = "json"))

# plot first tree leveraging the 'dot' format
if (requireNamespace('DiagrammeR', quietly = TRUE)) {
  DiagrammeR::grViz(xgb.dump(bst, dump_format = "dot")[[1L]])
}
```

`xgb.ExtMemDMatrix` *DMatrix from External Data*

Description

Create a special type of XGBoost 'DMatrix' object from external data supplied by an `xgb.DataIter()` object, potentially passed in batches from a bigger set that might not fit entirely in memory.

The data supplied by the iterator is accessed on-demand as needed, multiple times, without being concatenated, but note that fields like 'label' **will** be concatenated from multiple calls to the data iterator.

For more information, see the guide 'Using XGBoost External Memory Version': https://xgboost.readthedocs.io/en/stable/tutorials/external_memory.html

Usage

```
xgb.ExtMemDMatrix(
  data_iterator,
  cache_prefix = tempdir(),
  missing = NA,
  nthread = NULL
)
```

Arguments

`data_iterator`	A data iterator structure as returned by `xgb.DataIter()`, which includes an environment shared between function calls, and functions to access the data in batches on-demand.
`cache_prefix`	The path of cache file, caller must initialize all the directories in this path.

`missing`	A float value to represents missing values in data. Note that, while functions like `xgb.DMatrix()` can take a generic `NA` and interpret it correctly for different types like `numeric` and `integer`, if an `NA` value is passed here, it will not be adapted for different input types. For example, in R `integer` types, missing values are represented by integer number `-2147483648` (since machine 'integer' types do not have an inherent 'NA' value) - hence, if one passes `NA`, which is interpreted as a floating-point NaN by `xgb.ExtMemDMatrix()` and by `xgb.QuantileDMatrix.from_iterator()`, these integer missing values will not be treated as missing. This should not pose any problem for `numeric` types, since they do have an inheret NaN value.
`nthread`	Number of threads used for creating DMatrix.

Details

Be aware that construction of external data DMatrices **will cache data on disk** in a compressed format, under the path supplied in `cache_prefix`.

External data is not supported for the exact tree method.

Value

An 'xgb.DMatrix' object, with subclass 'xgb.ExtMemDMatrix', in which the data is not held internally but accessed through the iterator when needed.

See Also

`xgb.DataIter()`, `xgb.DataBatch()`, `xgb.QuantileDMatrix.from_iterator()`

Examples

```
data(mtcars)

# This custom environment will be passed to the iterator
# functions at each call. It is up to the user to keep
# track of the iteration number in this environment.
iterator_env <- as.environment(
  list(
    iter = 0,
    x = mtcars[, -1],
    y = mtcars[, 1]
  )
)

# Data is passed in two batches.
# In this example, batches are obtained by subsetting the 'x' variable.
# This is not advantageous to do, since the data is already loaded in memory
# and can be passed in full in one go, but there can be situations in which
# only a subset of the data will fit in the computer's memory, and it can
# be loaded in batches that are accessed one-at-a-time only.
iterator_next <- function(iterator_env) {
  curr_iter <- iterator_env[["iter"]]
```

```
    if (curr_iter >= 2) {
      # there are only two batches, so this signals end of the stream
      return(NULL)
    }

    if (curr_iter == 0) {
      x_batch <- iterator_env[["x"]][1:16, ]
      y_batch <- iterator_env[["y"]][1:16]
    } else {
      x_batch <- iterator_env[["x"]][17:32, ]
      y_batch <- iterator_env[["y"]][17:32]
    }
    on.exit({
      iterator_env[["iter"]] <- curr_iter + 1
    })

    # Function 'xgb.DataBatch' must be called manually
    # at each batch with all the appropriate attributes,
    # such as feature names and feature types.
    return(xgb.DataBatch(data = x_batch, label = y_batch))
  }

  # This moves the iterator back to its beginning
  iterator_reset <- function(iterator_env) {
    iterator_env[["iter"]] <- 0
  }

  data_iterator <- xgb.DataIter(
    env = iterator_env,
    f_next = iterator_next,
    f_reset = iterator_reset
  )
  cache_prefix <- tempdir()

  # DMatrix will be constructed from the iterator's batches
  dm <- xgb.ExtMemDMatrix(data_iterator, cache_prefix, nthread = 1)

  # After construction, can be used as a regular DMatrix
  params <- xgb.params(nthread = 1, objective = "reg:squarederror")
  model <- xgb.train(data = dm, nrounds = 2, params = params)

  # Predictions can also be called on it, and should be the same
  # as if the data were passed differently.
  pred_dm <- predict(model, dm)
  pred_mat <- predict(model, as.matrix(mtcars[, -1]))
```

`xgb.gblinear.history` *Extract gblinear coefficients history*

Description

A helper function to extract the matrix of linear coefficients' history from a gblinear model created while using the xgb.cb.gblinear.history callback (which must be added manually as by default it is not used).

Usage

```
xgb.gblinear.history(model, class_index = NULL)
```

Arguments

`model`	Either an `xgb.Booster` or a result of `xgb.cv()`, trained using the xgb.cb.gblinear.history callback, but **not** a booster loaded from `xgb.load()` or `xgb.load.raw()`.
`class_index`	zero-based class index to extract the coefficients for only that specific class in a multinomial multiclass model. When it is `NULL`, all the coefficients are returned. Has no effect in non-multiclass models.

Details

Note that this is an R-specific function that relies on R attributes that are not saved when using XGBoost's own serialization functions like `xgb.load()` or `xgb.load.raw()`.

In order for a serialized model to be accepted by this function, one must use R serializers such as `saveRDS()`.

Value

For an `xgb.train()` result, a matrix (either dense or sparse) with the columns corresponding to iteration's coefficients and the rows corresponding to boosting iterations.

For an `xgb.cv()` result, a list of such matrices is returned with the elements corresponding to CV folds.

When there is more than one coefficient per feature (e.g. multi-class classification) and `class_index` is not provided, the result will be reshaped into a vector where coefficients are arranged first by features and then by class (e.g. first 1 through N coefficients will be for the first class, then coefficients N+1 through 2N for the second class, and so on).

See Also

xgb.cb.gblinear.history, coef.xgb.Booster.

`xgb.get.DMatrix.data` *Get DMatrix Data*

Description

Get DMatrix Data

Usage

```
xgb.get.DMatrix.data(dmat)
```

Arguments

`dmat`	An `xgb.DMatrix` object, as returned by `xgb.DMatrix()`.

Value

The data held in the DMatrix, as a sparse CSR matrix (class `dgRMatrix` from package `Matrix`). If it had feature names, these will be added as column names in the output.

`xgb.get.DMatrix.num.non.missing` *Get Number of Non-Missing Entries in DMatrix*

Description

Get Number of Non-Missing Entries in DMatrix

Usage

```
xgb.get.DMatrix.num.non.missing(dmat)
```

Arguments

`dmat`	An `xgb.DMatrix` object, as returned by `xgb.DMatrix()`.

Value

The number of non-missing entries in the DMatrix.

xgb.get.DMatrix.qcut *Get Quantile Cuts from DMatrix*

Description

Get the quantile cuts (a.k.a. borders) from an `xgb.DMatrix` that has been quantized for the histogram method (`tree_method = "hist"`).

These cuts are used in order to assign observations to bins - i.e. these are ordered boundaries which are used to determine assignment condition `border_low < x < border_high`. As such, the first and last bin will be outside of the range of the data, so as to include all of the observations there.

If a given column has 'n' bins, then there will be 'n+1' cuts / borders for that column, which will be output in sorted order from lowest to highest.

Different columns can have different numbers of bins according to their range.

Usage

```
xgb.get.DMatrix.qcut(dmat, output = c("list", "arrays"))
```

Arguments

`dmat`	An `xgb.DMatrix` object, as returned by `xgb.DMatrix()`.
`output`	Output format for the quantile cuts. Possible options are: • "list"`will return the output as a list with one entry per column, where each column wil` has column names assigned to it. • `"arrays"` will return a list with entries `indptr` (base-0 indexing) and `data`. Here, the cuts for column 'i' are obtained by slicing 'data' from entries `indptr[i]+1` to `indptr[i+1]`.

Value

The quantile cuts, in the format specified by parameter `output`.

Examples

```
data(mtcars)

y <- mtcars$mpg
x <- as.matrix(mtcars[, -1])
dm <- xgb.DMatrix(x, label = y, nthread = 1)

# DMatrix is not quantized right away, but will be once a hist model is generated
model <- xgb.train(
  data = dm,
  params = xgb.params(tree_method = "hist", max_bin = 8, nthread = 1),
  nrounds = 3
)
```

```
# Now can get the quantile cuts
xgb.get.DMatrix.qcut(dm)
```

`xgb.get.num.boosted.rounds` *Get number of boosting in a fitted booster*

Description

Get number of boosting in a fitted booster

Usage

```
xgb.get.num.boosted.rounds(model)

## S3 method for class 'xgb.Booster'
length(x)
```

Arguments

`model`, `x`	A fitted `xgb.Booster` model.

Details

Note that setting booster parameters related to training continuation / updates through `xgb.model.parameters<-()` will reset the number of rounds to zero.

Value

The number of rounds saved in the model as an integer.

`xgb.ggplot.deepness` *Plot model tree depth*

Description

Visualizes distributions related to the depth of tree leaves.

- `xgb.plot.deepness()` uses base R graphics, while
- `xgb.ggplot.deepness()` uses "ggplot2".

Usage

```
xgb.ggplot.deepness(
  model = NULL,
  which = c("2x1", "max.depth", "med.depth", "med.weight")
)

xgb.plot.deepness(
  model = NULL,
  which = c("2x1", "max.depth", "med.depth", "med.weight"),
  plot = TRUE,
  ...
)
```

Arguments

`model`	Either an `xgb.Booster` model, or the "data.table" returned by `xgb.model.dt.tree()`.
`which`	Which distribution to plot (see details).
`plot`	Should the plot be shown? Default is `TRUE`.
`...`	Other parameters passed to `graphics::barplot()` or `graphics::plot()`.

Details

When `which = "2x1"`, two distributions with respect to the leaf depth are plotted on top of each other:

1. The distribution of the number of leaves in a tree model at a certain depth.
2. The distribution of the average weighted number of observations ("cover") ending up in leaves at a certain depth.

Those could be helpful in determining sensible ranges of the `max_depth` and `min_child_weight` parameters.

When `which = "max.depth"` or `which = "med.depth"`, plots of either maximum or median depth per tree with respect to the tree number are created.

Finally, `which = "med.weight"` allows to see how a tree's median absolute leaf weight changes through the iterations.

These functions have been inspired by the blog post `https://github.com/aysent/random-forest-leaf-visualization`

Value

The return value of the two functions is as follows:

- `xgb.plot.deepness()`: A "data.table" (invisibly). Each row corresponds to a terminal leaf in the model. It contains its information about depth, cover, and weight (used in calculating predictions). If `plot = TRUE`, also a plot is shown.
- `xgb.ggplot.deepness()`: When `which = "2x1"`, a list of two "ggplot" objects, and a single "ggplot" object otherwise.

See Also

`xgb.train()` and `xgb.model.dt.tree()`.

Examples

```
data(agaricus.train, package = "xgboost")
## Keep the number of threads to 2 for examples
nthread <- 2
data.table::setDTthreads(nthread)

## Change max_depth to a higher number to get a more significant result
model <- xgboost(
  agaricus.train$data, factor(agaricus.train$label),
  nrounds = 50,
  max_depth = 6,
  nthreads = nthread,
  subsample = 0.5,
  min_child_weight = 2
)

xgb.plot.deepness(model)
xgb.ggplot.deepness(model)

xgb.plot.deepness(
  model, which = "max.depth", pch = 16, col = rgb(0, 0, 1, 0.3), cex = 2
)

xgb.plot.deepness(
  model, which = "med.weight", pch = 16, col = rgb(0, 0, 1, 0.3), cex = 2
)
```

`xgb.ggplot.importance` *Plot feature importance*

Description

Represents previously calculated feature importance as a bar graph.

- `xgb.plot.importance()` uses base R graphics, while
- `xgb.ggplot.importance()` uses "ggplot".

Usage

```
xgb.ggplot.importance(
  importance_matrix = NULL,
  top_n = NULL,
  measure = NULL,
  rel_to_first = FALSE,
  n_clusters = seq_len(10),
  ...
)

xgb.plot.importance(
  importance_matrix = NULL,
  top_n = NULL,
  measure = NULL,
  rel_to_first = FALSE,
  left_margin = 10,
  cex = NULL,
  plot = TRUE,
  ...
)
```

Arguments

`importance_matrix`	A `data.table` as returned by `xgb.importance()`.
`top_n`	Maximal number of top features to include into the plot.
`measure`	The name of importance measure to plot. When `NULL`, 'Gain' would be used for trees and 'Weight' would be used for gblinear.
`rel_to_first`	Whether importance values should be represented as relative to the highest ranked feature, see Details.
`n_clusters`	A numeric vector containing the min and the max range of the possible number of clusters of bars.
`...`	Other parameters passed to `graphics::barplot()` (except `horiz`, `border`, `cex.names`, `names.arg`, and `las`). Only used in `xgb.plot.importance()`.
`left_margin`	Adjust the left margin size to fit feature names. When `NULL`, the existing `par("mar")` is used.
`cex`	Passed as `cex.names` parameter to `graphics::barplot()`.
`plot`	Should the barplot be shown? Default is `TRUE`.

Details

The graph represents each feature as a horizontal bar of length proportional to the importance of a feature. Features are sorted by decreasing importance. It works for both "gblinear" and "gbtree" models.

When `rel_to_first = FALSE`, the values would be plotted as in `importance_matrix`. For a "gbtree" model, that would mean being normalized to the total of 1 ("what is feature's importance contribution relative to the whole model?"). For linear models, `rel_to_first = FALSE` would show actual values of the coefficients. Setting `rel_to_first = TRUE` allows to see the picture from the perspective of "what is feature's importance contribution relative to the most important feature?"

The "ggplot" backend performs 1-D clustering of the importance values, with bar colors corresponding to different clusters having similar importance values.

Value

The return value depends on the function:

- `xgb.plot.importance()`: Invisibly, a "data.table" with `n_top` features sorted by importance. If `plot = TRUE`, the values are also plotted as barplot.
- `xgb.ggplot.importance()`: A customizable "ggplot" object. E.g., to change the title, set `+ ggtitle("A GRAPH NAME")`.

See Also

`graphics::barplot()`

Examples

```
data(agaricus.train)

## Keep the number of threads to 2 for examples
nthread <- 2
data.table::setDTthreads(nthread)

model <- xgboost(
  agaricus.train$data, factor(agaricus.train$label),
  nrounds = 2,
  max_depth = 3,
  nthreads = nthread
)

importance_matrix <- xgb.importance(model)
xgb.plot.importance(
  importance_matrix, rel_to_first = TRUE, xlab = "Relative importance"
)

gg <- xgb.ggplot.importance(
  importance_matrix, measure = "Frequency", rel_to_first = TRUE
)
gg
gg + ggplot2::ylab("Frequency")
```

`xgb.ggplot.shap.summary`
SHAP summary plot

Description

Visualizes SHAP contributions of different features.

Usage

```
xgb.ggplot.shap.summary(
  data,
  shap_contrib = NULL,
  features = NULL,
  top_n = 10,
  model = NULL,
  trees = NULL,
  target_class = NULL,
  approxcontrib = FALSE,
  subsample = NULL
)

xgb.plot.shap.summary(
  data,
  shap_contrib = NULL,
  features = NULL,
  top_n = 10,
  model = NULL,
  trees = NULL,
  target_class = NULL,
  approxcontrib = FALSE,
  subsample = NULL
)
```

Arguments

`data`	The data to explain as a `matrix`, `dgCMatrix`, or `data.frame`.
`shap_contrib`	Matrix of SHAP contributions of `data`. The default (`NULL`) computes it from `model` and `data`.
`features`	Vector of column indices or feature names to plot. When `NULL` (default), the `top_n` most important features are selected by `xgb.importance()`.
`top_n`	How many of the most important features (<= 100) should be selected? By default 1 for SHAP dependence and 10 for SHAP summary. Only used when `features = NULL`.
`model`	An `xgb.Booster` model. Only required when `shap_contrib = NULL` or `features = NULL`.
`trees`	Passed to `xgb.importance()` when `features = NULL`.
`target_class`	Only relevant for multiclass models. The default (`NULL`) averages the SHAP values over all classes. Pass a (0-based) class index to show only SHAP values of that class.
`approxcontrib`	Passed to `predict.xgb.Booster()` when `shap_contrib = NULL`.
`subsample`	Fraction of data points randomly picked for plotting. The default (`NULL`) will use up to 100k data points.

Details

A point plot (each point representing one observation from `data`) is produced for each feature, with the points plotted on the SHAP value axis. Each point (observation) is coloured based on its feature value.

The plot allows to see which features have a negative / positive contribution on the model prediction, and whether the contribution is different for larger or smaller values of the feature. Inspired by the summary plot of `https://github.com/shap/shap`.

Value

A `ggplot2` object.

See Also

`xgb.plot.shap()`, `xgb.ggplot.shap.summary()`, and the Python library `https://github.com/shap/shap`.

Examples

```
# See examples in xgb.plot.shap()
```

`xgb.importance` *Feature importance*

Description

Creates a `data.table` of feature importances.

Usage

```
xgb.importance(
  model = NULL,
  feature_names = getinfo(model, "feature_name"),
  trees = NULL
)
```

Arguments

`model`	Object of class `xgb.Booster`.
`feature_names`	Character vector used to overwrite the feature names of the model. The default is `NULL` (use original feature names).
`trees`	An integer vector of (base-1) tree indices that should be included into the importance calculation (only for the "gbtree" booster). The default (`NULL`) parses all trees. It could be useful, e.g., in multiclass classification to get feature importances for each class separately.

Details

This function works for both linear and tree models.

For linear models, the importance is the absolute magnitude of linear coefficients. To obtain a meaningful ranking by importance for linear models, the features need to be on the same scale (which is also recommended when using L1 or L2 regularization).

Value

A `data.table` with the following columns:

For a tree model:

- `Features`: Names of the features used in the model.
- `Gain`: Fractional contribution of each feature to the model based on the total gain of this feature's splits. Higher percentage means higher importance.
- `Cover`: Metric of the number of observation related to this feature.
- `Frequency`: Percentage of times a feature has been used in trees.

For a linear model:

- `Features`: Names of the features used in the model.
- `Weight`: Linear coefficient of this feature.
- `Class`: Class label (only for multiclass models). For objects of class `xgboost` (as produced by `xgboost()`), it will be a `factor`, while for objects of class `xgb.Booster` (as produced by `xgb.train()`), it will be a zero-based integer vector.

If `feature_names` is not provided and `model` doesn't have `feature_names`, the index of the features will be used instead. Because the index is extracted from the model dump (based on C++ code), it starts at 0 (as in C/C++ or Python) instead of 1 (usual in R).

Examples

```
# binary classification using "gbtree":
data("ToothGrowth")
x <- ToothGrowth[, c("len", "dose")]
y <- ToothGrowth$supp
model_tree_binary <- xgboost(
  x, y,
  nrounds = 5L,
  nthreads = 1L,
  booster = "gbtree",
  max_depth = 2L
)
xgb.importance(model_tree_binary)

# binary classification using "gblinear":
model_tree_linear <- xgboost(
  x, y,
  nrounds = 5L,
  nthreads = 1L,
  booster = "gblinear",
  learning_rate = 0.3
)
xgb.importance(model_tree_linear)

# multi-class classification using "gbtree":
data("iris")
x <- iris[, c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")]
y <- iris$Species
model_tree_multi <- xgboost(
  x, y,
  nrounds = 5L,
  nthreads = 1L,
  booster = "gbtree",
  max_depth = 3
)
# all classes clumped together:
xgb.importance(model_tree_multi)
# inspect importances separately for each class:
num_classes <- 3L
nrounds <- 5L
xgb.importance(
  model_tree_multi, trees = seq(from = 1, by = num_classes, length.out = nrounds)
)
xgb.importance(
  model_tree_multi, trees = seq(from = 2, by = num_classes, length.out = nrounds)
)
xgb.importance(
  model_tree_multi, trees = seq(from = 3, by = num_classes, length.out = nrounds)
)

# multi-class classification using "gblinear":
model_linear_multi <- xgboost(
  x, y,
  nrounds = 5L,
  nthreads = 1L,
  booster = "gblinear",
  learning_rate = 0.2
)
xgb.importance(model_linear_multi)
```

`xgb.is.same.Booster` *Check if two boosters share the same C object*

Description

Checks whether two booster objects refer to the same underlying C object.

Usage

```
xgb.is.same.Booster(obj1, obj2)
```

Arguments

`obj1`	Booster model to compare with `obj2`.
`obj2`	Booster model to compare with `obj1`.

Details

As booster objects (as returned by e.g. `xgb.train()`) contain an R 'externalptr' object, they don't follow typical copy-on-write semantics of other R objects - that is, if one assigns a booster to a different variable and modifies that new variable through in-place methods like `xgb.attr<-()`, the modification will be applied to both the old and the new variable, unlike typical R assignments which would only modify the latter.

This function allows checking whether two booster objects share the same 'externalptr', regardless of the R attributes that they might have.

In order to duplicate a booster in such a way that the copy wouldn't share the same 'externalptr', one can use function `xgb.copy.Booster()`.

Value

Either `TRUE` or `FALSE` according to whether the two boosters share the underlying C object.

See Also

`xgb.copy.Booster()`

Examples

```
library(xgboost)

data(mtcars)

y <- mtcars$mpg
x <- as.matrix(mtcars[, -1])

model <- xgb.train(
  params = xgb.params(nthread = 1),
  data = xgb.DMatrix(x, label = y, nthread = 1),
  nrounds = 3
)

model_shallow_copy <- model
xgb.is.same.Booster(model, model_shallow_copy) # same C object

model_deep_copy <- xgb.copy.Booster(model)
xgb.is.same.Booster(model, model_deep_copy) # different C objects

# In-place assignments modify all references,
# but not full/deep copies of the booster
xgb.attr(model_shallow_copy, "my_attr") <- 111
xgb.attr(model, "my_attr") # gets modified
xgb.attr(model_deep_copy, "my_attr") # doesn't get modified
```

xgb.load *Load XGBoost model from binary file*

Description

Load XGBoost model from binary model file.

Usage

```
xgb.load(modelfile)
```

Arguments

`modelfile`	The name of the binary input file.

Details

The input file is expected to contain a model saved in an XGBoost model format using either `xgb.save()` in R, or using some appropriate methods from other XGBoost interfaces. E.g., a model trained in Python and saved from there in XGBoost format, could be loaded from R.

Note: a model saved as an R object has to be loaded using corresponding R-methods, not by `xgb.load()`.

Value

An object of `xgb.Booster` class.

See Also

`xgb.save()`

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)

train <- agaricus.train
test <- agaricus.test

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

fname <- file.path(tempdir(), "xgb.ubj")
xgb.save(bst, fname)
bst <- xgb.load(fname)
```

`xgb.load.raw` *Load serialised XGBoost model from R's raw vector*

Description

User can generate raw memory buffer by calling `xgb.save.raw()`.

Usage

```
xgb.load.raw(buffer)
```

Arguments

`buffer`	The buffer returned by `xgb.save.raw()`.

`xgb.model.dt.tree` *Parse model text dump*

Description

Parse a boosted tree model text dump into a `data.table` structure.

Usage

```
xgb.model.dt.tree(model, trees = NULL, use_int_id = FALSE, ...)
```

Arguments

`model`	Object of class `xgb.Booster`. If it contains feature names (they can be set through `setinfo()`), they will be used in the output from this function. If the model contains categorical features, an error will be thrown.
`trees`	An integer vector of (base-1) tree indices that should be used. The default (`NULL`) uses all trees. Useful, e.g., in multiclass classification to get only the trees of one class.
`use_int_id`	A logical flag indicating whether nodes in columns "Yes", "No", and "Missing" should be represented as integers (when `TRUE`) or as "Tree-Node" character strings (when `FALSE`, default).

`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

Note that this function does not work with models that were fitted to categorical data, and is only applicable to tree-based boosters (not `gblinear`).

Value

A `data.table` with detailed information about tree nodes. It has the following columns:

- `Tree`: integer ID of a tree in a model (zero-based index).
- `Node`: integer ID of a node in a tree (zero-based index).
- `ID`: character identifier of a node in a model (only when `use_int_id = FALSE`).
- `Feature`: for a branch node, a feature ID or name (when available); for a leaf node, it simply labels it as `"Leaf"`.
- `Split`: location of the split for a branch node (split condition is always "less than").
- `Yes`: ID of the next node when the split condition is met.
- `No`: ID of the next node when the split condition is not met.
- `Missing`: ID of the next node when the branch value is missing.
- `Gain`: either the split gain (change in loss) or the leaf value.
- `Cover`: metric related to the number of observations either seen by a split or collected by a leaf during training.

When `use_int_id = FALSE`, columns "Yes", "No", and "Missing" point to model-wide node identifiers in the "ID" column. When `use_int_id = TRUE`, those columns point to node identifiers from the corresponding trees in the "Node" column.

Examples

```
# Basic use:

data(agaricus.train, package = "xgboost")
## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)
```

```
bst <- xgb.train(
  data = xgb.DMatrix(agaricus.train$data, label = agaricus.train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

# This bst model already has feature_names stored with it, so those would be used when
# feature_names is not set:
dt <- xgb.model.dt.tree(bst)

# How to match feature names of splits that are following a current 'Yes' branch:
merge(
  dt,
  dt[, .(ID, Y.Feature = Feature)], by.x = "Yes", by.y = "ID", all.x = TRUE
)[
  order(Tree, Node)
]
```

xgb.model.parameters<- *Accessors for model parameters*

Description

Only the setter for XGBoost parameters is currently implemented.

Usage

```
xgb.model.parameters(object) <- value
```

Arguments

object	Object of class `xgb.Booster`. **Will be modified in-place**.
value	A list (or an object coercible to a list) with the names of parameters to set and the elements corresponding to parameter values.

Details

Just like `xgb.attr()`, this function will make in-place modifications on the booster object which do not follow typical R assignment semantics - that is, all references to the same booster will also be updated, unlike assingment of R attributes which follow copy-on-write semantics.

See `xgb.copy.Booster()` for an example of this behavior.

Be aware that setting parameters of a fitted booster related to training continuation / updates will reset its number of rounds indicator to zero.

Value

The same booster `object`, which gets modified in-place.

Examples

```
data(agaricus.train, package = "xgboost")

train <- agaricus.train

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    learning_rate = 1,
    nthread = 2,
    objective = "binary:logistic"
  )
)

xgb.model.parameters(bst) <- list(learning_rate = 0.1)
```

`xgb.params` *XGBoost Parameters*

Description

Convenience function to generate a list of named XGBoost parameters, which can be passed as argument `params` to `xgb.train()`. See the online documentation for more details.

The purpose of this function is to enable IDE autocompletions and to provide in-package documentation for all the possible parameters that XGBoost accepts. The output from this function is just a regular R list containing the parameters that were set to non-default values. Note that this function will not perform any validation on the supplied arguments.

If passing `NULL` for a given parameter (the default for all of them), then the default value for that parameter will be used. Default values are automatically determined by the XGBoost core library upon calls to `xgb.train()` or `xgb.cv()`, and are subject to change over XGBoost library versions. Some of them might differ according to the booster type (e.g. defaults for regularization are different for linear and tree-based boosters).

Usage

```
xgb.params(
  objective = NULL,
  verbosity = NULL,
  nthread = NULL,
  seed = NULL,
  booster = NULL,
  eta = NULL,
  learning_rate = NULL,
  gamma = NULL,
  min_split_loss = NULL,
  max_depth = NULL,
  min_child_weight = NULL,
  max_delta_step = NULL,
  subsample = NULL,
  sampling_method = NULL,
  colsample_bytree = NULL,
  colsample_bylevel = NULL,
  colsample_bynode = NULL,
  lambda = NULL,
  reg_lambda = NULL,
  alpha = NULL,
  reg_alpha = NULL,
  tree_method = NULL,
  scale_pos_weight = NULL,
  updater = NULL,
  refresh_leaf = NULL,
  grow_policy = NULL,
  max_leaves = NULL,
  max_bin = NULL,
  num_parallel_tree = NULL,
  monotone_constraints = NULL,
  interaction_constraints = NULL,
  multi_strategy = NULL,
  base_score = NULL,
  eval_metric = NULL,
  seed_per_iteration = NULL,
  device = NULL,
  disable_default_eval_metric = NULL,
  use_rmm = NULL,
  max_cached_hist_node = NULL,
  max_cat_to_onehot = NULL,
  max_cat_threshold = NULL,
  sample_type = NULL,
  normalize_type = NULL,
  rate_drop = NULL,
  one_drop = NULL,
  skip_drop = NULL,
  feature_selector = NULL,
  top_k = NULL,
  num_class = NULL,
  tweedie_variance_power = NULL,
  huber_slope = NULL,
  quantile_alpha = NULL,
  aft_loss_distribution = NULL,
  aft_loss_distribution_scale = NULL,
  lambdarank_pair_method = NULL,
  lambdarank_num_pair_per_sample = NULL,
  lambdarank_normalization = NULL,
  lambdarank_score_normalization = NULL,
  lambdarank_unbiased = NULL,
  lambdarank_bias_norm = NULL,
  ndcg_exp_gain = NULL
)
```

Arguments

`objective` (default=`"reg:squarederror"`) Specify the learning task and the corresponding learning objective or a custom objective function to be used.

For custom objective, see Custom Objective and Evaluation Metric and Custom objective and metric for more information, along with the end note for function signatures.

Supported values are:

- `"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, output probability
- `"reg:pseudohubererror"`: regression with Pseudo Huber loss, a twice differentiable alternative to absolute loss.
- `"reg:absoluteerror"`: Regression with L1 error. When tree model is used, leaf value is refreshed after tree construction. If used in distributed training, the leaf value is calculated as the mean value from all workers, which is not guaranteed to be optimal.
  Version added: 1.7.0
- `"reg:quantileerror"`: Quantile loss, also known as "pinball loss". See later sections for its parameter and Quantile Regression for a worked example.
  Version added: 2.0.0
- `"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.
- `"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:ndcg"`: Use LambdaMART to perform pair-wise ranking where the normalized discounted cumulative gain (NDCG) is maximized. This objective supports position debiasing for click data.
- `"rank:map"`: Use LambdaMART to perform pair-wise ranking where the mean average precision (MAP) is maximized
- `"rank:pairwise"`: Use LambdaRank to perform pair-wise ranking using the `ranknet` objective.
- `"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.

`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.

`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.

`seed` Random number seed. If not specified, will take a random seed through R's own RNG engine.

`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.

`eta`, `learning_rate`

(two aliases for the same parameter) Step size shrinkage used in update to prevent 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: $[0, 1]$
- default value: 0.3 for tree-based boosters, 0.5 for linear booster.

Note: should only pass one of `eta` or `learning_rate`. Both refer to the same parameter and there's thus no difference between one or the other.

`gamma`, `min_split_loss`
: (two aliases for the same parameter) (for Tree Booster) (default=0, alias: `gamma`) Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `min_split_loss` is, the more conservative the algorithm will be. Note that a tree where no splits were made might still contain a single terminal node with a non-zero score.

  range: $[0, \infty)$

  Note: should only pass one of `gamma` or `min_split_loss`. Both refer to the same parameter and there's thus no difference between one or the other.

`max_depth`
: (for Tree Booster) (default=6, type=int32) 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: $[0, \infty)$

`min_child_weight`
: (for Tree Booster) (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: $[0, \infty)$

`max_delta_step`
: (for Tree Booster) (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: $[0, \infty)$

`subsample`
: (for Tree Booster) (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`
: (for Tree Booster) (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 `"hist"`; other tree methods only support `"uniform"` sampling.

`colsample_bytree`, `colsample_bylevel`, `colsample_bynode`
: (for Tree Booster) (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. This is not supported by the exact tree method.
- `"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.

One can set the `"feature_weights"` for DMatrix to define the probability of each feature being selected when using column sampling.

`lambda`, `reg_lambda`

(two aliases for the same parameter)

- For tree-based boosters:
  - L2 regularization term on weights. Increasing this value will make model more conservative.
  - default: 1
  - range: $[0, \infty]$
- For linear booster:
  - L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
  - default: 0
  - range: $[0, \infty)$

Note: should only pass one of `lambda` or `reg_lambda`. Both refer to the same parameter and there's thus no difference between one or the other.

`alpha`, `reg_alpha`

(two aliases for the same parameter)

- L1 regularization term on weights. Increasing this value will make model more conservative.
- For the linear booster, it's normalised to number of training examples.
- default: 0
- range: $[0, \infty)$

Note: should only pass one of `alpha` or `reg_alpha`. Both refer to the same parameter and there's thus no difference between one or the other.

`tree_method` (for Tree Booster) (default= `"auto"`) The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.

Choices: `"auto"`, `"exact"`, `"approx"`, `"hist"`, this is a combination of commonly used updaters. For other updaters like `"refresh"`, set the parameter `updater` directly.

- `"auto"`: Same as the `"hist"` tree method.
- `"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.

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

`updater`
: Has different meanings depending on the type of booster.

- For tree-based boosters: 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_quantile_histmaker"`: Grow tree using quantized histogram.
  - `"grow_gpu_hist"`: Enabled when `tree_method` is set to `"hist"` along with `device="cuda"`.
  - `"grow_gpu_approx"`: Enabled when `tree_method` is set to `"approx"` along with `device="cuda"`.
  - `"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`.
- For `booster="gblinear"`: (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. When the `device` parameter is set to `"cuda"` or `"gpu"`, a GPU variant would be used.

`refresh_leaf`
: (for Tree Booster) (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.

`grow_policy`
: (for Tree Booster) (default= `"depthwise"`)

- Controls a way new nodes are added to the tree.
- Currently supported only if `tree_method` is set to `"hist"` or `"approx"`.

- Choices: `"depthwise"`, `"lossguide"`
  - `"depthwise"`: split at nodes closest to the root.
  - `"lossguide"`: split at nodes with highest loss change.

`max_leaves` (for Tree Booster) (default=0, type=int32) Maximum number of nodes to be added. Not used by `"exact"` tree method.

`max_bin` (for Tree Booster) (default=256, type=int32)

- Only used if `tree_method` is set to `"hist"` or `"approx"`.
- Maximum number of discrete bins to bucket continuous features.
- Increasing this number improves the optimality of splits at the cost of higher computation time.

`num_parallel_tree`

(for Tree Booster) (default=1) Number of parallel trees constructed during each iteration. This option is used to support boosted random forest.

`monotone_constraints`

(for Tree Booster) Constraint of variable monotonicity. See Monotonic Constraints for more information.

`interaction_constraints`

(for Tree Booster) Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nest list, e.g. `list(c(0, 1), c(2, 3, 4))`, where each inner list is a group of indices of features (base-0 numeration) that are allowed to interact with each other. See Feature Interaction Constraints for more information.

`multi_strategy` (for Tree Booster) (default = `"one_output_per_tree"`) The strategy used for training multi-target models, including multi-target regression and multi-class classification. See Multiple Outputs for more information.

- `"one_output_per_tree"`: One model for each target.
- `"multi_output_tree"`: Use multi-target trees.

Version added: 2.0.0

Note: This parameter is working-in-progress.

`base_score`

- The initial prediction score of all instances, global bias
- The parameter is automatically estimated for selected objectives before training. To disable the estimation, specify a real number argument.
- If `base_margin` is supplied, `base_score` will not be added.
- 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 `rank:map`, etc.)
- User can add multiple evaluation metrics.
- 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"`: area under the receiver-operating characteristic 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.

- `"pre"`: Precision at k. Supports only learning to rank task.
- `"ndcg"`: normalized discounted cumulative gain
- `"map"`: mean average precision
  The `average precision` is defined as:
  $AP@l = \frac{1}{min(l,N)} \sum_{k=1}^{l} P@k \cdot I_{(k)}$
  where $I_{(k)}$ is an indicator function that equals to 1 when the document at k is relevant and 0 otherwise. The $P@k$ is the precision at k, and N is the total number of relevant documents. Lastly, the `mean average precision` is defined as the weighted average across all queries.
- `"ndcg@n"`, `"map@n"`, `"pre@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, the NDCG and MAP evaluate the score of a list without any positive samples as 1. By appending "-" to the evaluation metric name, we can ask XGBoost to evaluate these scores 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_per_iteration`
(default= `FALSE`) Seed PRNG determnisticly via iterator number.

`device` (default= `"cpu"`) Device for XGBoost to run. User can set it to one of the following values:

- `"cpu"`: Use CPU.
- `"cuda"`: Use a GPU (CUDA device).
- `"cuda:<ordinal>"`: `<ordinal>` is an integer that specifies the ordinal of the GPU (which GPU do you want to use if you have more than one devices).
- `"gpu"`: Default GPU device selection from the list of available and supported devices. Only `"cuda"` devices are supported currently.
- `"gpu:<ordinal>"`: Default GPU device selection from the list of available and supported devices. Only `"cuda"` devices are supported currently.

For more information about GPU acceleration, see XGBoost GPU Support. In distributed environments, ordinal selection is handled by distributed frameworks instead of XGBoost. As a result, using `"cuda:<ordinal>"` will result in an error. Use `"cuda"` instead.

Version added: 2.0.0

Note: if XGBoost was installed from CRAN, it won't have GPU support enabled, thus only `"cpu"` will be available. To get GPU support, the R package for XGBoost must be installed from source or from the GitHub releases - see instructions.

`disable_default_eval_metric`
(default= `FALSE`) Flag to disable default metric. Set to 1 or `TRUE` to disable.

`use_rmm` Whether to use RAPIDS Memory Manager (RMM) to allocate cache GPU memory. The primary memory is always allocated on the RMM pool when XGBoost is built (compiled) with the RMM plugin enabled. Valid values are `TRUE` and `FALSE`. See Using XGBoost with RAPIDS Memory Manager (RMM) plugin for details.

`max_cached_hist_node`
(for Non-Exact Tree Methods) (default = 65536) Maximum number of cached nodes for histogram. This can be used with the `"hist"` and the `"approx"` tree methods.

Version added: 2.0.0

- For most of the cases this parameter should not be set except for growing deep trees. After 3.0, this parameter affects GPU algorithms as well.

`max_cat_to_onehot`
(for Non-Exact Tree Methods) 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.

Version added: 1.6.0

`max_cat_threshold`
(for Non-Exact Tree Methods) Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting.

Version added: 1.7.0

`sample_type` (for Dart Booster) (default= `"uniform"`) Type of sampling algorithm.

- `"uniform"`: dropped trees are selected uniformly.
- `"weighted"`: dropped trees are selected in proportion to weight.

`normalize_type` (for Dart Booster) (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` (for Dart Booster) (default=0.0) Dropout rate (a fraction of previous trees to drop during the dropout).

range: $[0.0, 1.0]$

`one_drop` (for Dart Booster) (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` (for Dart Booster) (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: $[0.0, 1.0]$

`feature_selector`

(for Linear Booster) (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` (for Linear Booster) (default=0) The number of top features to select in `greedy` and `thrifty` feature selector. The value of 0 means using all the features.

`num_class` Number of classes when using multi-class classification objectives (e.g. `objective="multi:softprob"`)

`tweedie_variance_power`

(for Tweedie Regression (`"objective=reg:tweedie"`)) (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.

`huber_slope` (for using Pseudo-Huber (`"reg:pseudohubererror"`)) (default = 1.0) A parameter used for Pseudo-Huber loss to define the δ term.

`quantile_alpha` (for using Quantile Loss (`"reg:quantileerror"`)) A scalar or a list of targeted quantiles (passed as a numeric vector).

Version added: 2.0.0

`aft_loss_distribution`

(when using AFT Survival Loss (`"survival:aft"`) and Negative Log Likelihood of AFT metric (`"aft-nloglik"`)) Probability Density Function, `"normal"`, `"logistic"`, or `"extreme"`.

`aft_loss_distribution_scale`
: (when using AFT Survival Loss (`"survival:aft"`) and Negative Log Likelihood of AFT metric (`"aft-nloglik"`)) Scaling factor for the AFT distribution. Range: $(0, \infty)$.

`lambdarank_pair_method`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) (default = `"topk"`) How to construct pairs for pair-wise learning.
  - `"mean"`: Sample `lambdarank_num_pair_per_sample` pairs for each document in the query list.
  - `"topk"`: Focus on top-`lambdarank_num_pair_per_sample` documents. Construct $|query|$ pairs for each document at the top-`lambdarank_num_pair_per_sample` ranked by the model.

`lambdarank_num_pair_per_sample`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) It specifies the number of pairs sampled for each document when pair method is `"mean"`, or the truncation level for queries when the pair method is `"topk"`. For example, to train with `ndcg@6`, set `"lambdarank_num_pair_per_sample"` to 6 and `lambdarank_pair_method` to `"topk"`.
  range = $[1, \infty)$

`lambdarank_normalization`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) (default = `TRUE`) Whether to normalize the leaf value by lambda gradient. This can sometimes stagnate the training progress.
  Version added: 2.1.0

`lambdarank_score_normalization`
: Whether to normalize the delta metric by the difference of prediction scores. This can sometimes stagnate the training progress. With pairwise ranking, we can normalize the gradient using the difference between two samples in each pair to reduce influence from the pairs that have large difference in ranking scores. This can help us regularize the model to reduce bias and prevent overfitting. Similar to other regularization techniques, this might prevent training from converging.
  There was no normalization before 2.0. In 2.0 and later versions this is used by default. In 3.0, we made this an option that users can disable.
  Version added: 3.0.0

`lambdarank_unbiased`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) (default = `FALSE`) Specify whether do we need to debias input click data.

`lambdarank_bias_norm`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) (default = 2.0) L_p normalization for position debiasing, default is L_2. Only relevant when `lambdarank_unbiased` is set to `TRUE`.

`ndcg_exp_gain`
: (for learning to rank (`"rank:ndcg"`, `"rank:map"`, `"rank:pairwise"`)) (default = `TRUE`) Whether we should use exponential gain function for `NDCG`. There are two forms of gain function for `NDCG`, one is using relevance value directly while the other is using$2^{rel} - 1$ to emphasize on retrieving relevant documents. When `ndcg_exp_gain` is `TRUE` (the default), relevance degree cannot be greater than 31.

Value

A list with the entries that were passed non-NULL values. It is intended to be passed as argument `params` to `xgb.train()` or `xgb.cv()`.

`xgb.plot.multi.trees` *Project all trees on one tree*

Description

Visualization of the ensemble of trees as a single collective unit.

Usage

```
xgb.plot.multi.trees(
  model,
  features_keep = 5,
  plot_width = NULL,
  plot_height = NULL,
  render = TRUE,
  ...
)
```

Arguments

`model`	Object of class `xgb.Booster`. If it contains feature names (they can be set through `setinfo()`, they will be used in the output from this function.
`features_keep`	Number of features to keep in each position of the multi trees, by default 5.
`plot_width, plot_height`	Width and height of the graph in pixels. The values are passed to `DiagrammeR::render_graph()`.
`render`	Should the graph be rendered or not? The default is `TRUE`.
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

Note that this function does not work with models that were fitted to categorical data.

This function tries to capture the complexity of a gradient boosted tree model in a cohesive way by compressing an ensemble of trees into a single tree-graph representation. The goal is to improve the interpretability of a model generally seen as black box.

Note: this function is applicable to tree booster-based models only.

It takes advantage of the fact that the shape of a binary tree is only defined by its depth (therefore, in a boosting model, all trees have similar shape).

Moreover, the trees tend to reuse the same features.

The function projects each tree onto one, and keeps for each position the `features_keep` first features (based on the Gain per feature measure).

This function is inspired by this blog post: https://wellecks.wordpress.com/2015/02/21/peering-into-the-black-box-visualizing-lambdamart/

Value

Rendered graph object which is an htmlwidget of ' class `grViz`. Similar to "ggplot" objects, it needs to be printed when not running from the command line.

Examples

```
data(agaricus.train, package = "xgboost")

## Keep the number of threads to 2 for examples
nthread <- 2
data.table::setDTthreads(nthread)

model <- xgboost(
  agaricus.train$data, factor(agaricus.train$label),
  nrounds = 30,
  verbosity = 0L,
  nthreads = nthread,
  max_depth = 15,
  learning_rate = 1,
  min_child_weight = 50
)

p <- xgb.plot.multi.trees(model, features_keep = 3)
print(p)

# Below is an example of how to save this plot to a file.
if (require("DiagrammeR") && require("DiagrammeRsvg") && require("rsvg")) {
  fname <- file.path(tempdir(), "tree.pdf")
  gr <- xgb.plot.multi.trees(model, features_keep = 3, render = FALSE)
  export_graph(gr, fname, width = 1500, height = 600)
}
```

xgb.plot.shap *SHAP dependence plots*

Description

Visualizes SHAP values against feature values to gain an impression of feature effects.

Usage

```
xgb.plot.shap(
  data,
  shap_contrib = NULL,
  features = NULL,
  top_n = 1,
  model = NULL,
  trees = NULL,
  target_class = NULL,
  approxcontrib = FALSE,
  subsample = NULL,
  n_col = 1,
  col = rgb(0, 0, 1, 0.2),
  pch = ".",
  discrete_n_uniq = 5,
  discrete_jitter = 0.01,
  ylab = "SHAP",
  plot_NA = TRUE,
  col_NA = rgb(0.7, 0, 1, 0.6),
  pch_NA = ".",
  pos_NA = 1.07,
  plot_loess = TRUE,
  col_loess = 2,
  span_loess = 0.5,
  which = c("1d", "2d"),
  plot = TRUE,
  ...
)
```

Arguments

`data`	The data to explain as a `matrix`, `dgCMatrix`, or `data.frame`.
`shap_contrib`	Matrix of SHAP contributions of `data`. The default (`NULL`) computes it from `model` and `data`.
`features`	Vector of column indices or feature names to plot. When `NULL` (default), the `top_n` most important features are selected by `xgb.importance()`.
`top_n`	How many of the most important features (<= 100) should be selected? By default 1 for SHAP dependence and 10 for SHAP summary. Only used when `features = NULL`.

`model`	An `xgb.Booster` model. Only required when `shap_contrib = NULL` or `features = NULL`.
`trees`	Passed to `xgb.importance()` when `features = NULL`.
`target_class`	Only relevant for multiclass models. The default (`NULL`) averages the SHAP values over all classes. Pass a (0-based) class index to show only SHAP values of that class.
`approxcontrib`	Passed to `predict.xgb.Booster()` when `shap_contrib = NULL`.
`subsample`	Fraction of data points randomly picked for plotting. The default (`NULL`) will use up to 100k data points.
`n_col`	Number of columns in a grid of plots.
`col`	Color of the scatterplot markers.
`pch`	Scatterplot marker.
`discrete_n_uniq`	Maximal number of unique feature values to consider the feature as discrete.
`discrete_jitter`	Jitter amount added to the values of discrete features.
`ylab`	The y-axis label in 1D plots.
`plot_NA`	Should contributions of cases with missing values be plotted? Default is `TRUE`.
`col_NA`	Color of marker for missing value contributions.
`pch_NA`	Marker type for `NA` values.
`pos_NA`	Relative position of the x-location where `NA` values are shown: `min(x) + (max(x) - min(x)) * pos_NA`.
`plot_loess`	Should loess-smoothed curves be plotted? (Default is `TRUE`). The smoothing is only done for features with more than 5 distinct values.
`col_loess`	Color of loess curves.
`span_loess`	The span parameter of `stats::loess()`.
`which`	Whether to do univariate or bivariate plotting. Currently, only "1d" is implemented.
`plot`	Should the plot be drawn? (Default is `TRUE`). If `FALSE`, only a list of matrices is returned.
`...`	Other parameters passed to `graphics::plot()`.

Details

These scatterplots represent how SHAP feature contributions depend of feature values. The similarity to partial dependence plots is that they also give an idea for how feature values affect predictions. However, in partial dependence plots, we see marginal dependencies of model prediction on feature value, while SHAP dependence plots display the estimated contributions of a feature to the prediction for each individual case.

When `plot_loess = TRUE`, feature values are rounded to three significant digits and weighted LOESS is computed and plotted, where the weights are the numbers of data points at each rounded value.

Note: SHAP contributions are on the scale of the model margin. E.g., for a logistic binomial objective, the margin is on log-odds scale. Also, since SHAP stands for "SHapley Additive exPlanation" (model prediction = sum of SHAP contributions for all features + bias), depending on the objective used, transforming SHAP contributions for a feature from the marginal to the prediction space is not necessarily a meaningful thing to do.

Value

In addition to producing plots (when `plot = TRUE`), it silently returns a list of two matrices:

- `data`: Feature value matrix.
- `shap_contrib`: Corresponding SHAP value matrix.

References

1. Scott M. Lundberg, Su-In Lee, "A Unified Approach to Interpreting Model Predictions", NIPS Proceedings 2017, https://arxiv.org/abs/1705.07874
2. Scott M. Lundberg, Su-In Lee, "Consistent feature attribution for tree ensembles", https://arxiv.org/abs/1706.06060

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)
nrounds <- 20

model_binary <- xgboost(
  agaricus.train$data, factor(agaricus.train$label),
  nrounds = nrounds,
  verbosity = 0L,
  learning_rate = 0.1,
  max_depth = 3L,
  subsample = 0.5,
  nthreads = nthread
)

xgb.plot.shap(agaricus.test$data, model = model_binary, features = "odor=none")

contr <- predict(model_binary, agaricus.test$data, type = "contrib")
xgb.plot.shap(agaricus.test$data, contr, model = model_binary, top_n = 12, n_col = 3)

# Summary plot
xgb.ggplot.shap.summary(agaricus.test$data, contr, model = model_binary, top_n = 12)

# Multiclass example - plots for each class separately:
x <- as.matrix(iris[, -5])
set.seed(123)
```

```
is.na(x[sample(nrow(x) * 4, 30)]) <- TRUE # introduce some missing values

model_multiclass <- xgboost(
  x, iris$Species,
  nrounds = nrounds,
  verbosity = 0,
  max_depth = 2,
  subsample = 0.5,
  nthreads = nthread
)
nclass <- 3
trees0 <- seq(from = 1, by = nclass, length.out = nrounds)
col <- rgb(0, 0, 1, 0.5)

xgb.plot.shap(
  x,
  model = model_multiclass,
  trees = trees0,
  target_class = 0,
  top_n = 4,
  n_col = 2,
  col = col,
  pch = 16,
  pch_NA = 17
)

xgb.plot.shap(
  x,
  model = model_multiclass,
  trees = trees0 + 1,
  target_class = 1,
  top_n = 4,
  n_col = 2,
  col = col,
  pch = 16,
  pch_NA = 17
)

xgb.plot.shap(
  x,
  model = model_multiclass,
  trees = trees0 + 2,
  target_class = 2,
  top_n = 4,
  n_col = 2,
  col = col,
  pch = 16,
  pch_NA = 17
)

# Summary plot
xgb.ggplot.shap.summary(x, model = model_multiclass, target_class = 0, top_n = 4)
```

xgb.plot.tree *Plot boosted trees*

Description

Read a tree model text dump and plot the model.

Usage

```
xgb.plot.tree(
  model,
  tree_idx = 1,
  plot_width = NULL,
  plot_height = NULL,
  with_stats = FALSE,
  ...
)
```

Arguments

`model`	Object of class `xgb.Booster`. If it contains feature names (they can be set through `setinfo()`, they will be used in the output from this function.
`tree_idx`	An integer of the tree index that should be used. This is an 1-based index.
`plot_width, plot_height`	Width and height of the graph in pixels. The values are passed to `DiagrammeR::render_graph()`.
`with_stats`	Whether to dump some additional statistics about the splits. When this option is on, the model dump contains two additional values: gain is the approximate loss function gain we get in each split; cover is the sum of second order gradient in each node.
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** ... will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

The content of each node is visualized as follows:

- For non-terminal nodes, it will display the split condition (number or name if available, and the condition that would decide to which node to go next).
- Those nodes will be connected to their children by arrows that indicate whether the branch corresponds to the condition being met or not being met.
- Terminal (leaf) nodes contain the margin to add when ending there.

The "Yes" branches are marked by the "< split_value" label. The branches also used for missing values are marked as bold (as in "carrying extra capacity").

This function uses GraphViz as DiagrammeR backend.

Value

Rendered graph object which is an htmlwidget of ' class `grViz`. Similar to "ggplot" objects, it needs to be printed when not running from the command line.

Examples

```
data("ToothGrowth")
x <- ToothGrowth[, c("len", "dose")]
y <- ToothGrowth$supp
model <- xgboost(
  x, y,
  nthreads = 1L,
  nrounds = 3L,
  max_depth = 3L
)

# plot the first tree
xgb.plot.tree(model, tree_idx = 1)

# Below is an example of how to save this plot to a file.
if (require("DiagrammeR") && require("htmlwidgets")) {
  fname <- file.path(tempdir(), "plot.html'")
  gr <- xgb.plot.tree(model, tree_idx = 1)
  htmlwidgets::saveWidget(gr, fname)
}
```

`xgb.QuantileDMatrix.from_iterator` *QuantileDMatrix from External Data*

Description

Create an `xgb.QuantileDMatrix` object (exact same class as would be returned by calling function `xgb.QuantileDMatrix()`, with the same advantages and limitations) from external data supplied by `xgb.DataIter()`, potentially passed in batches from a bigger set that might not fit entirely in memory, same way as `xgb.ExtMemDMatrix()`.

Note that, while external data will only be loaded through the iterator (thus the full data might not be held entirely in-memory), the quantized representation of the data will get created in-memory, being concatenated from multiple calls to the data iterator. The quantized version is typically lighter than the original data, so there might be cases in which this representation could potentially fit in memory even if the full data does not.

For more information, see the guide 'Using XGBoost External Memory Version': `https://xgboost.readthedocs.io/en/stable/tutorials/external_memory.html`

Usage

```
xgb.QuantileDMatrix.from_iterator(
  data_iterator,
  missing = NA,
  nthread = NULL,
  ref = NULL,
  max_bin = NULL
)
```

Arguments

`data_iterator`	A data iterator structure as returned by `xgb.DataIter()`, which includes an environment shared between function calls, and functions to access the data in batches on-demand.
`missing`	A float value to represents missing values in data. Note that, while functions like `xgb.DMatrix()` can take a generic `NA` and interpret it correctly for different types like `numeric` and `integer`, if an `NA` value is passed here, it will not be adapted for different input types. For example, in R `integer` types, missing values are represented by integer number `-2147483648` (since machine 'integer' types do not have an inherent 'NA' value) - hence, if one passes `NA`, which is interpreted as a floating-point NaN by `xgb.ExtMemDMatrix()` and by `xgb.QuantileDMatrix.from_iterator()`, these integer missing values will not be treated as missing. This should not pose any problem for `numeric` types, since they do have an inheret NaN value.
`nthread`	Number of threads used for creating DMatrix.
`ref`	The training dataset that provides quantile information, needed when creating validation/test dataset with `xgb.QuantileDMatrix()`. Supplying the training DMatrix as a reference means that the same quantisation applied to the training data is applied to the validation/test data
`max_bin`	The number of histogram bin, should be consistent with the training parameter `max_bin`. This is only supported when constructing a QuantileDMatrix.

Value

An 'xgb.DMatrix' object, with subclass 'xgb.QuantileDMatrix'.

See Also

`xgb.DataIter()`, `xgb.DataBatch()`, `xgb.ExtMemDMatrix()`, `xgb.QuantileDMatrix()`

`xgb.save` *Save XGBoost model to binary file*

Description

Save XGBoost model to a file in binary or JSON format.

Usage

```
xgb.save(model, fname)
```

Arguments

`model`	Model object of `xgb.Booster` class.
`fname`	Name of the file to write. Its extension determines the serialization format: • ".ubj": Use the universal binary JSON format (recommended). This format uses binary types for e.g. floating point numbers, thereby preventing any loss of precision when converting to a human-readable JSON text or similar. • ".json": Use plain JSON, which is a human-readable format. • ".deprecated": Use **deprecated** binary format. This format will not be able to save attributes introduced after v1 of XGBoost, such as the "best_iteration" attribute that boosters might keep, nor feature names or user-specifiec attributes. • If the format is not specified by passing one of the file extensions above, will default to UBJ.

Details

This methods allows to save a model in an XGBoost-internal binary or text format which is universal among the various xgboost interfaces. In R, the saved model file could be read later using either the `xgb.load()` function or the `xgb_model` parameter of `xgb.train()`.

Note: a model can also be saved as an R object (e.g., by using `readRDS()` or `save()`). However, it would then only be compatible with R, and corresponding R methods would need to be used to load it. Moreover, persisting the model with `readRDS()` or `save()` might cause compatibility problems in future versions of XGBoost. Consult a-compatibility-note-for-saveRDS-save to learn how to persist models in a future-proof way, i.e., to make the model accessible in future releases of XGBoost.

See Also

`xgb.load()`

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)

train <- agaricus.train
test <- agaricus.test

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

fname <- file.path(tempdir(), "xgb.ubj")
xgb.save(bst, fname)
bst <- xgb.load(fname)
```

`xgb.save.raw` *Save XGBoost model to R's raw vector*

Description

Save XGBoost model from `xgboost()` or `xgb.train()`. Call `xgb.load.raw()` to load the model back from raw vector.

Usage

```
xgb.save.raw(model, raw_format = "ubj")
```

Arguments

`model`	The model object.
`raw_format`	The format for encoding the booster: • "json": Encode the booster into JSON text document. • "ubj": Encode the booster into Universal Binary JSON. • "deprecated": Encode the booster into old customized binary format.

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)

train <- agaricus.train
test <- agaricus.test

bst <- xgb.train(
  data = xgb.DMatrix(train$data, label = train$label, nthread = 1),
  nrounds = 2,
  params = xgb.params(
    max_depth = 2,
    nthread = nthread,
    objective = "binary:logistic"
  )
)

raw <- xgb.save.raw(bst)
bst <- xgb.load.raw(raw)
```

`xgb.set.config, xgb.get.config`
Set and get global configuration

Description

Global configuration consists of a collection of parameters that can be applied in the global scope. See `https://xgboost.readthedocs.io/en/stable/parameter.html` for the full list of parameters supported in the global configuration. Use `xgb.set.config()` to update the values of one or more global-scope parameters. Use `xgb.get.config()` to fetch the current values of all global-scope parameters (listed in `https://xgboost.readthedocs.io/en/stable/parameter.html`).

Usage

```
xgb.set.config(...)

xgb.get.config()
```

Arguments

`...`	List of parameters to be set, as keyword arguments

Details

Note that serialization-related functions might use a globally-configured number of threads, which is managed by the system's OpenMP (OMP) configuration instead. Typically, XGBoost methods accept an `nthreads` parameter, but some methods like `readRDS()` might get executed before such parameter can be supplied.

The number of OMP threads can in turn be configured for example through an environment variable `OMP_NUM_THREADS` (needs to be set before R is started), or through `RhpcBLASctl::omp_set_num_threads`.

Value

`xgb.set.config()` returns `TRUE` to signal success. `xgb.get.config()` returns a list containing all global-scope parameters and their values.

Examples

```
# Set verbosity level to silent (0)
xgb.set.config(verbosity = 0)
# Now global verbosity level is 0
config <- xgb.get.config()
print(config$verbosity)
# Set verbosity level to warning (1)
xgb.set.config(verbosity = 1)
# Now global verbosity level is 1
config <- xgb.get.config()
print(config$verbosity)
```

`xgb.slice.Booster` *Slice Booster by Rounds*

Description

Creates a new booster including only a selected range of rounds / iterations from an existing booster, as given by the sequence `seq(start, end, step)`.

Usage

```
xgb.slice.Booster(
  model,
  start,
  end = xgb.get.num.boosted.rounds(model),
  step = 1L
)

## S3 method for class 'xgb.Booster'
x[i]
```

Arguments

`model, x`	A fitted `xgb.Booster` object, which is to be sliced by taking only a subset of its rounds / iterations.
`start`	Start of the slice (base-1 and inclusive, like R's `seq()`).
`end`	End of the slice (base-1 and inclusive, like R's `seq()`). Passing a value of zero here is equivalent to passing the full number of rounds in the booster object.
`step`	Step size of the slice. Passing '1' will take every round in the sequence defined by `(start, end)`, while passing '2' will take every second value, and so on.
`i`	The indices - must be an increasing sequence as generated by e.g. `seq(...)`.

Details

Note that any R attributes that the booster might have, will not be copied into the resulting object.

Value

A sliced booster object containing only the requested rounds.

Examples

```
data(mtcars)

y <- mtcars$mpg
x <- as.matrix(mtcars[, -1])

dm <- xgb.DMatrix(x, label = y, nthread = 1)
model <- xgb.train(data = dm, params = xgb.params(nthread = 1), nrounds = 5)
model_slice <- xgb.slice.Booster(model, 1, 3)
# Prediction for first three rounds
predict(model, x, predleaf = TRUE)[, 1:3]

# The new model has only those rounds, so
# a full prediction from it is equivalent
predict(model_slice, x, predleaf = TRUE)
```

`xgb.slice.DMatrix` *Slice DMatrix*

Description

Get a new DMatrix containing the specified rows of original xgb.DMatrix object.

Usage

```
xgb.slice.DMatrix(object, idxset, allow_groups = FALSE)

## S3 method for class 'xgb.DMatrix'
object[idxset, colset = NULL]
```

Arguments

`object`	Object of class `xgb.DMatrix`.
`idxset`	An integer vector of indices of rows needed (base-1 indexing).
`allow_groups`	Whether to allow slicing an `xgb.DMatrix` with `group` (or equivalently `qid`) field. Note that in such case, the result will not have the groups anymore - they need to be set manually through `setinfo()`.
`colset`	Currently not used (columns subsetting is not available).

Examples

```
data(agaricus.train, package = "xgboost")

dtrain <- with(agaricus.train, xgb.DMatrix(data, label = label, nthread = 2))

dsub <- xgb.slice.DMatrix(dtrain, 1:42)
labels1 <- getinfo(dsub, "label")

dsub <- dtrain[1:42, ]
labels2 <- getinfo(dsub, "label")
all.equal(labels1, labels2)
```

`xgb.train` *Fit XGBoost Model*

Description

Fits an XGBoost model to given data in DMatrix format (e.g. as produced by `xgb.DMatrix()`). See the tutorial Introduction to Boosted Trees for a longer explanation of what XGBoost does, and the rest of the XGBoost Tutorials for further explanations XGBoost's features and usage.

Compared to function `xgboost()` which is a user-friendly function targeted towards interactive usage, `xgb.train` is a lower-level interface which allows finer-grained control and exposes further functionalities offered by the core library (such as learning-to-rank objectives), but which works exclusively with XGBoost's own data format ("DMatrices") instead of with regular R objects.

The syntax of this function closely mimics the same function from the Python package for XGBoost, and is recommended to use for package developers over `xgboost()` as it will provide a more stable interface (with fewer breaking changes) and lower overhead from data validations.

See also the migration guide if coming from a previous version of XGBoost in the 1.x series.

Usage

```
xgb.train(
  params = xgb.params(),
  data,
  nrounds,
  evals = list(),
  objective = NULL,
  custom_metric = NULL,
  verbose = 1,
  print_every_n = 1L,
  early_stopping_rounds = NULL,
  maximize = NULL,
  save_period = NULL,
  save_name = "xgboost.model",
  xgb_model = NULL,
  callbacks = list(),
  ...
)
```

Arguments

`params`	List of XGBoost parameters which control the model building process. See the online documentation and the documentation for `xgb.params()` for details. Should be passed as list with named entries. Parameters that are not specified in this list will use their default values. A list of named parameters can be created through the function `xgb.params()`, which accepts all valid parameters as function arguments.
`data`	Training dataset. `xgb.train()` accepts only an `xgb.DMatrix` as the input. Note that there is a function `xgboost()` which is meant to accept R data objects as inputs, such as data frames and matrices.
`nrounds`	Max number of boosting iterations.
`evals`	Named list of `xgb.DMatrix` datasets to use for evaluating model performance. Metrics specified in either `eval_metric` (under params) or `custom_metric` (function argument here) will be computed for each of these datasets during each boosting iteration, and stored in the end as a field named `evaluation_log` in the resulting object. When either `verbose>=1` or `xgb.cb.print.evaluation()` callback is engaged, the performance results are continuously printed out during the training. E.g., specifying `evals=list(validation1=mat1, validation2=mat2)` allows to track the performance of each round's model on `mat1` and `mat2`.
`objective`	Customized objective function. Should take two arguments: the first one will be the current predictions (either a numeric vector or matrix depending on the number of targets / classes), and the second one will be the `data` DMatrix object that is used for training. It should return a list with two elements `grad` and `hess` (in that order), as either numeric vectors or numeric matrices depending on the number of targets / classes (same dimension as the predictions that are passed as first argument).
`custom_metric`	Customized evaluation function. Just like `objective`, should take two arguments, with the first one being the predictions and the second one the `data` DMatrix. Should return a list with two elements `metric` (name that will be displayed for this metric, should be a string / character), and `value` (the number that the function calculates, should be a numeric scalar).

	Note that even if passing `custom_metric`, objectives also have an associated default metric that will be evaluated in addition to it. In order to disable the built-in metric, one can pass parameter `disable_default_eval_metric = TRUE`.
`verbose`	If 0, xgboost will stay silent. If 1, it will print information about performance. If 2, some additional information will be printed out. Note that setting `verbose > 0` automatically engages the `xgb.cb.print.evaluation(period=1)` callback function.
`print_every_n`	When passing `verbose>0`, evaluation logs (metrics calculated on the data passed under `evals`) will be printed every nth iteration according to the value passed here. The first and last iteration are always included regardless of this 'n'. Only has an effect when passing data under `evals` and when passing `verbose>0`. The parameter is passed to the `xgb.cb.print.evaluation()` callback.
`early_stopping_rounds`	Number of boosting rounds after which training will be stopped if there is no improvement in performance (as measured by the evaluatiation metric that is supplied or selected by default for the objective) on the evaluation data passed under `evals`. Must pass `evals` in order to use this functionality. Setting this parameter adds the `xgb.cb.early.stop()` callback. If `NULL`, early stopping will not be used.
`maximize`	If `feval` and `early_stopping_rounds` are set, then this parameter must be set as well. When it is `TRUE`, it means the larger the evaluation score the better. This parameter is passed to the `xgb.cb.early.stop()` callback.
`save_period`	When not `NULL`, model is saved to disk after every `save_period` rounds. 0 means save at the end. The saving is handled by the `xgb.cb.save.model()` callback.
`save_name`	the name or path for periodically saved model file.
`xgb_model`	A previously built model to continue the training from. Could be either an object of class `xgb.Booster`, or its raw data, or the name of a file with a previously saved model.
`callbacks`	A list of callback functions to perform various task during boosting. See `xgb.Callback()`. Some of the callbacks are automatically created depending on the parameters' values. User can provide either existing or their own callback methods in order to customize the training process. Note that some callbacks might try to leave attributes in the resulting model object, such as an evaluation log (a `data.table` object) - be aware that these objects are kept as R attributes, and thus do not get saved when using XGBoost's own serializaters like `xgb.save()` (but are kept when using R serializers like `saveRDS()`).
`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option.

If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option.

Important: `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

Compared to `xgboost()`, the `xgb.train()` interface supports advanced features such as `evals`, customized objective and evaluation metric functions, among others, with the difference these work `xgb.DMatrix` objects and do not follow typical R idioms.

Parallelization is automatically enabled if OpenMP is present. Number of threads can also be manually specified via the `nthread` parameter.

While in XGBoost language bindings, the default random seed defaults to zero, in R, if a parameter `seed` is not manually supplied, it will generate a random seed through R's own random number generator, whose seed in turn is controllable through `set.seed`. If `seed` is passed, it will override the RNG from R.

The following callbacks are automatically created when certain parameters are set:

- `xgb.cb.print.evaluation()` is turned on when `verbose > 0` and the `print_every_n` parameter is passed to it.
- `xgb.cb.evaluation.log()` is on when `evals` is present.
- `xgb.cb.early.stop()`: When `early_stopping_rounds` is set.
- `xgb.cb.save.model()`: When `save_period > 0` is set.

Note that objects of type `xgb.Booster` as returned by this function behave a bit differently from typical R objects (it's an 'altrep' list class), and it makes a separation between internal booster attributes (restricted to jsonifyable data), accessed through `xgb.attr()` and shared between interfaces through serialization functions like `xgb.save()`; and R-specific attributes (typically the result from a callback), accessed through `attributes()` and `attr()`, which are otherwise only used in the R interface, only kept when using R's serializers like `saveRDS()`, and not anyhow used by functions like `predict.xgb.Booster()`.

Be aware that one such R attribute that is automatically added is `params` - this attribute is assigned from the `params` argument to this function, and is only meant to serve as a reference for what went into the booster, but is not used in other methods that take a booster object - so for example, changing the booster's configuration requires calling `xgb.config<-` or `xgb.model.parameters<-`, while simply modifying `attributes(model)$params$<...>` will have no effect elsewhere.

Value

An object of class `xgb.Booster`.

References

Tianqi Chen and Carlos Guestrin, "XGBoost: A Scalable Tree Boosting System", 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016, https://arxiv.org/abs/1603.02754

See Also

xgb.Callback(), predict.xgb.Booster(), xgb.cv()

Examples

```
data(agaricus.train, package = "xgboost")
data(agaricus.test, package = "xgboost")

## Keep the number of threads to 1 for examples
nthread <- 1
data.table::setDTthreads(nthread)

dtrain <- with(
  agaricus.train, xgb.DMatrix(data, label = label, nthread = nthread)
)
dtest <- with(
  agaricus.test, xgb.DMatrix(data, label = label, nthread = nthread)
)
evals <- list(train = dtrain, eval = dtest)

## A simple xgb.train example:
param <- xgb.params(
  max_depth = 2,
  nthread = nthread,
  objective = "binary:logistic",
  eval_metric = "auc"
)
bst <- xgb.train(param, dtrain, nrounds = 2, evals = evals, verbose = 0)

## An xgb.train example where custom objective and evaluation metric are
## used:
logregobj <- function(preds, dtrain) {
   labels <- getinfo(dtrain, "label")
   preds <- 1/(1 + exp(-preds))
   grad <- preds - labels
   hess <- preds * (1 - preds)
   return(list(grad = grad, hess = hess))
}
evalerror <- function(preds, dtrain) {
  labels <- getinfo(dtrain, "label")
  err <- as.numeric(sum(labels != (preds > 0)))/length(labels)
  return(list(metric = "error", value = err))
}

# These functions could be used by passing them as 'objective' and
# 'eval_metric' parameters in the params list:
param <- xgb.params(
  max_depth = 2,
  nthread = nthread,
  objective = logregobj,
  eval_metric = evalerror
)
```

```
bst <- xgb.train(param, dtrain, nrounds = 2, evals = evals, verbose = 0)

# ... or as dedicated 'objective' and 'custom_metric' parameters of xgb.train:
bst <- xgb.train(
  within(param, rm("objective", "eval_metric")),
  dtrain, nrounds = 2, evals = evals,
  objective = logregobj, custom_metric = evalerror
)


## An xgb.train example of using variable learning rates at each iteration:
param <- xgb.params(
  max_depth = 2,
  learning_rate = 1,
  nthread = nthread,
  objective = "binary:logistic",
  eval_metric = "auc"
)
my_learning_rates <- list(learning_rate = c(0.5, 0.1))

bst <- xgb.train(
 param,
 dtrain,
 nrounds = 2,
 evals = evals,
 verbose = 0,
 callbacks = list(xgb.cb.reset.parameters(my_learning_rates))
)

## Early stopping:
bst <- xgb.train(
  param, dtrain, nrounds = 25, evals = evals, early_stopping_rounds = 3
)
```

xgboost *Fit XGBoost Model*

Description

Fits an XGBoost model (boosted decision tree ensemble) to given x/y data.

See the tutorial Introduction to Boosted Trees for a longer explanation of what XGBoost does, and the rest of the XGBoost Tutorials for further explanations XGBoost's features and usage.

This function is intended to provide a user-friendly interface for XGBoost that follows R's conventions for model fitting and predictions, but which doesn't expose all of the possible functionalities of the core XGBoost library.

See `xgb.train()` for a more flexible low-level alternative which is similar across different language bindings of XGBoost and which exposes additional functionalities such as training on external memory data and learning-to-rank objectives.

See also the migration guide if coming from a previous version of XGBoost in the 1.x series.

By default, most of the parameters here have a value of NULL, which signals XGBoost to use its default value. Default values are automatically determined by the XGBoost core library, and are subject to change over XGBoost library versions. Some of them might differ according to the booster type (e.g. defaults for regularization are different for linear and tree-based boosters). See `xgb.params()` and the online documentation for more details about parameters - but note that some of the parameters are not supported in the `xgboost()` interface.

Usage

```
xgboost(
  x,
  y,
  objective = NULL,
  nrounds = 100L,
  max_depth = NULL,
  learning_rate = NULL,
  min_child_weight = NULL,
  min_split_loss = NULL,
  reg_lambda = NULL,
  weights = NULL,
  verbosity = if (is.null(eval_set)) 0L else 1L,
  monitor_training = verbosity > 0,
  eval_set = NULL,
  early_stopping_rounds = NULL,
  print_every_n = 1L,
  eval_metric = NULL,
  nthreads = parallel::detectCores(),
  seed = 0L,
  base_margin = NULL,
  monotone_constraints = NULL,
  interaction_constraints = NULL,
  reg_alpha = NULL,
  max_bin = NULL,
  max_leaves = NULL,
  booster = NULL,
  subsample = NULL,
  sampling_method = NULL,
  feature_weights = NULL,
  colsample_bytree = NULL,
  colsample_bylevel = NULL,
  colsample_bynode = NULL,
  tree_method = NULL,
  max_delta_step = NULL,
  scale_pos_weight = NULL,
  updater = NULL,
  grow_policy = NULL,
  num_parallel_tree = NULL,
  multi_strategy = NULL,
  base_score = NULL,
  seed_per_iteration = NULL,
  device = NULL,
  disable_default_eval_metric = NULL,
  use_rmm = NULL,
  max_cached_hist_node = NULL,
  max_cat_to_onehot = NULL,
  max_cat_threshold = NULL,
  sample_type = NULL,
  normalize_type = NULL,
  rate_drop = NULL,
  one_drop = NULL,
  skip_drop = NULL,
  feature_selector = NULL,
  top_k = NULL,
  tweedie_variance_power = NULL,
  huber_slope = NULL,
  quantile_alpha = NULL,
  aft_loss_distribution = NULL,
  ...
)
```

Arguments

`x` The features / covariates. Can be passed as:

- A numeric or integer `matrix`.
- A `data.frame`, in which all columns are one of the following types:
  - `numeric`
  - `integer`
  - `logical`
  - `factor`

  Columns of `factor` type will be assumed to be categorical, while other column types will be assumed to be numeric.
- A sparse matrix from the `Matrix` package, either as `dgCMatrix` or `dgRMatrix` class.

Note that categorical features are only supported for `data.frame` inputs, and are automatically determined based on their types. See `xgb.train()` with `xgb.DMatrix()` for more flexible variants that would allow something like categorical features on sparse matrices.

`y` The response variable. Allowed values are:

- A numeric or integer vector (for regression tasks).
- A factor or character vector (for binary and multi-class classification tasks).
- A logical (boolean) vector (for binary classification tasks).
- A numeric or integer matrix or `data.frame` with numeric/integer columns (for multi-task regression tasks).

- A `Surv` object from the 'survival' package (for survival tasks).

If `objective` is `NULL`, the right task will be determined automatically based on the class of `y`.

If `objective` is not `NULL`, it must match with the type of `y` - e.g. `factor` types of `y` can only be used with classification objectives and vice-versa.

For binary classification, the last factor level of `y` will be used as the "positive" class - that is, the numbers from `predict` will reflect the probabilities of belonging to this class instead of to the first factor level. If `y` is a `logical` vector, then `TRUE` will be set as the last level.

`objective` Optimization objective to minimize based on the supplied data, to be passed by name as a string / character (e.g. `reg:absoluteerror`). See the Learning Task Parameters page and the `xgb.params()` documentation for more detailed information on allowed values.

If `NULL` (the default), will be automatically determined from `y` according to the following logic:

- If `y` is a factor with 2 levels, will use `binary:logistic`.
- If `y` is a factor with more than 2 levels, will use `multi:softprob` (number of classes will be determined automatically, should not be passed under `params`).
- If `y` is a `Surv` object from the `survival` package, will use `survival:aft` (note that the only types supported are left / right / interval censored).
- Otherwise, will use `reg:squarederror`.

If `objective` is not `NULL`, it must match with the type of `y` - e.g. `factor` types of `y` can only be used with classification objectives and vice-versa.

Note that not all possible `objective` values supported by the core XGBoost library are allowed here - for example, objectives which are a variation of another but with a different default prediction type (e.g. `multi:softmax` vs. `multi:softprob`) are not allowed, and neither are ranking objectives, nor custom objectives at the moment.

Supported values are:

- `"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:pseudohubererror"`: regression with Pseudo Huber loss, a twice differentiable alternative to absolute loss.
- `"reg:absoluteerror"`: Regression with L1 error. When tree model is used, leaf value is refreshed after tree construction. If used in distributed training, the leaf value is calculated as the mean value from all workers, which is not guaranteed to be optimal.
- `"reg:quantileerror"`: Quantile loss, also known as "pinball loss". See later sections for its parameter and Quantile Regression for a worked example.
- `"binary:logistic"`: logistic regression for binary classification, output probability

- `"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.
- `"multi:softprob"`: multi-class classification throgh multinomial logistic likelihood.
- `"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.

The following values are **NOT** supported by `xgboost`, but are supported by `xgb.train()` (see `xgb.params()` for details):

- `"reg:logistic"`
- `"binary:logitraw"`
- `"multi:softmax"`
- `"rank:ndcg"`
- `"rank:map"`
- `"rank:pairwise"`

`nrounds` Number of boosting iterations / rounds.

Note that the number of default boosting rounds here is not automatically tuned, and different problems will have vastly different optimal numbers of boosting rounds.

`max_depth` (for Tree Booster) (default=6, type=int32) 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: $[0, \infty)$

`learning_rate` (alias: `eta`) Step size shrinkage used in update to prevent overfitting. After each boosting step, we can directly get the weights of new features, and `learning_rate` shrinks the feature weights to make the boosting process more conservative.

- range: $[0, 1]$
- default value: 0.3 for tree-based boosters, 0.5 for linear booster.

`min_child_weight`
: (for Tree Booster) (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: $[0, \infty)$

`min_split_loss`
: (for Tree Booster) (default=0, alias: `gamma`) Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `min_split_loss` is, the more conservative the algorithm will be. Note that a tree where no splits were made might still contain a single terminal node with a non-zero score.

  range: $[0, \infty)$

`reg_lambda`
: (alias: `lambda`)

  - For tree-based boosters:
    - L2 regularization term on weights. Increasing this value will make model more conservative.
    - default: 1
    - range: $[0, \infty]$
  - For linear booster:
    - L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
    - default: 0
    - range: $[0, \infty)$

`weights`
: Sample weights for each row in `x` and `y`. If `NULL` (the default), each row will have the same weight.

  If not `NULL`, should be passed as a numeric vector with length matching to the number of rows in `x`.

`verbosity`
: Verbosity of printing messages. Valid values of 0 (silent), 1 (warning), 2 (info), and 3 (debug).

`monitor_training`
: Whether to monitor objective optimization progress on the input data. Note that same 'x' and 'y' data are used for both model fitting and evaluation.

`eval_set`
: Subset of the data to use as evaluation set. Can be passed as:

  - A vector of row indices (base-1 numeration) indicating the observations that are to be designed as evaluation data.
  - A number between zero and one indicating a random fraction of the input data to use as evaluation data. Note that the selection will be done uniformly at random, regardless of argument `weights`.

  If passed, this subset of the data will be excluded from the training procedure, and the evaluation metric(s) supplied under `eval_metric` will be calculated on this dataset after each boosting iteration (pass `verbosity>0` to have these metrics printed during training). If `eval_metric` is not passed, a default metric will be selected according to `objective`.

If passing a fraction, in classification problems, the evaluation set will be chosen in such a way that at least one observation of each class will be kept in the training data.

For more elaborate evaluation variants (e.g. custom metrics, multiple evaluation sets, etc.), one might want to use `xgb.train()` instead.

`early_stopping_rounds`
Number of boosting rounds after which training will be stopped if there is no improvement in performance (as measured by the last metric passed under `eval_metric`, or by the default metric for the objective if `eval_metric` is not passed) on the evaluation data from `eval_set`. Must pass `eval_set` in order to use this functionality.

If `NULL`, early stopping will not be used.

`print_every_n` When passing `verbosity>0` and either `monitor_training=TRUE` or `eval_set`, evaluation logs (metrics calculated on the training and/or evaluation data) will be printed every nth iteration according to the value passed here. The first and last iteration are always included regardless of this 'n'.

Only has an effect when passing `verbosity>0`.

`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 `rank:map`, etc.)
- User can add multiple evaluation metrics.
- 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"`: area under the receiver-operating characteristic 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.
- `"pre"`: Precision at k. Supports only learning to rank task.
- `"ndcg"`: normalized discounted cumulative gain
- `"map"`: mean average precision
  The `average precision` is defined as:
  $AP@l = \frac{1}{min(l,N)} \sum_{k=1}^{l} P@k \cdot I_{(k)}$
  where $I_{(k)}$ is an indicator function that equals to 1 when the document at k is relevant and 0 otherwise. The $P@k$ is the precision at k, and N is the total number of relevant documents. Lastly, the `mean average precision` is defined as the weighted average across all queries.
- `"ndcg@n"`, `"map@n"`, `"pre@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, the NDCG and MAP evaluate the score of a list without any positive samples as 1. By appending "-" to the evaluation metric name, we can ask XGBoost to evaluate these scores 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.

nthreads
: Number of parallel threads to use. If passing zero, will use all CPU threads.

seed
: Seed to use for random number generation. If passing NULL, will draw a random number using R's PRNG system to use as seed.

base_margin
: Base margin used for boosting from existing model.

  If passing it, will start the gradient boosting procedure from the scores that are provided here - for example, one can pass the raw scores from a previous model, or some per-observation offset, or similar.

  Should be either a numeric vector or numeric matrix (for multi-class and multi-target objectives) with the same number of rows as x and number of columns corresponding to number of optimization targets, and should be in the untransformed scale (for example, for objective binary:logistic, it should have log-odds, not probabilities; and for objective multi:softprob, should have number of columns matching to number of classes in the data).

  Note that, if it contains more than one column, then columns will not be matched by name to the corresponding y - base_margin should have the same column order that the model will use (for example, for objective multi:softprob, columns of base_margin will be matched against levels(y) by their position, regardless of what colnames(base_margin) returns).

  If NULL, will start from zero, but note that for most objectives, an intercept is usually added (controllable through parameter base_score instead) when base_margin is not passed.

monotone_constraints
: Optional monotonicity constraints for features.

  Can be passed either as a named list (when x has column names), or as a vector. If passed as a vector and x has column names, will try to match the elements by name.

  A value of +1 for a given feature makes the model predictions / scores constrained to be a monotonically increasing function of that feature (that is, as the value of the feature increases, the model prediction cannot decrease), while a value of -1 makes it a monotonically decreasing function. A value of zero imposes no constraint.

  The input for monotone_constraints can be a subset of the columns of x if named, in which case the columns that are not referred to in monotone_constraints will be assumed to have a value of zero (no constraint imposed on the model for those features).

  See the tutorial Monotonic Constraints for a more detailed explanation.

`interaction_constraints`

Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a list of vectors referencing columns in the data, e.g. `list(c(1, 2), c(3, 4, 5))` (with these numbers being column indices, numeration starting at 1 - i.e. the first sublist references the first and second columns) or `list(c("Sepal.Length", "Sepal.Width"), c("Petal.Length", "Petal.Width"))` (references columns by names), where each vector is a group of indices of features that are allowed to interact with each other.

See the tutorial Feature Interaction Constraints for more information.

`reg_alpha`

(alias: `reg_alpha`)

- L1 regularization term on weights. Increasing this value will make model more conservative.
- For the linear booster, it's normalised to number of training examples.
- default: 0
- range: $[0, \infty)$

`max_bin`

(for Tree Booster) (default=256, type=int32)

- Only used if `tree_method` is set to `"hist"` or `"approx"`.
- Maximum number of discrete bins to bucket continuous features.
- Increasing this number improves the optimality of splits at the cost of higher computation time.

`max_leaves`

(for Tree Booster) (default=0, type=int32) Maximum number of nodes to be added. Not used by `"exact"` tree method.

`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.

`subsample`

(for Tree Booster) (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`

(for Tree Booster) (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 `"hist"`; other tree methods only support `"uniform"` sampling.

`feature_weights`

Feature weights for column sampling.

Can be passed either as a vector with length matching to columns of `x`, or as a named list (only if `x` has column names) with names matching to columns of 'x'. If it is a named vector, will try to match the entries to column names of x by name.

If NULL (the default), all columns will have the same weight.

`colsample_bytree`, `colsample_bylevel`, `colsample_bynode`

(for Tree Booster) (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. This is not supported by the exact tree method.
- "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.

One can set the "feature_weights" for DMatrix to define the probability of each feature being selected when using column sampling.

`tree_method`

(for Tree Booster) (default= "auto") The tree construction algorithm used in XGBoost. See description in the reference paper and Tree Methods.

Choices: "auto", "exact", "approx", "hist", this is a combination of commonly used updaters. For other updaters like "refresh", set the parameter updater directly.

- "auto": Same as the "hist" tree method.
- "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.

`max_delta_step`

(for Tree Booster) (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: $[0, \infty)$

`scale_pos_weight`

(for Tree Booster) (default=1) Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: sum(negative instances) / sum(positive

See Parameters Tuning for more discussion. Also, see Higgs Kaggle competition demo for examples: R, py1, py2, py3.

`updater`

(for Linear Booster) (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 multi-threaded but still produces a deterministic solution. When the device parameter is set to "cuda" or "gpu", a GPU variant would be used.

grow_policy (for Tree Booster) (default= "depthwise")

- Controls a way new nodes are added to the tree.
- Currently supported only if tree_method is set to "hist" or "approx".
- Choices: "depthwise", "lossguide"
  - "depthwise": split at nodes closest to the root.
  - "lossguide": split at nodes with highest loss change.

num_parallel_tree
(for Tree Booster) (default=1) Number of parallel trees constructed during each iteration. This option is used to support boosted random forest.

multi_strategy (for Tree Booster) (default = "one_output_per_tree") The strategy used for training multi-target models, including multi-target regression and multi-class classification. See Multiple Outputs for more information.

- "one_output_per_tree": One model for each target.
- "multi_output_tree": Use multi-target trees.

Version added: 2.0.0

Note: This parameter is working-in-progress.

base_score
- The initial prediction score of all instances, global bias
- The parameter is automatically estimated for selected objectives before training. To disable the estimation, specify a real number argument.
- If base_margin is supplied, base_score will not be added.
- For sufficient number of iterations, changing this value will not have too much effect.

seed_per_iteration
(default= FALSE) Seed PRNG determnisticly via iterator number.

device (default= "cpu") Device for XGBoost to run. User can set it to one of the following values:

- "cpu": Use CPU.
- "cuda": Use a GPU (CUDA device).
- "cuda:<ordinal>": <ordinal> is an integer that specifies the ordinal of the GPU (which GPU do you want to use if you have more than one devices).
- "gpu": Default GPU device selection from the list of available and supported devices. Only "cuda" devices are supported currently.
- "gpu:<ordinal>": Default GPU device selection from the list of available and supported devices. Only "cuda" devices are supported currently.

For more information about GPU acceleration, see XGBoost GPU Support. In distributed environments, ordinal selection is handled by distributed frameworks instead of XGBoost. As a result, using `"cuda:<ordinal>"` will result in an error. Use `"cuda"` instead.

Version added: 2.0.0

Note: if XGBoost was installed from CRAN, it won't have GPU support enabled, thus only `"cpu"` will be available. To get GPU support, the R package for XGBoost must be installed from source or from the GitHub releases - see instructions.

`disable_default_eval_metric`
: (default= `FALSE`) Flag to disable default metric. Set to 1 or `TRUE` to disable.

`use_rmm`
: Whether to use RAPIDS Memory Manager (RMM) to allocate cache GPU memory. The primary memory is always allocated on the RMM pool when XGBoost is built (compiled) with the RMM plugin enabled. Valid values are `TRUE` and `FALSE`. See Using XGBoost with RAPIDS Memory Manager (RMM) plugin for details.

`max_cached_hist_node`
: (for Non-Exact Tree Methods) (default = 65536) Maximum number of cached nodes for histogram. This can be used with the `"hist"` and the `"approx"` tree methods.

  Version added: 2.0.0

  - For most of the cases this parameter should not be set except for growing deep trees. After 3.0, this parameter affects GPU algorithms as well.

`max_cat_to_onehot`
: (for Non-Exact Tree Methods) 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.

  Version added: 1.6.0

`max_cat_threshold`
: (for Non-Exact Tree Methods) Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting.

  Version added: 1.7.0

`sample_type`
: (for Dart Booster) (default= `"uniform"`) Type of sampling algorithm.

  - `"uniform"`: dropped trees are selected uniformly.
  - `"weighted"`: dropped trees are selected in proportion to weight.

`normalize_type`
: (for Dart Booster) (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`
: (for Dart Booster) (default=0.0) Dropout rate (a fraction of previous trees to drop during the dropout).

  range: $[0.0, 1.0]$

`one_drop` (for Dart Booster) (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` (for Dart Booster) (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: $[0.0, 1.0]$

`feature_selector`

(for Linear Booster) (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` (for Linear Booster) (default=0) The number of top features to select in `greedy` and `thrifty` feature selector. The value of 0 means using all the features.

`tweedie_variance_power`

(for Tweedie Regression (`"objective=reg:tweedie"`)) (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.

`huber_slope` (for using Pseudo-Huber (`"reg:pseudohubererror"`)) (default = 1.0) A parameter used for Pseudo-Huber loss to define the δ term.

`quantile_alpha` (for using Quantile Loss (`"reg:quantileerror"`)) A scalar or a list of targeted quantiles (passed as a numeric vector).

Version added: 2.0.0

`aft_loss_distribution`

(when using AFT Survival Loss (`"survival:aft"`) and Negative Log Likelihood of AFT metric (`"aft-nloglik"`)) Probability Density Function, `"normal"`, `"logistic"`, or `"extreme"`.

`...`	Not used. Some arguments that were part of this function in previous XGBoost versions are currently deprecated or have been renamed. If a deprecated or renamed argument is passed, will throw a warning (by default) and use its current equivalent instead. This warning will become an error if using the 'strict mode' option. If some additional argument is passed that is neither a current function argument nor a deprecated or renamed argument, a warning or error will be thrown depending on the 'strict mode' option. **Important:** `...` will be removed in a future version, and all the current deprecation warnings will become errors. Please use only arguments that form part of the function signature.

Details

For package authors using 'xgboost' as a dependency, it is highly recommended to use `xgb.train()` in package code instead of `xgboost()`, since it has a more stable interface and performs fewer data conversions and copies along the way.

Value

A model object, inheriting from both `xgboost` and `xgb.Booster`. Compared to the regular `xgb.Booster` model class produced by `xgb.train()`, this `xgboost` class will have an additional attribute `metadata` containing information which is used for formatting prediction outputs, such as class names for classification problems.

References

- Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.
- https://xgboost.readthedocs.io/en/stable/

Examples

```
data(mtcars)

# Fit a small regression model on the mtcars data
model_regression <- xgboost(mtcars[, -1], mtcars$mpg, nthreads = 1, nrounds = 3)
predict(model_regression, mtcars, validate_features = TRUE)

# Task objective is determined automatically according to the type of 'y'
data(iris)
model_classif <- xgboost(iris[, -5], iris$Species, nthreads = 1, nrounds = 5)
predict(model_classif, iris[1:10,])
predict(model_classif, iris[1:10,], type = "class")

# Can nevertheless choose a non-default objective if needed
model_poisson <- xgboost(
  mtcars[, -1], mtcars$mpg,
  objective = "count:poisson",
  nthreads = 1,
  nrounds = 3
)

# Can calculate evaluation metrics during boosting rounds
data(ToothGrowth)
xgboost(
  ToothGrowth[, c("len", "dose")],
  ToothGrowth$supp,
  eval_metric = c("auc", "logloss"),
  eval_set = 0.2,
  monitor_training = TRUE,
  verbosity = 1,
  nthreads = 1,
  nrounds = 3
)
```

---

xgboost-options *XGBoost Options*

Description

XGBoost offers an option setting for controlling the behavior of deprecated and removed function arguments.

Some of the arguments in functions like `xgb.train()` or `predict.xgb.Booster()` been renamed from how they were in previous versions, or have been removed.

In order to make the transition to newer XGBoost versions easier, some of these parameters are still accepted but issue a warning when using them. **Note that these warnings will become errors in the future!!** - this is just a temporary workaround to make the transition easier.

One can optionally use 'strict mode' to turn these warnings into errors, in order to ensure that code calling xgboost will still work once those are removed in future releases.

Currently, the only supported option is `xgboost.strict_mode`, which can be set to `TRUE` or `FALSE` (default).

In addition to an R option, it can also be enabled through by setting environment variable `XGB_STRICT_MODE=1`. If set, this environment variable will take precedence over the option.

Examples

```
options("xgboost.strict_mode" = FALSE)
options("xgboost.strict_mode" = TRUE)
Sys.setenv("XGB_STRICT_MODE" = "1")
Sys.setenv("XGB_STRICT_MODE" = "0")
```

Index