R documentation

of all in 'man'

July 30, 2020

R topics documented:

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

Do not use `saveRDS` *or* `save` *for long-term archival of models. Instead, use* `xgb.save` *or* `xgb.save.raw`.

Description

It is a common practice to use the built-in `saveRDS` function (or `save`) to persist R objects to the disk. While it is possible to persist `xgb.Booster` objects using `saveRDS`, it is not advisable to do so if the model is to be accessed in the future. If you train a model with the current version of XGBoost and persist it with `saveRDS`, the model is not guaranteed to be accessible in later releases of XGBoost. To ensure that your model can be accessed in future releases of XGBoost, use `xgb.save` or `xgb.save.raw` instead.

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'd like to persist the XGBoost model as part of another R object.

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

# Save as a stand-alone file; load it with xgb.load()
xgb.save(bst, 'xgb.model')
bst2 <- xgb.load('xgb.model')

# Save as a stand-alone file (JSON); load it with xgb.load()
xgb.save(bst, 'xgb.model.json')
bst2 <- xgb.load('xgb.model.json')

# 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().
saveRDS(obj, 'my_object.rds')
# Read back the R object
obj2 <- readRDS('my_object.rds')
# 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

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 environments. 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-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`.

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.gblinear.history` *Callback closure for collecting the model coefficients history of a gblinear booster during its training.*

Description

Callback closure for collecting the model coefficients history of a gblinear booster during its training.

Usage

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

Arguments

`sparse`	when set to FALSE/TURE, 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.

Callback function expects the following values to be set in its calling frame: `bst` (or `bst_folds`).

Value

Results are stored in the `coefs` element of the closure. The `xgb.gblinear.history` convenience function provides an easy way to access it. With `xgb.train`, it is either a dense of a sparse matrix. While with `xgb.cv`, it is a list (an element per each fold) of such matrices.

See Also

callbacks, xgb.gblinear.history.

Examples

```
#### Binary classification:
#
# In the iris dataset, it is hard to linearly separate Versicolor class from the rest
# without considering the 2nd order interactions:
require(magrittr)
x <- model.matrix(Species ~ .^2, iris)[,-1]
colnames(x)
dtrain <- xgb.DMatrix(scale(x), label = 1*(iris$Species == "versicolor"))
param <- list(booster = "gblinear", objective = "reg:logistic", eval_metric = "auc",
              lambda = 0.0003, alpha = 0.0003, nthread = 2)
# For 'shotgun', which is a default linear updater, using high eta 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(param, dtrain, list(tr=dtrain), nrounds = 200, eta = 1.,
                 callbacks = list(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(param, dtrain, list(tr=dtrain), nrounds = 200, eta = 0.8,
                 updater = 'coord_descent', feature_selector = 'thrifty', top_k = 1,
                 callbacks = list(cb.gblinear.history()))
xgb.gblinear.history(bst) %>% matplot(type = 'l')
#  Componentwise boosting is known to have similar effect to Lasso regularization.
# Try experimenting with various values of top_k, eta, nrounds,
# as well as different feature_selectors.

# For xgb.cv:
bst <- xgb.cv(param, dtrain, nfold = 5, nrounds = 100, eta = 0.8,
              callbacks = list(cb.gblinear.history()))
# coefficients in the CV fold #3
xgb.gblinear.history(bst)[[3]] %>% matplot(type = 'l')


#### Multiclass classification:
#
dtrain <- xgb.DMatrix(scale(x), label = as.numeric(iris$Species) - 1)
param <- list(booster = "gblinear", objective = "multi:softprob", num_class = 3,
              lambda = 0.0003, alpha = 0.0003, nthread = 2)
# For the default linear updater 'shotgun' it sometimes is helpful
# to use smaller eta to reduce instability
bst <- xgb.train(param, dtrain, list(tr=dtrain), nrounds = 70, eta = 0.5,
                 callbacks = list(cb.gblinear.history()))
# Will plot the coefficient paths separately for each class:
```

```
xgb.gblinear.history(bst, class_index = 0) %>% matplot(type = 'l')
xgb.gblinear.history(bst, class_index = 1) %>% matplot(type = 'l')
xgb.gblinear.history(bst, class_index = 2) %>% matplot(type = 'l')

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

`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, showsd = TRUE)
```

Arguments

`period`	results would be printed every number of periods
`showsd`	whether standard deviations should be printed (when available)

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 resetting the booster's parameters at each iteration.*

Description

Callback closure for resetting 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 nrounds 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 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)
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.

group can be setup by setinfo but can't be retrieved by getinfo.

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,
  predcontrib = FALSE,
  approxcontrib = FALSE,
  predinteraction = FALSE,
  reshape = FALSE,
  training = 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.
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).

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 either of predleaf, predcontrib, or predinteraction flags is TRUE.

training
: whether is the prediction result used for training. For dart booster, training predicting will perform dropout.

...
: 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, while there are multiple trees per iteration, `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`.

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.

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.

When `predcontrib = TRUE` and it is not a multiclass setting, the output is a matrix object with `num_features + 1` columns. The last "+ 1" column in a matrix corresponds to bias. For a multiclass case, a list of `num_class` elements is returned, where each element is such a matrix. The contribution values are on the scale of untransformed margin (e.g., for binary classification would mean that the contributions are log-odds deviations from bias).

When `predinteraction = TRUE` and it is not a multiclass setting, the output is a 3d array with dimensions `c(nrow,num_features + 1,num_features + 1)`. The off-diagonal (in the last two dimensions) elements represent different features interaction contributions. The array is symmetric

WRT the last two dimensions. The "+ 1" columns corresponds to bias. Summing this array along the last dimension should produce practically the same result as predict with `predcontrib = TRUE`. For a multiclass case, a list of `num_class` elements is returned, where each element is such an array.

References

Scott M. Lundberg, Su-In Lee, "A Unified Approach to Interpreting Model Predictions", NIPS Proceedings 2017, https://arxiv.org/abs/1705.07874

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')
train <- agaricus.train
test <- agaricus.test

bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
               eta = 0.5, nthread = 2, nrounds = 5, objective = "binary:logistic")
# use all trees by default
pred <- predict(bst, test$data)
# use only the 1st tree
pred1 <- predict(bst, test$data, ntreelimit = 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 BIAS
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 <- 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: number of rows in each group (to use with rank:pairwise objective).

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 original xgb.DMatrix object*

Description

Get a new DMatrix containing the specified rows of original 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.serialize 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')
if (file.exists('xgb.model')) file.remove('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.Booster.complete` *Restore missing parts of an incomplete xgb.Booster object.*

Description

It attempts to complete an `xgb.Booster` object by restoring either its missing raw model memory dump (when it has no `raw` data but its `xgb.Booster.handle` is valid) or its missing internal handle (when its `xgb.Booster.handle` is not valid but it has a raw Booster memory dump).

Usage

```
xgb.Booster.complete(object, saveraw = TRUE)
```

Arguments

`object`	object of class `xgb.Booster`
`saveraw`	a flag indicating whether to append `raw` Booster memory dump data when it doesn't already exist.

Details

While this method is primarily for internal use, it might be useful in some practical situations.

E.g., when an `xgb.Booster` model is saved as an R object and then is loaded as an R object, its handle (pointer) to an internal xgboost model would be invalid. The majority of xgboost methods should still work for such a model object since those methods would be using `xgb.Booster.complete` internally. However, one might find it to be more efficient to call the `xgb.Booster.complete` function explicitly once after loading a model as an R-object. That would prevent further repeated implicit reconstruction of an internal booster model.

Value

An object of xgb.Booster class.

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")
saveRDS(bst, "xgb.model.rds")

# Warning: The resulting RDS file is only compatible with the current XGBoost version.
# Refer to the section titled "a-compatibility-note-for-saveRDS-save".
bst1 <- readRDS("xgb.model.rds")
if (file.exists("xgb.model.rds")) file.remove("xgb.model.rds")
# the handle is invalid:
print(bst1$handle)

bst1 <- xgb.Booster.complete(bst1)
# now the handle points to a valid internal booster model:
print(bst1$handle)
```

---

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
value	A JSON string.

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

`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')
nrounds = 4

bst = xgb.train(params = param, data = dtrain, nrounds = nrounds, 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 = nrounds, 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,
  train_folds = NULL,
  verbose = TRUE,
  print_every_n = 1L,
  early_stopping_rounds = NULL,
  maximize = NULL,
  callbacks = list(),
  ...
)
```

Arguments

params	the list of parameters. The complete list of parameters is available in the online documentation. Below is a shorter summary: • objective objective function, common ones are – reg:squarederror Regression with squared loss. – binary:logistic logistic regression for classification. – See xgb.train() for complete list of objectives. • eta step size of each boosting step • max_depth maximum depth of the tree • nthread number of thread used in training, if not set, all threads are used 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: • error binary classification error rate

- rmse Rooted mean square error
- logloss negative log-likelihood function
- auc Area under curve
- aucpr Area under PR 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 — customized 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.

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

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 explicitly passed.
- `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 `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.
- `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 list 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 *Construct xgb.DMatrix object*

Description

Construct xgb.DMatrix object from either a dense matrix, a sparse matrix, or a local file. Supported input file formats are either a libsvm text file or a binary file that was created previously by `xgb.DMatrix.save`).

Usage

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

Arguments

data	a matrix object (either numeric or integer), a dgCMatrix object, or a character string representing a filename.
info	a named list of additional information to store in the xgb.DMatrix object. See setinfo for the specific allowed kinds of
missing	a float value to represents missing values in data (used only when input is a dense matrix). It is useful when a 0 or some other extreme value represents missing values in data.
silent	whether to suppress printing an informational message after loading from a file.
...	the info data could be passed directly as parameters, without creating an info list.

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')
if (file.exists('xgb.DMatrix.data')) file.remove('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')
if (file.exists('xgb.DMatrix.data')) file.remove('xgb.DMatrix.data')
```

---

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"),
  ...
)
```

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. 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 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' 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'
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'))
```

---

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 `cb.gblinear.history()` callback.

Usage

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

Arguments

`model`	either an `xgb.Booster` or a result of `xgb.cv()`, trained using the `cb.gblinear.history()` callback.
`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.

Value

For an `xgb.train` result, a matrix (either dense or sparse) with the columns corresponding to iteration's coefficients (in the order as `xgb.dump()` would return) 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.

`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 https://github.com/aysent/random-forest-leaf-visualization.

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 clustering of the importance values, with bar colors corresponding 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 *Importance of features in a model.*

Description

Creates a data.table of feature importances in a model.

Usage

```
xgb.importance(
  feature_names = NULL,
  model = NULL,
  trees = NULL,
  data = NULL,
  label = NULL,
  target = NULL
)
```

Arguments

feature_names	character vector of feature names. If the model already contains feature names, those would be used when feature_names=NULL (default value). Non-null feature_names could be provided to override those in the model.
model	object of class xgb.Booster.
trees	(only for the gbtree booster) an integer vector of tree indices that should be included into the importance calculation. If set to NULL, all trees of the model are parsed. It could be useful, e.g., in multiclass classification to get feature importances for each class separately. IMPORTANT: the tree index in xgboost models is zero-based (e.g., use trees = 0:4 for first 5 trees).
data	deprecated.
label	deprecated.
target	deprecated.

Details

This function works for both linear and tree models.

For linear models, the importance is the absolute magnitude of linear coefficients. For that reason, in order to obtain a meaningful ranking by importance for a linear model, the features need to be on the same scale (which you also would want to do when using either L1 or L2 regularization).

Value

For a tree model, a data.table with the following columns:

- Features names of the features used in the model;
- Gain represents fractional contribution of each feature to the model based on the total gain of this feature's splits. Higher percentage means a more important predictive feature.
- Cover metric of the number of observation related to this feature;
- Frequency percentage representing the relative number of times a feature have been used in trees.

A linear model's importance data.table has the following columns:

- Features names of the features used in the model;
- Weight the linear coefficient of this feature;
- Class (only for multiclass models) class label.

If feature_names is not provided and model doesn't have feature_names, 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

```
# binomial classification using gbtree:
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(model = bst)

# binomial classification using gblinear:
bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label, booster = "gblinear",
               eta = 0.3, nthread = 1, nrounds = 20, objective = "binary:logistic")
xgb.importance(model = bst)

# multiclass classification using gbtree:
nclass <- 3
nrounds <- 10
mbst <- xgboost(data = as.matrix(iris[, -5]), label = as.numeric(iris$Species) - 1,
               max_depth = 3, eta = 0.2, nthread = 2, nrounds = nrounds,
               objective = "multi:softprob", num_class = nclass)
# all classes clumped together:
xgb.importance(model = mbst)
# inspect importances separately for each class:
xgb.importance(model = mbst, trees = seq(from=0, by=nclass, length.out=nrounds))
xgb.importance(model = mbst, trees = seq(from=1, by=nclass, length.out=nrounds))
xgb.importance(model = mbst, trees = seq(from=2, by=nclass, length.out=nrounds))

# multiclass classification using gblinear:
mbst <- xgboost(data = scale(as.matrix(iris[, -5])), label = as.numeric(iris$Species) - 1,
               booster = "gblinear", eta = 0.2, nthread = 1, nrounds = 15,
               objective = "multi:softprob", num_class = nclass)
xgb.importance(model = mbst)
```

---

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 input file.

Details

The input file is expected to contain a model saved in an xgboost-internal binary format using either xgb.save or cb.save.model 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 xgb.load.

Value

An object of xgb.Booster class.

See Also

xgb.save, xgb.Booster.complete.

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')
if (file.exists('xgb.model')) file.remove('xgb.model')
pred <- predict(bst, test$data)
```

---

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 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,
  trees = NULL,
  use_int_id = FALSE,
  ...
)
```

Arguments

feature_names	character vector of feature names. If the model already contains feature names, those would be used when feature_names=NULL (default value). Non-null feature_names could be provided to override those in the model.
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). text takes precedence over model.
trees	an integer vector of tree indices that should be parsed. If set to NULL, all trees of the model are parsed. It could be useful, e.g., in multiclass classification to get only the trees of one certain class. IMPORTANT: the tree index in xgboost models is zero-based (e.g., use trees = 0:4 for first 5 trees).
use_int_id	a logical flag indicating whether nodes in columns "Yes", "No", "Missing" should be represented as integers (when FALSE) or as "Tree-Node" character strings (when FALSE).
...	currently not used.

Value

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

The columns of the data.table are:

- 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, 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.

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')

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))

# 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(model = 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,
  render = TRUE,
  ...
)
```

Arguments

model	produced by the xgb.train function.
feature_names	names of each feature as a character vector.
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
render	a logical flag for whether the graph should be rendered (see Value).
...	currently not used

Details

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

When render = TRUE: returns a rendered graph object which is an htmlwidget of class grViz. Similar to ggplot objects, it needs to be printed to see it when not running from command line.

When render = FALSE: silently returns a graph object which is of DiagrammeR's class dgr_graph. This could be useful if one wants to modify some of the graph attributes before rendering the graph with render_graph.

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, verbose = 0)

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

## Not run:
# Below is an example of how to save this plot to a file.
# Note that for `export_graph` to work, the DiagrammeRsvg and rsvg packages must also be installed.
library(DiagrammeR)
gr <- xgb.plot.multi.trees(model=bst, features_keep = 3, render=FALSE)
export_graph(gr, 'tree.pdf', width=1500, height=600)

## End(Not run)
```

xgb.plot.shap *SHAP contribution dependency plots*

Description

Visualizing the SHAP feature contribution to prediction dependencies on feature value.

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	data as a matrix or dgCMatrix.
shap_contrib	a matrix of SHAP contributions that was computed earlier for the above data. When it is NULL, it is computed internally using model and data.
features	a vector of either column indices or of feature names to plot. When it is NULL, feature importance is calculated, and top_n high ranked features are taken.
top_n	when features is NULL, top_n [1, 100] most important features in a model are taken.

model	an xgb.Booster model. It has to be provided when either shap_contrib or features is missing.
trees	passed to xgb.importance when features = NULL.
target_class	is only relevant for multiclass models. When it is set to a 0-based class index, only SHAP contributions for that specific class are used. If it is not set, SHAP importances are averaged over all classes.
approxcontrib	passed to predict.xgb.Booster when shap_contrib = NULL.
subsample	a random fraction of data points to use for plotting. When it is NULL, it is set so that up to 100K data points are used.
n_col	a number of columns in a grid of plots.
col	color of the scatterplot markers.
pch	scatterplot marker.
discrete_n_uniq	a maximal number of unique values in a feature to consider it as discrete.
discrete_jitter	an amount parameter of jitter added to discrete features' positions.
ylab	a y-axis label in 1D plots.
plot_NA	whether the contributions of cases with missing values should also be plotted.
col_NA	a color of marker for missing value contributions.
pch_NA	a marker type for NA values.
pos_NA	a relative position of the x-location where NA values are shown: min(x) + (max(x) -min(x)) * pos_NA.
plot_loess	whether to plot loess-smoothed curves. The smoothing is only done for features with more than 5 distinct values.
col_loess	a color to use for the loess curves.
span_loess	the span parameter in loess's call.
which	whether to do univariate or bivariate plotting. NOTE: only 1D is implemented so far.
plot	whether a plot should be drawn. If FALSE, only a lits of matrices is returned.
...	other parameters passed to plot.

Details

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

When plot_loess = TRUE is set, feature values are rounded to 3 significant digits and weighted LOESS is computed and plotted, where weights are the numbers of data points at each rounded value.

Note: SHAP contributions are shown on the scale of model margin. E.g., for a logistic binomial objective, the margin is prediction before a sigmoidal transform into probability-like values. 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` the values of selected features;
- `shap_contrib` the contributions of selected features.

References

Scott M. Lundberg, Su-In Lee, "A Unified Approach to Interpreting Model Predictions", NIPS Proceedings 2017, https://arxiv.org/abs/1705.07874

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')

bst <- xgboost(agaricus.train$data, agaricus.train$label, nrounds = 50,
               eta = 0.1, max_depth = 3, subsample = .5,
               method = "hist", objective = "binary:logistic", nthread = 2, verbose = 0)

xgb.plot.shap(agaricus.test$data, model = bst, features = "odor=none")
contr <- predict(bst, agaricus.test$data, predcontrib = TRUE)
xgb.plot.shap(agaricus.test$data, contr, model = bst, top_n = 12, n_col = 3)

# multiclass example - plots for each class separately:
nclass <- 3
nrounds <- 20
x <- as.matrix(iris[, -5])
set.seed(123)
is.na(x[sample(nrow(x) * 4, 30)]) <- TRUE # introduce some missing values
mbst <- xgboost(data = x, label = as.numeric(iris$Species) - 1, nrounds = nrounds,
                max_depth = 2, eta = 0.3, subsample = .5, nthread = 2,
                objective = "multi:softprob", num_class = nclass, verbose = 0)
trees0 <- seq(from=0, by=nclass, length.out=nrounds)
col <- rgb(0, 0, 1, 0.5)
xgb.plot.shap(x, model = mbst, trees = trees0, target_class = 0, top_n = 4,
              n_col = 2, col = col, pch = 16, pch_NA = 17)
xgb.plot.shap(x, model = mbst, trees = trees0 + 1, target_class = 1, top_n = 4,
              n_col = 2, col = col, pch = 16, pch_NA = 17)
xgb.plot.shap(x, model = mbst, trees = trees0 + 2, target_class = 2, top_n = 4,
              n_col = 2, col = col, pch = 16, pch_NA = 17)
```

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,
  trees = NULL,
  plot_width = NULL,
  plot_height = NULL,
  render = TRUE,
  show_node_id = FALSE,
  ...
)
```

Arguments

feature_names	names of each feature as a character vector.
model	produced by the xgb.train function.
trees	an integer vector of tree indices that should be visualized. If set to NULL, all trees of the model are included. IMPORTANT: the tree index in xgboost model is zero-based (e.g., use trees = 0:2 for the first 3 trees in a model).
plot_width	the width of the diagram in pixels.
plot_height	the height of the diagram in pixels.
render	a logical flag for whether the graph should be rendered (see Value).
show_node_id	a logical flag for whether to show node id's in the graph.
...	currently not used.

Details

The content of each node is organised that way:

- Feature name.
- 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 seen by a split or collected by a leaf during training. The deeper in the tree a node is, the lower this metric will be.
- Gain (for split nodes): the information gain metric of a split (corresponds to the importance of the node in the model).
- Value (for leafs): the margin value that the leaf may contribute to prediction.

The tree root nodes also indicate the Tree index (0-based).

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

This function uses GraphViz as a backend of DiagrammeR.

Value

When `render = TRUE`: returns a rendered graph object which is an `htmlwidget` of class `grViz`. Similar to ggplot objects, it needs to be printed to see it when not running from command line.

When `render = FALSE`: silently returns a graph object which is of DiagrammeR's class `dgr_graph`. This could be useful if one wants to modify some of the graph attributes before rendering the graph with `render_graph`.

Examples

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

bst <- xgboost(data = agaricus.train$data, label = agaricus.train$label, max_depth = 3,
               eta = 1, nthread = 2, nrounds = 2,objective = "binary:logistic")
# plot all the trees
xgb.plot.tree(model = bst)
# plot only the first tree and display the node ID:
xgb.plot.tree(model = bst, trees = 0, show_node_id = TRUE)

## Not run:
# Below is an example of how to save this plot to a file.
# Note that for `export_graph` to work, the DiagrammeRsvg and rsvg packages must also be installed.
library(DiagrammeR)
gr <- xgb.plot.tree(model=bst, trees=0:1, render=FALSE)
export_graph(gr, 'tree.pdf', width=1500, height=1900)
export_graph(gr, 'tree.png', width=1500, height=1900)

## End(Not run)
```

---

xgb.save *Save xgboost model to binary file*

Description

Save xgboost model to a file in binary format.

Usage

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

Arguments

model	model object of xgb.Booster class.
fname	name of the file to write.

Details

This methods allows to save a model in an xgboost-internal binary format which is universal among the various xgboost interfaces. In R, the saved model file could be read-in 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) will 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, xgb.Booster.complete.

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')
if (file.exists('xgb.model')) file.remove('xgb.model')
pred <- predict(bst, test$data)
```

---

xgb.save.raw *Save xgboost model to R's raw vector, user can call xgb.load.raw 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(raw)
pred <- predict(bst, test$data)
```

`xgb.serialize` *Serialize the booster instance into R's raw vector. The serialization method differs from* `xgb.save.raw` *as the latter one saves only the model but not parameters. This serialization format is not stable across different xgboost versions.*

Description

Serialize the booster instance into R's raw vector. The serialization method differs from `xgb.save.raw` as the latter one saves only the model but not parameters. This serialization format is not stable across different xgboost versions.

Usage

```
xgb.serialize(booster)
```

Arguments

`booster`	the booster instance

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.serialize(bst)
bst <- xgb.unserialize(raw)
```

xgb.train *eXtreme Gradient Boosting Training*

Description

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

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 = NULL,
  save_name = "xgboost.model",
  xgb_model = NULL,
  callbacks = list(),
  ...
)
```

Arguments

`params` the list of parameters. The complete list of parameters is available in the online documentation. Below is a shorter summary:

1. General Parameters

- `booster` which booster to use, can be `gbtree` or `gblinear`. Default: `gbtree`.

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 `nrounds`. 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.
- `interaction_constraints` A list of vectors specifying feature indices of permitted interactions. Each item of the list represents one permitted interaction where specified features are allowed to interact with each other. Feature index values should start from `0` (`0` references the first column). Leave argument unspecified for no interaction constraints.

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:squarederror` Regression with squared loss (Default).
  - `reg:squaredlogerror`: regression with squared log loss $1/2*(log(pred+1)-log(label+1))^2$. All inputs are required to be greater than -1. Also, see metric rmsle for possible issue with this objective.
  - `reg:logistic` logistic regression.
  - `reg:pseudohubererror`: regression with Pseudo Huber loss, a twice differentiable alternative to absolute loss.
  - `binary:logistic` logistic regression for binary classification. Output probability.
  - `binary:logitraw` logistic regression for binary classification, output score before logistic transformation.
  - `binary:hinge`: hinge loss for binary classification. This makes predictions of 0 or 1, rather than producing probabilities.
  - `count:poisson`: poisson regression for count data, output mean of poisson distribution. `max_delta_step` is set to 0.7 by default in poisson regression (used to safeguard optimization).
  - `survival:cox`: Cox regression for right censored survival time data (negative values are considered right censored). Note that predictions are returned on the hazard ratio scale (i.e., as HR = exp(marginal_prediction) in the proportional hazard function `h(t) = h0(t) * HR)`.
  - `survival:aft`: Accelerated failure time model for censored survival time data. See Survival Analysis with Accelerated Failure Time for details.
  - `aft_loss_distribution`: Probabilty Density Function used by `survival:aft` and `aft-nloglik` metric.
  - `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.
  - `rank:ndcg`: Use LambdaMART to perform list-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized.
  - `rank:map`: Use LambdaMART to perform list-wise ranking where Mean Average Precision (MAP) is maximized.
  - `reg:gamma`: gamma regression with log-link. Output is a mean of gamma distribution. It might be useful, e.g., for modeling insurance claims severity, or for any outcome that might be gamma-distributed.

  - reg:tweedie: Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any outcome that might be Tweedie-distributed.

- base_score 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 — training dataset. xgb.train accepts only an xgb.DMatrix as the input. xgboost, in addition, also accepts matrix, dgCMatrix, or name of a local data file.

nrounds — max number of boosting iterations.

watchlist — named list of xgb.DMatrix datasets to use for evaluating model performance. Metrics specified in either eval_metric or feval 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 cb.print.evaluation callback is engaged, the performance results are continuously printed out during the training. E.g., specifying watchlist=list(validation1=mat1,validatio allows to track 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 — customized evaluation function. Returns list(metric='metric-name',value='metric-value') with given prediction and dtrain.

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 cb.print.evaluation(period=1) callback function.

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 following 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. http://wiki.fast.ai/index.php/Log_Loss
- 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.
- aucpr Area under the PR curve. https://en.wikipedia.org/wiki/Precision_and_recall 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 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 stored 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 explicitly 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).
- `feature_names` names of the training dataset features (only when column names were defined in training data).
- `nfeatures` number of features in training data.

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

`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(train = dtrain, eval = dtest)

## A simple xgb.train example:
param <- list(max_depth = 2, eta = 1, verbose = 0, 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, verbose = 0, 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, verbose = 0, 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, verbose = 0, nthread = 2,
              objective = "binary:logistic", eval_metric = "auc")
my_etas <- list(eta = c(0.5, 0.1))
bst <- xgb.train(param, dtrain, nrounds = 2, watchlist,
                 callbacks = list(cb.reset.parameters(my_etas)))

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

## 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)
```

---

xgb.unserialize *Load the instance back from* xgb.serialize

Description

Load the instance back from xgb.serialize

Usage

```
xgb.unserialize(buffer)
```

Arguments

buffer	the buffer containing booster instance saved by xgb.serialize

---

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