# Package 'xgboost'

January 5, 2017

**Type** Package

**Title** Extreme Gradient Boosting

**Version** 0.6-4

**Date** 2017-01-04

**Author** Tianqi Chen <tianqi.tchen@gmail.com>, Tong He <hetong007@gmail.com>, Michael Benesty <michael@benesty.fr>, Vadim Khotilovich <khotilovich@gmail.com>, Yuan Tang <terrytangyuan@gmail.com>

**Maintainer** Tong He <hetong007@gmail.com>

**Description** Extreme Gradient Boosting, which is an efficient implementation of the gradient boosting framework 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

**VignetteBuilder** knitr

**Suggests** knitr, rmarkdown, ggplot2 (>= 1.0.1), DiagrammeR (>= 0.9.0), Ckmeans.1d.dp (>= 3.3.1), vcd (>= 1.3), testthat, igraph (>= 1.0.1)

**Depends** R (>= 3.3.0)

**Imports** Matrix (>= 1.1-0), methods, data.table (>= 1.9.6), magrittr (>= 1.5), stringi (>= 0.5.2)

**RoxygenNote** 5.0.1

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2017-01-05 10:40:06

# R topics documented:

## 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

This data set 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

This data set 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.

---

`callbacks` *Callback closures for booster training.*

---

## Description

These are used to perform various service tasks either during boosting iterations or at the end. This approach helps to modularize many of such tasks without bloating the main training methods, and it offers .

## Details

By default, a callback function is run after each boosting iteration. An R-attribute `is_pre_iteration` could be set for a callback to define a pre-iteration function.

When a callback function has `finalize` parameter, its finalizer part will also be run after the boosting is completed.

WARNING: side-effects!!! Be aware that these callback functions access and modify things in the environment from which they are called from, which is a fairly uncommon thing to do in R.

To write a custom callback closure, make sure you first understand the main concepts about R envoronments. Check either R documentation on `environment` or the Environments chapter from the "Advanced R" book by Hadley Wickham. Further, the best option is to read the code of some of the existing callbacks - choose ones that do something similar to what you want to achieve. Also, you would need to get familiar with the objects available inside of the `xgb.train` and `xgb.cv` internal environments.

## See Also

`cb.print.evaluation`, `cb.evaluation.log`, `cb.reset.parameters`, `cb.early.stop`, `cb.save.model`, `cb.cv.predict`, `xgb.train`, `xgb.cv`

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

## Description

Callback closure for returning cross-validation based predictions.

## Usage

```
cb.cv.predict(save_models = FALSE)
```

## Arguments

| | |
|---|---|
| save_models | a flag for whether to save the folds' models. |

## Details

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

It is a "finalizer" callback and it uses early stopping information whenever it is available, thus it must be run after the early stopping callback if the early stopping is used.

Callback function expects the following values to be set in its calling frame: `bst_folds`, `basket`, `data`, `end_iteration`, `params`, `num_parallel_tree`, `num_class`.

## Value

Predictions are returned 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-profided 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`.

## See Also

`callbacks`

cb.early.stop *Callback closure to activate the early stopping.*

## Description

Callback closure to activate the early stopping.

## Usage

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

## 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 watchlist was labelled as dtest, and one wants to use the AUC in test data for early stopping regardless of where it is in the watchlist, 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. |

## Details

This callback function determines the condition for early stopping by setting the stop_condition = TRUE flag in its calling frame.

The following additional fields are assigned to the model's R object:

- best_score the evaluation score at the best iteration
- best_iteration at which boosting iteration the best score has occurred (1-based index)
- best_ntreelimit to use with the ntreelimit parameter in predict. It differs from best_iteration in multiclass or random forest settings.

The Same values are also stored as xgb-attributes:

- best_iteration is stored as a 0-based iteration index (for interoperability of binary models)
- best_msg message string is also stored.

At least one data element is required in the evaluation watchlist for early stopping to work.

Callback function expects the following values to be set in its calling frame: stop_condition, bst_evaluation, rank, bst (or bst_folds and basket), iteration, begin_iteration, end_iteration, num_parallel_tree.

## See Also

callbacks, xgb.attr

---

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

---

## Description

Callback closure for logging the evaluation history

## Usage

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

## Details

This callback function appends the current iteration evaluation results `bst_evaluation` available in the calling parent frame to the `evaluation_log` list in a calling frame.

The finalizer callback (called with `finalize = TURE` in the end) converts the `evaluation_log` list into a final data.table.

The iteration evaluation result `bst_evaluation` must be a named numeric vector.

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.

Callback function expects the following values to be set in its calling frame: `evaluation_log`, `bst_evaluation`, `iteration`.

## See Also

callbacks

---

`cb.print.evaluation` *Callback closure for printing the result of evaluation*

---

## Description

Callback closure for printing the result of evaluation

## Usage

```
cb.print.evaluation(period = 1)
```

## Arguments

| | |
|---|---|
| `period` | results would be printed every number of periods |

## Details

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

Callback function expects the following values to be set in its calling frame: `bst_evaluation` (also `bst_evaluation_err` when available), `iteration`, `begin_iteration`, `end_iteration`.

## See Also

`callbacks`

---

`cb.reset.parameters` *Callback closure for restetting the booster's parameters at each iteration.*

---

## Description

Callback closure for restetting the booster's parameters at each iteration.

## Usage

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

## Arguments

| | |
|---|---|
| `new_params` | a list where each element corresponds to a parameter that needs 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

This is a "pre-iteration" callback function used to reset booster's parameters at the beginning of each iteration.

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

Callback function expects the following values to be set in its calling frame: `bst` or `bst_folds`, `iteration`, `begin_iteration`, `end_iteration`.

## See Also

`callbacks`

## cb.save.model *Callback closure for saving a model file.*

### Description

Callback closure for saving a model file.

### Usage

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

### 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_ the file saved at iteration 50 would be named "xgboost_0050.model". |

### Details

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

Callback function expects the following values to be set in its calling frame: bst, iteration, begin_iteration, end_iteration.

### See Also

callbacks

## 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)

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 returnten 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)
dimnames(dtrain)
colnames(dtrain)
colnames(dtrain) <- make.names(1:ncol(train$data))
print(dtrain, verbose=TRUE)
```

---

# getinfo *Get information of an xgb.DMatrix object*

---

## Description

Get information of an xgb.DMatrix object

## Usage

```
getinfo(object, ...)

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

## Arguments

| | |
|---|---|
| object | Object of class xgb.DMatrix |
| ... | other parameters |
| name | the name of the information field to get (see details) |

## Details

The name field can be one of the following:

- label: label Xgboost learn from ;
- weight: to do a weight rescale ;
- base_margin: base margin is the base prediction Xgboost will boost from ;
- nrow: number of rows of the xgb.DMatrix.

## Examples

```
data(agaricus.train, package='xgboost')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)

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

labels2 <- getinfo(dtrain, 'label')
stopifnot(all(labels2 == 1-labels))
```

---

# predict.xgb.Booster *Predict method for eXtreme Gradient Boosting model*

---

## Description

Predicted values based on either xgboost model or model handle object.

## Usage

```
## S3 method for class 'xgb.Booster'
predict(object, newdata, missing = NA,
  outputmargin = FALSE, ntreelimit = NULL, predleaf = FALSE,
  reshape = FALSE, ...)

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

## Arguments

| | |
|---|---|
| object | Object of class xgb.Booster or xgb.Booster.handle |
| newdata | takes matrix, dgCMatrix, local data file or xgb.DMatrix. |
| missing | Missing is only used when input is dense matrix. Pick a float value that represents missing values in data (e.g., sometimes 0 or some other extreme value is used). |
| outputmargin | whether the prediction should be returned in the for of original untransformed sum of predictions from boosting iterations' results. E.g., setting outputmargin=TRUE for logistic regression would result in predictions for log-odds instead of probabilities. |
| ntreelimit | limit the number of model's trees or boosting iterations used in prediction (see Details). It will use all the trees by default (NULL value). |
| predleaf | whether predict leaf index instead. |
| reshape | whether to reshape the vector of predictions to a matrix form when there are several prediction outputs per case. This option has no effect when predleaf = TRUE. |
| ... | Parameters passed to predict.xgb.Booster |

## Details

Note that `ntreelimit` is not necessarily equal to the number of boosting iterations and it is not necessarily equal to the number of trees in a model. E.g., in a random forest-like model, `ntreelimit` would limit the number of trees. But for multiclass classification, there are multiple trees per iteration, but `ntreelimit` limits the number of boosting iterations.

Also note that `ntreelimit` 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`.

## Value

For regression or binary classification, it returns a vector of length `nrows(newdata)`. For multiclass classification, either a `num_class * nrows(newdata)` vector or a `(nrows(newdata), num_class)` dimension matrix is returned, depending on the `reshape` value.

When `predleaf = TRUE`, the output is a matrix object with the number of columns corresponding to the number of trees.

## See Also

`xgb.train`.

## Examples

```
## binary classification:

data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test

bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
# use all trees by default
pred <- predict(bst, test$data)
# use only the 1st tree
pred <- predict(bst, test$data, ntreelimit = 1)


## multiclass classification in iris dataset:

lb <- as.numeric(iris$Species) - 1
num_class <- 3
set.seed(11)
bst <- xgboost(data = as.matrix(iris[, -5]), label = lb,
               max_depth = 4, eta = 0.5, nthread = 2, nrounds = 10, 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)
```

```
# reshape it to a num_class-columns matrix
pred <- matrix(pred, ncol=num_class, byrow=TRUE)
# 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 that to the predictions from softmax:
set.seed(11)
bst <- xgboost(data = as.matrix(iris[, -5]), label = lb,
               max_depth = 4, eta = 0.5, nthread = 2, nrounds = 10, 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]), ntreelimit=5)
sum(pred5 != lb)/length(lb)


## random forest-like model of 25 trees for binary classification:

set.seed(11)
bst <- xgboost(data = train$data, label = train$label, max_depth = 5,
               nthread = 2, nrounds = 1, objective = "binary:logistic",
               num_parallel_tree = 25, subsample = 0.6, colsample_bytree = 0.1)
# Inspect the prediction error vs number of trees:
lb <- test$label
dtest <- xgb.DMatrix(test$data, label=lb)
err <- sapply(1:25, function(n) {
  pred <- predict(bst, dtest, ntreelimit=n)
  sum((pred > 0.5) != lb)/length(lb)
})
plot(err, type='l', ylim=c(0,0.1), xlab='#trees')
```

---

`print.xgb.Booster` *Print xgb.Booster*

---

## Description

Print information about xgb.Booster.

## Usage

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

### Arguments

| | |
|---|---|
| x | an xgb.Booster object |
| verbose | whether to print detailed data (e.g., attribute values) |
| ... | not currently used |

### Examples

```
data(agaricus.train, package='xgboost')
train <- agaricus.train
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
attr(bst, 'myattr') <- 'memo'

print(bst)
print(bst, verbose=TRUE)
```

---

## 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 = train$data, label = train$label, nfold = 5, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 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')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)

dtrain
print(dtrain, verbose=TRUE)
```

setinfo *Set information of an xgb.DMatrix object*

## Description

Set information of an xgb.DMatrix object

## Usage

```
setinfo(object, ...)

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

## Arguments

| | |
|---|---|
| object | Object of class "xgb.DMatrix" |
| ... | other parameters |
| name | the name of the field to get |
| info | the specific field of information to set |

## Details

The name field can be one of the following:

- label: label Xgboost learn from ;
- weight: to do a weight rescale ;
- base_margin: base margin is the base prediction Xgboost will boost from ;
- group.

## Examples

```
data(agaricus.train, package='xgboost')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)

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

slice *Get a new DMatrix containing the specified rows of orginal xgb.DMatrix object*

## Description

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

## Usage

```
slice(object, ...)

## S3 method for class 'xgb.DMatrix'
slice(object, idxset, ...)

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

## Arguments

| | |
|---|---|
| object | Object of class "xgb.DMatrix" |
| ... | other parameters (currently not used) |
| idxset | a integer vector of indices of rows needed |
| colset | currently not used (columns subsetting is not available) |

## Examples

```
data(agaricus.train, package='xgboost')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)

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

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 or xgb.Booster.handle. |
| name | a non-empty character string specifying which attribute is to be accessed. |
| value | a value of an attribute for xgb.attr<-; for xgb.attributes<- it's 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 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.parameters<- to set or change model parameters.

The attribute setters would usually work more efficiently for xgb.Booster.handle than for xgb.Booster, since only just a handle (pointer) would need to be copied. That would only matter if attributes need to be set many times. Note, however, that when feeding a handle of an xgb.Booster object to the attribute setters, the raw model cache of an xgb.Booster object would not be automatically updated, and it would be user's responsibility to call xgb.save.raw to update it.

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

## 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 attribute stored in a model or NULL if a model has no stored attributes.

## Examples

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

```
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 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")

xgb.save(bst, 'xgb.model')
bst1 <- xgb.load('xgb.model')
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.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) |
| `...` | currently not used |

## 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.fb.com/publications/practical-lessons-from-predicting-clicks-on-ads-at-facebook/.

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

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

## Examples

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
dtrain <- xgb.DMatrix(data = agaricus.train$data, label = agaricus.train$label)
dtest <- xgb.DMatrix(data = agaricus.test$data, label = agaricus.test$label)

param <- list(max_depth=2, eta=1, silent=1, objective='binary:logistic')
nround = 4

bst = xgb.train(params = param, data = dtrain, nrounds = nround, nthread = 2)

# 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$label)
new.dtest <- xgb.DMatrix(data = new.features.test, label = agaricus.test$label)
watchlist <- list(train = new.dtrain)
bst <- xgb.train(params = param, data = new.dtrain, nrounds = nround, nthread = 2)

# 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 = list(), data, nrounds, nfold, label = NULL, missing = NA,
  prediction = FALSE, showsd = TRUE, metrics = list(), obj = NULL,
  feval = NULL, stratified = TRUE, folds = NULL, verbose = TRUE,
  print_every_n = 1L, early_stopping_rounds = NULL, maximize = NULL,
  callbacks = list(), ...)
```

## Arguments

| | |
|---|---|
| params | the list of parameters. Commonly used ones are:<br>• objective objective function, common ones are<br>– reg:linear linear regression<br>– binary:logistic logistic regression for classification<br>• eta step size of each boosting step<br>• max_depth maximum depth of the tree<br>• nthread number of thread used in training, if not set, all threads are used<br>See xgb.train for further details. See also demo/ for walkthrough example in R. |
| data | takes an xgb.DMatrix, matrix, or dgCMatrix as the input. |
| nrounds | the max number of iterations |
| nfold | the original dataset is randomly partitioned into nfold equal size subsamples. |
| label | vector of response values. Should be provided only when data is an R-matrix. |
| missing | is only used when input is a dense matrix. By default is set to NA, which means that NA values should be considered as 'missing' by the algorithm. Sometimes, 0 or other extreme value might be used to represent missing values. |
| prediction | A logical value indicating whether to return the test fold predictions from each CV model. This parameter engages the cb.cv.predict callback. |
| showsd | boolean, 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:<br>• error binary classification error rate<br>• rmse Rooted mean square error |

- logloss negative log-likelihood function
- auc Area under curve
- merror Exact matching error, used to evaluate multi-class classification

obj — customized objective function. Returns gradient and second order gradient with given prediction and dtrain.

feval — custimized evaluation function. Returns list(metric='metric-name', value='metric-value') with given prediction and dtrain.

stratified — a boolean indicating whether sampling of folds should be stratified by the values of outcome labels.

folds — list provides a possibility to use a list of 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.

verbose — boolean, print the statistics during the process

print_every_n — Print each n-th iteration evaluation messages when verbose>0. Default is 1 which means all messages are printed. This parameter is passed to the cb.print.evaluation callback.

early_stopping_rounds — If NULL, the early stopping function is not triggered. If set to an integer k, training with a validation set will stop if the performance doesn't improve for k rounds. Setting this parameter engages the cb.early.stop callback.

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 cb.early.stop callback.

callbacks — a list of callback functions to perform various task during boosting. See callbacks. 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.

... — other parameters to pass to params.

## 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 http://en.wikipedia.org/wiki/Cross-validation_%28statistics%29#k-fold_cross-validation

## Value

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

- call a function call.

- `params` parameters that were passed to the xgboost library. Note that it does not capture parameters changed by the `cb.reset.parameters` callback.
- `callbacks` callback functions that were either automatically assigned or explicitely passed.
- `evaluation_log` evaluation history storead 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 `cb.evaluation.log` callback.
- `niter` number of boosting iterations.
- `folds` the list of CV folds' indices - either those passed through the `folds` parameter or randomly generated.
- `best_iteration` iteration number with the best evaluation metric value (only available with early stopping).
- `best_ntreelimit` the `ntreelimit` value corresponding to the best iteration, which could further be used in `predict` method (only available with early stopping).
- `pred` CV prediction values available when `prediction` is set. It is either vector or matrix (see `cb.cv.predict`).
- `models` a liost of the CV folds' models. It is only available with the explicit setting of the `cb.cv.predict(save_models = TRUE)` callback.

## Examples

```
data(agaricus.train, package='xgboost')
dtrain <- xgb.DMatrix(agaricus.train$data, label = agaricus.train$label)
cv <- xgb.cv(data = dtrain, nrounds = 3, nthread = 2, nfold = 5, metrics = list("rmse","auc"),
                 max_depth = 3, eta = 1, objective = "binary:logistic")
print(cv)
print(cv, verbose=TRUE)
```

---

`xgb.DMatrix` *Contruct xgb.DMatrix object*

---

## Description

Contruct xgb.DMatrix object from dense matrix, sparse matrix or local file (that was created previously by saving an `xgb.DMatrix`).

## Usage

```
xgb.DMatrix(data, info = list(), missing = NA, ...)
```

### Arguments

| | |
|---|---|
| data | a matrix object, a dgCMatrix object or a character representing a filename |
| info | a list of information of the xgb.DMatrix object |
| missing | Missing is only used when input is dense matrix, pick a float value that represents missing value. Sometime a data use 0 or other extreme value to represents missing values. |
| ... | other information to pass to info. |

### Examples

```
data(agaricus.train, package='xgboost')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)
xgb.DMatrix.save(dtrain, 'xgb.DMatrix.data')
dtrain <- xgb.DMatrix('xgb.DMatrix.data')
```

---

## 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')
train <- agaricus.train
dtrain <- xgb.DMatrix(train$data, label=train$label)
xgb.DMatrix.save(dtrain, 'xgb.DMatrix.data')
dtrain <- xgb.DMatrix('xgb.DMatrix.data')
```

xgb.dump *Save xgboost model to text file*

## Description

Save a xgboost model to text file. Could be parsed later.

## Usage

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

## 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 function will return the model as a character vector. |
| fmap | feature map file representing the type of feature. Detailed description could be found at https://github.com/dmlc/xgboost/wiki/Binary-Classification#dump-model. See demo/ for walkthrough example in R, and https://github.com/dmlc/xgboost/blob/master/demo/data/featmap.txt for example Format. |
| with_stats | whether dump statistics of splits When this option is on, the model dump comes with two additional statistics: 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' or 'json' format could be specified. |
| ... | currently not used |

## 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 <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")
# save the model in file 'xgb.model.dump'
xgb.dump(bst, 'xgb.model.dump', 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'))
```

---

`xgb.ggplot.deepness` *Plot model trees deepness*

---

## Description

Visualizes distributions related to depth of tree leafs. `xgb.plot.deepness` uses base R graphics, while `xgb.ggplot.deepness` uses the ggplot backend.

## 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 generated by the `xgb.train` function or a data.table result of the `xgb.model.dt.tree` function. |
| `which` | which distribution to plot (see details). |
| `plot` | (base R barplot) whether a barplot should be produced. If FALSE, only a data.table is returned. |
| `...` | other parameters passed to `barplot` or `plot`. |

## Details

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

- the distribution of the number of leafs in a tree model at a certain depth;
- the distribution of average weighted number of observations ("cover") ending up in leafs at 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 tree number are created. And `which="med.weight"` allows to see how a tree's median absolute leaf weight changes through the iterations.

This function was inspired by the blog post http://aysent.github.io/2015/11/08/random-forest-leaf-visualizati
html.

## Value

Other than producing plots (when `plot=TRUE`), the `xgb.plot.deepness` function silently returns a processed data.table where each row corresponds to a terminal leaf in a tree model, and contains information about leaf's depth, cover, and weight (which is used in calculating predictions).

The `xgb.ggplot.deepness` silently returns either a list of two ggplot graphs when `which="2x1"` or a single ggplot graph for the other `which` options.

## See Also

`xgb.train`, `xgb.model.dt.tree`.

## Examples

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

# Change max_depth to a higher number to get a more significant result
bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label, max_depth = 6,
               eta = 0.1, nthread = 2, nrounds = 50, objective = "binary:logistic",
               subsample = 0.5, min_child_weight = 2)

xgb.plot.deepness(bst)
xgb.ggplot.deepness(bst)

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

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

---

# xgb.ggplot.importance *Plot feature importance as a bar graph*

---

## Description

Represents previously calculated feature importance as a bar graph. `xgb.plot.importance` uses base R graphics, while `xgb.ggplot.importance` uses the ggplot backend.

## Usage

```
xgb.ggplot.importance(importance_matrix = NULL, top_n = NULL,
  measure = NULL, rel_to_first = FALSE, n_clusters = c(1:10), ...)

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 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 | (ggplot only) a numeric vector containing the min and the max range of the possible number of clusters of bars. |
| ... | other parameters passed to barplot (except horiz, border, cex.names, names.arg, and las). |
| left_margin | (base R barplot) allows to adjust the left margin size to fit feature names. When it is NULL, the existing par('mar') is used. |
| cex | (base R barplot) passed as cex.names parameter to barplot. |
| plot | (base R barplot) whether a barplot should be produced. If FALSE, only a data.table is returned. |

## Details

The graph represents each feature as a horizontal bar of length proportional to the importance of a feature. Features are shown ranked in a decreasing importance order. It works for importances from both gblinear and gbtree models.

When rel_to_first = FALSE, the values would be plotted as they were in importance_matrix. For 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 method also performs 1-D custering of the importance values, with bar colors coresponding to different clusters that have somewhat similar importance values.

## Value

The xgb.plot.importance function creates a barplot (when plot=TRUE) and silently returns a processed data.table with n_top features sorted by importance.

The xgb.ggplot.importance function returns a ggplot graph which could be customized afterwards. E.g., to change the title of the graph, add + ggtitle("A GRAPH NAME") to the result.

## See Also

barplot.

## Examples

```
data(agaricus.train)

bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label, max_depth = 3,
               eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")

importance_matrix <- xgb.importance(colnames(agaricus.train$data), model = bst)

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 + ggplot2::ylab("Frequency")
```

---

# xgb.importance *Show importance of features in a model*

---

## Description

Create a data.table of the most important features of a model.

## Usage

```
xgb.importance(feature_names = NULL, model = NULL, data = NULL,
  label = NULL, target = function(x) ((x + label) == 2))
```

## Arguments

| | |
|---|---|
| feature_names | names of each feature as a character vector. Can be extracted from a sparse matrix (see example). If model dump already contains feature names, this argument should be NULL. |
| model | generated by the xgb.train function. |
| data | the dataset used for the training step. Will be used with label parameter for co-occurence computation. More information in Detail part. This parameter is optional. |
| label | the label vector used for the training step. Will be used with data parameter for co-occurence computation. More information in Detail part. This parameter is optional. |
| target | a function which returns TRUE or 1 when an observation should be count as a co-occurence and FALSE or 0 otherwise. Default function is provided for computing co-occurences in a binary classification. The target function should have only one parameter. This parameter will be used to provide each important feature vector after having applied the split condition, therefore these vector will be only made of 0 and 1 only, whatever was the information before. More information in Detail part. This parameter is optional. |

## Details

This function is for both linear and tree models.

`data.table` is returned by the function. The columns are:

- `Features` name of the features as provided in `feature_names` or already present in the model dump;
- `Gain` contribution of each feature to the model. For boosted tree model, each gain of each feature of each tree is taken into account, then average per feature to give a vision of the entire model. Highest percentage means important feature to predict the `label` used for the training (only available for tree models);
- `Cover` metric of the number of observation related to this feature (only available for tree models);
- `Weight` percentage representing the relative number of times a feature have been taken into trees.

If you don't provide `feature_names`, index of the features will be used instead.

Because the index is extracted from the model dump (made on the C++ side), it starts at 0 (usual in C++) instead of 1 (usual in R).

Co-occurence count ————————

The gain gives you indication about the information of how a feature is important in making a branch of a decision tree more pure. However, with this information only, you can't know if this feature has to be present or not to get a specific classification. In the example code, you may wonder if odor=none should be `TRUE` to not eat a mushroom.

Co-occurence computation is here to help in understanding this relation between a predictor and a specific class. It will count how many observations are returned as `TRUE` by the `target` function (see parameters). When you execute the example below, there are 92 times only over the 3140 observations of the train dataset where a mushroom have no odor and can be eaten safely.

If you need to remember only one thing: unless you want to leave us early, don't eat a mushroom which has no odor :-)

## Value

A `data.table` of the features used in the model with their average gain (and their weight for boosted tree model) in the model.

## Examples

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

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

xgb.importance(colnames(agaricus.train$data), model = bst)

# Same thing with co-occurence computation this time
xgb.importance(colnames(agaricus.train$data), model = bst,
               data = agaricus.train$data, label = agaricus.train$label)
```

## xgb.load *Load xgboost model from binary file*

### Description

Load xgboost model from the binary model file

### Usage

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

### Arguments

| | |
|---|---|
| modelfile | the name of the binary file. |

### Examples

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2,objective = "binary:logistic")
xgb.save(bst, 'xgb.model')
bst <- xgb.load('xgb.model')
pred <- predict(bst, test$data)
```

## xgb.model.dt.tree *Parse a boosted tree model text dump*

### Description

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

### Usage

```
xgb.model.dt.tree(feature_names = NULL, model = NULL, text = NULL,
  n_first_tree = NULL)
```

### Arguments

| | |
|---|---|
| feature_names | character vector of feature names. If the model already contains feature names, this argument should be NULL (default value) |
| model | object of class xgb.Booster |
| text | character vector previously generated by the xgb.dump function (where parameter with_stats = TRUE should have been set). |
| n_first_tree | limit the parsing to the n first trees. If set to NULL, all trees of the model are parsed. |

## Value

A `data.table` with detailed information about model trees' nodes.

The columns of the `data.table` are:

- `Tree`: ID of a tree in a model
- `Node`: ID of a node in a tree
- `ID`: unique identifier of a node in a model
- `Feature`: for a branch node, it's a feature id or name (when available); for a leaf note, 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 branch value is missing
- `Quality`: either the split gain (change in loss) or the leaf value
- `Cover`: metric related to the number of observation either seen by a split or collected by a leaf during training.

## Examples

```
# Basic use:

data(agaricus.train, package='xgboost')

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

(dt <- xgb.model.dt.tree(colnames(agaricus.train$data), 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.parameters<- *Accessors for model parameters.*

---

## Description

Only the setter for xgboost parameters is currently implemented.

## Usage

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

### Arguments

| | |
|---|---|
| object | Object of class xgb.Booster or xgb.Booster.handle. |
| 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

Note that the setter would usually work more efficiently for xgb.Booster.handle than for xgb.Booster, since only just a handle would need to be copied.

### Examples

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

bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2, objective = "binary:logistic")

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

---

## xgb.plot.multi.trees *Project all trees on one tree and plot it*

---

### Description

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

### Usage

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

### Arguments

| | |
|---|---|
| model | dump generated by the xgb.train function. |
| feature_names | names of each feature as a character vector. Can be extracted from a sparse matrix (see example). If model dump already contains feature names, this argument should be NULL. |
| features_keep | number of features to keep in each position of the multi trees. |
| plot_width | width in pixels of the graph to produce |
| plot_height | height in pixels of the graph to produce |
| ... | currently not used |

## Details

This function tries to capture the complexity of gradient boosted tree ensemble in a cohesive way.

The goal is to improve the interpretability of the model generally seen as black box. The function is dedicated to boosting applied to decision trees only.

The purpose is to move from an ensemble of trees to a single tree only.

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

Moreover, the trees tend to reuse the same features.

The function will project each tree on one, and keep for each position the `features_keep` first features (based on 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

Two graphs showing the distribution of the model deepness.

## Examples

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

bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label, max_depth = 15,
                eta = 1, nthread = 2, nrounds = 30, objective = "binary:logistic",
                min_child_weight = 50)

p <- xgb.plot.multi.trees(model = bst, feature_names = colnames(agaricus.train$data),
                          features_keep = 3)
print(p)
```

---

# xgb.plot.tree *Plot a boosted tree model*

---

## Description

Read a tree model text dump and plot the model.

## Usage

```
xgb.plot.tree(feature_names = NULL, model = NULL, n_first_tree = NULL,
  plot_width = NULL, plot_height = NULL, ...)
```

### Arguments

| | |
|---|---|
| feature_names | names of each feature as a character vector. Can be extracted from a sparse matrix (see example). If model dump already contains feature names, this argument should be NULL. |
| model | generated by the xgb.train function. Avoid the creation of a dump file. |
| n_first_tree | limit the plot to the n first trees. If NULL, all trees of the model are plotted. Performance can be low for huge models. |
| plot_width | the width of the diagram in pixels. |
| plot_height | the height of the diagram in pixels. |
| ... | currently not used. |

### Details

The content of each node is organised that way:

- feature value;
- cover: the sum of second order gradient of training data classified to the leaf, if it is square loss, this simply corresponds to the number of instances in that branch. Deeper in the tree a node is, lower this metric will be;
- gain: metric the importance of the node in the model.

The function uses GraphViz library for that purpose.

### Value

A DiagrammeR of the model.

### Examples

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

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

xgb.plot.tree(feature_names = colnames(agaricus.train$data), model = bst)
```

---

## xgb.save *Save xgboost model to binary file*

---

### Description

Save xgboost model from xgboost or xgb.train

### Usage

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

### Arguments

| | |
|---|---|
| model | the model object. |
| fname | the name of the file to write. |

### Examples

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2,objective = "binary:logistic")
xgb.save(bst, 'xgb.model')
bst <- xgb.load('xgb.model')
pred <- predict(bst, test$data)
```

---

## xgb.save.raw *Save xgboost model to R's raw vector, user can call xgb.load to load the model back from raw vector*

---

### Description

Save xgboost model from xgboost or xgb.train

### Usage

```
xgb.save.raw(model)
```

### Arguments

| | |
|---|---|
| model | the model object. |

### Examples

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 1, nthread = 2, nrounds = 2,objective = "binary:logistic")
raw <- xgb.save.raw(bst)
bst <- xgb.load(raw)
pred <- predict(bst, test$data)
```

xgb.train *eXtreme Gradient Boosting Training*

## Description

xgb.train is an advanced interface for training an xgboost model. The xgboost function provides a simpler interface.

## Usage

```
xgb.train(params = list(), data, nrounds, watchlist = list(), obj = NULL,
  feval = 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(), ...)

xgboost(data = NULL, label = NULL, missing = NA, weight = NULL,
  params = list(), nrounds, verbose = 1, print_every_n = 1L,
  early_stopping_rounds = NULL, maximize = NULL, save_period = 0,
  save_name = "xgboost.model", xgb_model = NULL, callbacks = list(), ...)
```

## Arguments

params the list of parameters. The complete list of parameters is available at http://xgboost.readthedocs.io/en/latest/parameter.html. Below is a shorter summary:

1. General Parameters

- booster which booster to use, can be gbtree or gblinear. Default: gbtree
- silent 0 means printing running messages, 1 means silent mode. Default: 0

2. Booster Parameters

2.1. Parameter for Tree Booster

- eta control the learning rate: scale the contribution of each tree by a factor of 0 < eta < 1 when it is added to the current approximation. Used to prevent overfitting by making the boosting process more conservative. Lower value for eta implies larger value for nrounds: low eta value means model more robust to overfitting but slower to compute. Default: 0.3
- gamma minimum loss reduction required to make a further partition on a leaf node of the tree. the larger, the more conservative the algorithm will be.
- max_depth maximum depth of a tree. Default: 6
- min_child_weight 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 mode, this simply corresponds to minimum number of instances needed to be in each node. The larger, the more conservative the algorithm will be. Default: 1

- subsample subsample ratio of the training instance. Setting it to 0.5 means that xgboost randomly collected half of the data instances to grow trees and this will prevent overfitting. It makes computation shorter (because less data to analyse). It is advised to use this parameter with eta and increase nround. Default: 1
- colsample_bytree subsample ratio of columns when constructing each tree. Default: 1
- num_parallel_tree Experimental parameter. number of trees to grow per round. Useful to test Random Forest through Xgboost (set colsample_bytree < 1, subsample < 1 and round = 1) accordingly. Default: 1
- monotone_constraints A numerical vector consists of 1, 0 and -1 with its length equals to the number of features in the training data. 1 is increasing, -1 is decreasing and 0 is no constraint.

2.2. Parameter for Linear Booster

- lambda L2 regularization term on weights. Default: 0
- lambda_bias L2 regularization term on bias. Default: 0
- alpha L1 regularization term on weights. (there is no L1 reg on bias because it is not important). Default: 0

3. Task Parameters

- objective specify the learning task and the corresponding learning objective, users can pass a self-defined function to it. The default objective options are below:
  - reg:linear linear regression (Default).
  - reg:logistic logistic regression.
  - binary:logistic logistic regression for binary classification. Output probability.
  - binary:logitraw logistic regression for binary classification, output score before logistic transformation.
  - num_class set the number of classes. To use only with multiclass objectives.
  - multi:softmax set xgboost to do multiclass classification using the softmax objective. Class is represented by a number and should be from 0 to num_class - 1.
  - multi:softprob same as softmax, but prediction outputs a vector of ndata * nclass elements, which can be further reshaped to ndata, nclass matrix. The result contains predicted probabilities of each data point belonging to each class.
  - rank:pairwise set xgboost to do ranking task by minimizing the pairwise loss.
- base_score the initial prediction score of all instances, global bias. Default: 0.5
- eval_metric evaluation metrics for validation data. Users can pass a self-defined function to it. Default: metric will be assigned according to objective(rmse for regression, and error for classification, mean average precision for ranking). List is provided in detail section.

| | |
|---|---|
| data | input dataset. xgb.train takes only an xgb.DMatrix as the input. xgboost, in addition, also accepts matrix, dgCMatrix, or local data file. |
| nrounds | the max number of iterations |
| watchlist | what information should be printed when verbose=1 or verbose=2. Watchlist is used to specify validation set monitoring during training. For example user can specify watchlist=list(validation1=mat1, validation2=mat2) to watch the performance of each round's model on mat1 and mat2 |
| obj | customized objective function. Returns gradient and second order gradient with given prediction and dtrain. |
| feval | custimized evaluation function. Returns list(metric='metric-name', value='metric-value') with given prediction and dtrain. |
| verbose | If 0, xgboost will stay silent. If 1, xgboost will print information of performance. If 2, xgboost will print some additional information. Setting verbose > 0 automatically engages the cb.evaluation.log and cb.print.evaluation callback functions. |
| print_every_n | Print each n-th iteration evaluation messages when verbose>0. Default is 1 which means all messages are printed. This parameter is passed to the cb.print.evaluation callback. |
| early_stopping_rounds | If NULL, the early stopping function is not triggered. If set to an integer k, training with a validation set will stop if the performance doesn't improve for k rounds. Setting this parameter engages the cb.early.stop callback. |
| 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 cb.early.stop callback. |
| save_period | when it is non-NULL, model is saved to disk after every save_period rounds, 0 means save at the end. The saving is handled by the 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 callbacks. 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. |
| ... | other parameters to pass to params. |
| label | vector of response values. Should not be provided when data is a local data file name or an xgb.DMatrix. |
| missing | by default is set to NA, which means that NA values should be considered as 'missing' by the algorithm. Sometimes, 0 or other extreme value might be used to represent missing values. This parameter is only used when input is a dense matrix. |
| weight | a vector indicating the weight for each row of the input. |

## Details

These are the training functions for xgboost.

The xgb.train interface supports advanced features such as watchlist, customized objective and evaluation metric functions, therefore it is more flexible than the xgboost interface.

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

The evaluation metric is chosen automatically by Xgboost (according to the objective) when the eval_metric parameter is not provided. User may set one or several eval_metric parameters. Note that when using a customized metric, only this single metric can be used. The folloiwing is the list of built-in metrics for which Xgboost provides optimized implementation:

- rmse root mean square error. http://en.wikipedia.org/wiki/Root_mean_square_error
- logloss negative log-likelihood. http://en.wikipedia.org/wiki/Log-likelihood
- mlogloss multiclass logloss. https://www.kaggle.com/wiki/MultiClassLogLoss/
- error Binary classification error rate. It is calculated as (# wrong cases) / (# all cases). By default, it uses the 0.5 threshold for predicted values to define negative and positive instances. Different threshold (e.g., 0.) could be specified as "error@0."
- merror Multiclass classification error rate. It is calculated as (# wrong cases) / (# all cases).
- auc Area under the curve. http://en.wikipedia.org/wiki/Receiver_operating_characteristic#'Area_under_curve for ranking evaluation.
- ndcg Normalized Discounted Cumulative Gain (for ranking task). http://en.wikipedia.org/wiki/NDCG

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

- cb.print.evaluation is turned on when verbose > 0; and the print_every_n parameter is passed to it.
- cb.evaluation.log is on when verbose > 0 and watchlist is present.
- cb.early.stop: when early_stopping_rounds is set.
- cb.save.model: when save_period > 0 is set.

## Value

An object of class xgb.Booster with the following elements:

- handle a handle (pointer) to the xgboost model in memory.
- raw a cached memory dump of the xgboost model saved as R's raw type.
- niter number of boosting iterations.
- evaluation_log evaluation history storead as a data.table with the first column corresponding to iteration number and the rest corresponding to evaluation metrics' values. It is created by the cb.evaluation.log callback.
- call a function call.
- params parameters that were passed to the xgboost library. Note that it does not capture parameters changed by the cb.reset.parameters callback.

- `callbacks` callback functions that were either automatically assigned or explicitely passed.
- `best_iteration` iteration number with the best evaluation metric value (only available with early stopping).
- `best_ntreelimit` the `ntreelimit` value corresponding to the best iteration, which could further be used in `predict` method (only available with early stopping).
- `best_score` the best evaluation metric value during early stopping. (only available with early stopping).

## See Also

`callbacks`, `predict.xgb.Booster`, `xgb.cv`

## Examples

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

dtrain <- xgb.DMatrix(agaricus.train$data, label = agaricus.train$label)
dtest <- xgb.DMatrix(agaricus.test$data, label = agaricus.test$label)
watchlist <- list(eval = dtest, train = dtrain)

## A simple xgb.train example:
param <- list(max_depth = 2, eta = 1, silent = 1, nthread = 2,
              objective = "binary:logistic", eval_metric = "auc")
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist)


## 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 either:
#  as 'objective' and 'eval_metric' parameters in the params list:
param <- list(max_depth = 2, eta = 1, silent = 1, nthread = 2,
              objective = logregobj, eval_metric = evalerror)
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist)

#  or through the ... arguments:
param <- list(max_depth = 2, eta = 1, silent = 1, nthread = 2)
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist,
                 objective = logregobj, eval_metric = evalerror)
```

```
#  or as dedicated 'obj' and 'feval' parameters of xgb.train:
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist,
                 obj = logregobj, feval = evalerror)


## An xgb.train example of using variable learning rates at each iteration:
param <- list(max_depth = 2, eta = 1, silent = 1, nthread = 2)
my_etas <- list(eta = c(0.5, 0.1))
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist,
                 callbacks = list(cb.reset.parameters(my_etas)))


## Explicit use of the cb.evaluation.log callback allows to run
## xgb.train silently but still store the evaluation results:
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist,
                 verbose = 0, callbacks = list(cb.evaluation.log()))
print(bst$evaluation_log)

## An 'xgboost' interface example:
bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label,
               max_depth = 2, eta = 1, nthread = 2, nrounds = 2,
               objective = "binary:logistic")
pred <- predict(bst, agaricus.test$data)
```

---

`xgboost-deprecated` *Deprecation notices.*

---

## Description

At this time, some of the parameter names were changed in order to make the code style more uniform. The deprecated parameters would be removed in the next release.

## Details

To see all the current deprecated and new parameters, check the `xgboost:::depr_par_lut` table.

A deprecation warning is shown when any of the deprecated parameters is used in a call. An additional warning is shown when there was a partial match to a deprecated parameter (as R is able to partially match parameter names).

# Index