h2oml rf — Random forest for regression and classification

Description

The `h2oml` *rf* commands implement the random forest method for regression, binary classification, and multiclass classification. `h2oml rfregress` implements random forest regression for continuous responses; `h2oml rfbinclass` implements random forest classification for binary responses; and `h2oml rfmulticlass` implements random forest classification for multiclass responses (categorical responses with more than two categories).

The `h2oml` *rf* commands provide only measures of performance. See [H2OML] **h2oml postestimation** for commands to compute and explain predictions, examine variable importance, and perform other postestimation analyses.

For an introduction to decision trees and random forest, see [H2OML] **Intro**.

Quick start

Before running the `h2oml` *rf* commands, an H2O cluster must be initialized and data must be imported to an H2O frame; see [H2OML] **H2O setup** and *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml**.

Perform random forest regression of response `y1` on predictors `x1` through `x100`

```
h2oml rfregress y1 x1-x100
```

Same as above, but perform classification for binary response `y2`, report measures of fit for the validation frame named `valid`, and set an H2O random-number seed for reproducibility

```
h2oml rfbinclass y2 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but for categorical response `y3` and instead of a validation frame, use 3-fold cross-validation

```
h2oml rfmulticlass y3 x1-x100, cv(3) h2orseed(123)
```

Same as above, but set the number of trees to 30, the maximum tree depth to 10, and the number of predictors to sample to 6

```
h2oml rfmulticlass y3 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) predsampvalue(6)
```

Same as above, but use the default exhaustive grid search to select the optimal number of trees and the maximum tree depth that minimize the log-loss metric

```
h2oml rfmulticlass y3 x1-x100, cv(3) h2orseed(123) predsampvalue(6)  ///
    ntrees(10(5)100) maxdepth(3(1)10)                 ///
    tune(metric(logloss))
```

Same as above, but use a random grid search, set an H2O random-number seed for this search, and limit the maximum search time to 200 seconds

```
h2oml rfmulticlass y3 x1-x100, cv(3) h2orseed(123) predsampvalue(6) ///
    ntrees(10(5)100) maxdepth(3(1)10)                ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200))
```

Same as above, but use early stopping for the grid search with the default stopping log-loss metric

```
h2oml rfmulticlass y3 x1-x100, cv(3) h2orseed(123) predsampvalue(6) ///
    ntrees(10(5)100) maxdepth(3(1)10)                ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200) ///
    stop(5))
```

Menu

Statistics > H2O machine learning

Syntax

Random forest regression

`h2oml rfregress` *response_reg predictors* [, *rfopts*]

Random forest binary classification for binary response

`h2oml rfbinclass` *response_bin predictors* [, *rfopts*]

Random forest multiclass classification for categorical response

`h2oml rfmulticlass` *response_mult predictors* [, *rfopts*]

response_reg, *response_bin*, *response_mult*, and *predictors* correspond to column names of the current H2O frame.

rfopts	Description
Model	
`validframe(`*framename*`)`	specify the name of the H2O frame containing the validation dataset that will be used to evaluate the performance of the model
`cv[ (# [ , `*cvmethod*` ]) ]`	specify the number of folds and method for cross-validation
`cv(`*colname*`)`	specify the name of the variable (H2O column) for cross-validation that identifies the fold to which each observation is assigned
`balanceclasses`	balance the distribution of classes (categories of the response variable) by oversampling minority classes with `h2oml rfbinclass` or `h2oml rfmulticlass`
`h2orseed(#)`	set H2O random-number seed for random forest
`encode(`*encode_type*`)`	specify H2O encoding type for categorical predictors; default is `encode(enum)`
`auc`	enable potentially time-consuming calculation of the area under the curve (AUC) and area under the precision–recall curve (AUCPR) and metrics for multiclass classification with `h2oml rfmulticlass`
`stop[ (# [ , `*stop_opts*` ]) ]`	specify the number of training iterations and other criteria for stopping random forest training if the stopping metric does not improve
`maxtime(#)`	specify the maximum run time in seconds for random forest; by default, no time restriction is imposed
`scoreevery(#)`	specify that metrics be scored after every # trees during training
Hyperparameter	
`ntrees(#|`*numlist*`)`	specify the number of trees to build the random forest model; default is `ntrees(50)`
`maxdepth(#|`*numlist*`)`	specify the maximum depth of each tree; default is `maxdepth(20)`
`minobsleaf(#|`*numlist*`)`	specify the minimum number of observations per child for splitting a leaf node; default is `minobsleaf(1)`
`predsampvalue(#|`*numlist*`)`	specify rules for how to sample predictors; default is `predsampvalue(-1)`
`samprate(#|`*numlist*`)`	specify the sampling rate for randomly selecting a fraction of observations to build a tree; default is `samprate(0.632)`
`minsplitthreshold(#|`*numlist*`)`	specify the threshold for the minimum relative improvement needed for a node split; default is `minsplitthreshold(1e-05)`
`binscat(#|`*numlist*`)`	specify the number of bins to build the histogram for node splits for categorical predictors (`enum` columns in H2O); default is `binscat(1024)`
`binsroot(#|`*numlist*`)`	specify the number of bins to build the histogram for root node splits for continuous predictors (`real` and `int` columns in H2O); default is `binsroot(1024)`
`binscont(#|`*numlist*`)`	specify the number of bins to build the histogram for node splits for continuous predictors (`real` and `int` columns in H2O); default is `binscont(20)`

Tuning

`tune(`*tune_opts*`)`	specify hyperparameter tuning options for selecting the best-performing model

Only one of `validframe()` or `cv[ () ]` is allowed.

If neither `validframe()` nor `cv[ () ]` is specified, the performance metrics are reported for the training dataset.

When *numlist* is specified in one or more hyperparameter options, tuning is performed for those hyperparameters.

`collect` is allowed; see **[U] 11.1.10 Prefix commands**.

See **[U] 20 Estimation and postestimation commands** for more capabilities of estimation commands.

cvmethod	Description
`random`	randomly split the training dataset into folds; the default
`modulo`	evenly split the training dataset into folds using the modulo operation
`stratify`	evenly distribute observations from the different classes of the response to all folds

stop_opts	Description
`metric(`*metric_option*`)`	specify the stopping metric for training or grid search
`tolerance(`#`)`	specify the tolerance value by which a model must improve before the training or grid search stops; default is `tolerance(1e-3)`

tune_opts	Description
`metric(`*metric_option*`)`	specify the metric for selecting the best-performing model
`grid(`*gridspec*`)`	specify whether to perform an exhaustive or random search for all hyperparameter combinations
`maxmodels(`#`)`	specify the maximum number of models considered in the grid search; default is all configurations
`maxtime(`#`)`	specify the maximum run time for the grid search in seconds; default is no time limit
`stop[ (`# `[ , `*stop_opts*` ]) ]`	specify the number of iterations and other criteria for stopping random forest training if the stopping metric does not improve in the grid search
`parallel(`#`)`	specify the number of models to build in parallel during the grid search; default is `parallel(1)`, sequential model building
`nooutput`	suppress the table summarizing hyperparameter tuning

If any of `maxmodels()`, `maxtime()`, or `stop[ () ]` is specified, then `grid(random)` is implied.

Options

Model

`validframe(`*framename*`)` specifies the H2O frame name of the validation dataset used to evaluate the performance of the model. This option is often used when the number of observations is large and the data-splitting approach is the three-way (training-validation-testing) or two-way (training-validation)

holdout method. For definitions of different data-splitting approaches, see *Three-way and two-way holdout method* in [H2OML] **Intro**. If neither `validframe()` nor `cv[()]` is specified, the model is evaluated using the training dataset. Only one of `validframe()` or `cv[()]` may be specified.

`cv(`*cvspec*`)` and `cv` use cross-validation to evaluate model performance. *cvspec* is one of `#[,` *cvmethod* `]` or *colname*. Only one of `cv()` or `validframe()` may be specified.

`cv[(#[,` *cvmethod* `])]` specifies the number of folds for cross-validation and, optionally, the cross-validation method. This option is preferred when the number of observations is small for the training-validation-testing split method.

`cv` is a synonym for `cv(10)`.

cvmethod specifies the cross-validation method and may be one of `random`, `modulo`, or `stratify`.

`random` specifies that training data be randomly split into the specified number of folds. It is recommended for large datasets and may lead to imbalanced folds. This is the default.

`modulo` specifies that a deterministic assignment approach that evenly splits data into the specified number of folds be used. For example, if `cv(3, modulo)` is specified, then training observations 1, 4, 7, … are assigned to fold 1; observations 2, 5, 8, … to fold 2, etc.

`stratify` specifies to try to evenly distribute observations from the different classes of the response across all folds. This approach is useful when the number of classes is large and the available dataset is small. `stratify` is not allowed when the response is H2O type `real`.

`cv(`*colname*`)` specifies the name of the variable (H2O column) that is used to split the data into subsets according to *colname*. It provides a custom grouping index for the cross-validation split. This option is suitable when the data are non-i.i.d. or for comparing different models using cross-validation. The variable should be categorical (H2O data type `enum`).

`balanceclasses` is used with `h2oml rfbinclass` and `h2oml rfmulticlass`. It specifies to oversample the minority classes of the response to balance the class distribution. The imbalanced data can lead to wrong performance evaluation, and oversampling tries to balance data by increasing the minority classes. This can increase the size of the dataset. Minority classes are not oversampled by default.

`h2orseed(#)` sets the H2O random-number seed for H2O model reproducibility of the random forest estimation. This option is not equivalent to the `rseed()` option available with other commands or the `set seed` command. For reproducibility in H2O, see [H2OML] **H2O reproducibility** and H2O's reproducibility page.

`encode(`*encode_type*`)` specifies the H2O encoding type to handle categorical variables, which in H2O are supported as the data type `enum`. See https://www.stata.com/h2o/h2o18/h2oframe_describe.html for information on the H2O data types. *encode_type* may be one of `enum`, `enumfreq`, `onehotexplicit`, `binary`, `eigen`, `label`, or `sortbyresponse`. For details, see [H2OML] ***encode_option***. The default is `encode(enum)`.

`auc` is used with `h2oml rfmulticlass`. It enables calculation of AUC and AUCPR metrics. Because the computation of these metrics requires a large amount of memory and computational cost, by default, H2O does not calculate these metrics. This option must be specified if you plan to use the postestimation command `h2omlestat aucmulticlass` or to use one of these metrics for the early stopping. When the number of classes in the response variable is greater than 50, H2O disables this option.

`stop` and `stop(#` [`, metric(`*metric_option*`) tolerance(#)`]`)` specify the rules for early stopping for random forest. Early-stopping rules help prevent the overfitting of machine learning methods and may reduce the generalization error, which measures how well a model predicts outcome for new data; see *Preliminaries* in [H2OML] **Intro**. `stop(#)` specifies the number of stopping rounds or training iterations needed to stop model training when the selected stopping metric does not improve by `tolerance()`. For example, if `metric(logloss)` is used and the specified number of training iterations is 3, the model will stop training after the performance has been scored three consecutive times without any improvement in `logloss` by the specified `tolerance()`. For reproducibility, it is recommended to use `stop()` with option `scoreevery(#)`.

`stop` is a synonym for `stop(5)`.

`metric(`*metric_option*`)` specifies the metric used for early stopping. The list of allowed metrics is provided in [H2OML] ***metric_option***. The default is `metric(deviance)` for regression and `metric(logloss)` for binary and multiclass classification.

`tolerance(#)` specifies the tolerance value by which `metric()` must improve during training. If the `metric()` does not improve by # after the number of consecutive grid value configurations specified in `stop(#)`, the training stops. The default is `tolerance(1e-3)`.

`maxtime(#)` specifies the maximum run time in seconds for the random forest. No time limitation is imposed by default.

`scoreevery(#)` specifies that metrics be scored after every # trees during model training. This option is useful in combination with `stop()` for reproducibility. When used with early stopping, the specified number of iterations needed to stop applies to the number of scoring iterations that H2O has performed. The default is to use H2O's assessment of a reasonable ratio of training iterations to scoring time, which may not always guarantee reproducibility. For details on reproducibility, see [H2OML] **H2O reproducibility**.

Hyperparameter

When *numlist* is specified in one or more hyperparameter options below, tuning is performed for those hyperparameters.

`ntrees(#|`*numlist*`)` specifies the number of trees to build the model. The default is `ntrees(50)`. The specified number of trees and the actual number of trees used during estimation can differ. This can happen if the early-stopping rules have been specified or the performance of the model is not changing after adding an additional tree.

`maxdepth(#|`*numlist*`)` specifies the maximum depth of each tree. The default is `maxdepth(20)`. The splitting is stopped when the tree's depth reaches the specified number. A deeper tree provides a better training accuracy but may overfit the data.

`minobsleaf(#|`*numlist*`)` specifies the minimum number of observations required for splitting a leaf node. The default is `minobsleaf(1)`. For example, if we specify `minobsleaf(50)`, then the node will split if the training samples in each of the left and right children are at least 50.

`predsampvalue(#|`*numlist*`)` specifies rules for how to sample predictors. The sampling is without replacement. The accepted values are $\{-2, -1\}$ and any integer greater than 1 and less than the number of predictors p. If the default `predsampvalue(-1)` is selected, then in each split, the square root of the number of predictors are sampled for classification and $\lfloor p/3 \rfloor$ are sampled for regression. `predsampvalue(-2)` specifies that all predictors will be used. Finally, for $d > 0$,

`predsampvalue(`*d*`)` indicates that from the total number of predictors, $d \leq p$ will be sampled. `predsampvalue()` reduces the correlation among trees and introduces additional randomness to the estimation method that might improve generalization of the model to new data.

`samprate(`#|*numlist*`)` specifies the sampling rate for the observations. The sampling is without replacement. The sampling rate must be in the range $(0, 1]$. The default is `samprate(0.632)`. The observation sampling introduces an additional randomization to the estimation method that might improve generalization of the model to the new data.

`minsplitthreshold(`#|*numlist*`)` specifies the threshold for the required minimum relative improvement in the impurity measure in order for a split to occur. The default is `minsplitthreshold(1e-05)`. A well-tuned `minsplitthreshold()` increases generalization because it precludes splits that lead to overfitting.

`binscat(`#|*numlist*`)` specifies the number of bins to be included in the histogram for each categorical (H2O type `enum`) predictor. The specified number should be greater than 1. The default is `binscat(1024)`. The histogram is used to split the tree node at the optimal point. Categorical predictors are split by first assigning an integer to each distinct level. Then the method bins the ordered integers according to the specified number of bins. Finally, the optimal split point is selected among the bins. For details, see https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/algo-params/nbins_cats.html. For categorical predictors with many levels, a larger value of `binscat()` leads to overfitting, and a smaller value adds randomness to the split decisions. Therefore, `binscat()` is an important tuning parameter for datasets that contain categorical variables with many levels.

`binsroot(`#|*numlist*`)` specifies the number of bins to use at the root node of each tree for splitting continuous (H2O type `real` or `int`) predictors. For the subsequent nodes, the specified # is divided by 2, and the resulting number is used for splitting. The default is `binsroot(1024)`. This option is used in combination with `binscont()`, which controls the point when the method stops dividing by 2. The histogram is used to split the node at the optimal point. As the tree gets deeper, each subsequent node includes predictors with a smaller range, and the bins are uniformly spread over this range. If the number of observations in a node is smaller than the specified value, then the method creates empty bins. If the number of bins is large, the method evaluates each individual observation as a potential split point, which may increase the computation time. The number specified in `binscont()` must be smaller than the number specified in `binsroot()`.

`binscont(`#|*numlist*`)` specifies the minimum number of bins in the histogram for the continuous (H2O type `real` or `int`) predictors. The default is `binscont(20)`. This option is used in combination with `binsroot()`. The number specified in `binsroot()` must be greater than the number specified in `binscont()`.

In practice, a model is more generalizable to other datasets if `binsroot()` and `binscat()` are small and tends to overfit for large values of `binscont()`, `binsroot()`, and `binscat()`.

Tuning

`tune(`*tune_opts*`)` specifies options for the grid search method for tuning hyperparameters. In machine learning, hyperparameter tuning is an important step in selecting a model that can be generalized to other datasets. Because of the high dimensionality of hyperparameters and their types (continuous, discrete, and categorical), manually setting and testing hyperparameters is time consuming and inefficient. Grid search methods are designed to achieve optimal model performance within specified constraints such as time allocated for tuning or computational resources. Tuning begins with

the selection of the predetermined hyperparameters that you want to tune. Below, we describe the available suboptions for controlling the tuning procedure. *tune_opts* may be `metric()`, `grid()`, `maxmodels()`, `maxtime()`, `stop[()]`, or `nooutput`.

`metric(`*metric_option*`)` specifies the metric for tuning. Allowed metrics are provided in [H2OML] ***metric_option***. The default is `metric(deviance)` for regression and `metric(logloss)` for classification.

`grid(`*gridspec*`)` specifies whether to implement an exhaustive search or a random search for all hyperparameter combinations. *gridspec* is one of `cartesian` or `random[, h2orseed(#)]`.

`grid(cartesian)` implements an exhaustive search for every possible combination in the search space. This approach is recommended if the number of hyperparameters or the search space is small. The default is `grid(cartesian)`.

`grid(random[, h2orseed(#)])` implements a random search for all hyperparameter combinations. It is recommended to use `grid(random)` with `maxmodels()` and `maxtime()` to reduce the computation time. If `maxtime()`, `maxmodels()`, or `stop()` is specified, then `grid(random)` is implied.

`h2orseed(#)` sets an H2O random-number seed for the random grid search for reproducibility. See [H2OML] **H2O reproducibility** and H2O's reproducibility page for details. The behavior of `h2orseed()` is different from the `rseed()` option allowed by many commands and the `set seed` command.

`maxmodels(#)` specifies the maximum number of models to be considered in a grid search. By default, all possible configurations are considered. If this option is specified, `grid(random)` is implied.

`maxtime(#)` specifies the maximum run time for the grid search in seconds. By default, there is no time limitation. If this option is specified, `grid(random)` is implied. This option can be specified with option `maxmodels()` during the grid search. If `maxtime()` is also specified for the model training, then each model building starts with a limit equal to the minimum of the `maxtime()` for the model training, and the remaining time is used for the grid search.

`stop` and `stop(#[, metric(`*metric_option*`) tolerance(#)])` specify the rules for early stopping for the grid search. This option implies `grid(random)`. `stop(#)` specifies the number of grid value configurations needed to stop the grid search when the selected metric does not improve by `tolerance()`. For example, if the selected metric is the default for the binary and multiclass classification (`metric(logloss)`) and we specify `stop(3)`, the grid search will stop after three consecutive grid values chosen by the grid search do not lead to the improvement of the `logloss` by the specified `tolerance()`.

`stop` is a synonym for `stop(5)`.

`metric(`*metric_option*`)` specifies the metric used for early stopping. Allowed metrics are provided in [H2OML] ***metric_option***. The default is `metric(deviance)` for regression and `metric(logloss)` for classification.

`tolerance(#)` specifies the tolerance value by which `metric()` must improve during the grid search. If the `metric()` does not improve by # after the number of consecutive grid value configurations specified in `stop(#)`, the grid search stops. The default is `tolerance(1e-3)`.

`parallel(#)` specifies the number of models to build in parallel during the grid search. This option enables parallel model building, which reduces computational time. The default, `parallel(1)`, specifies sequential model building. `parallel(0)` enables adaptive parallelism, in which the

number of models to be built in parallel is automatically determined by H2O. Any integer greater than 1 specifies the exact number of models to be built in parallel. This option is particularly useful for improving speed when tuning many hyperparameters. However, results for models built in parallel may not be reproducible; see [H2OML] **H2O reproducibility** for details.

`nooutput` suppresses the table summarizing hyperparameter tuning.

Remarks and examples

We assume you have read the introduction to decision trees and ensemble methods in [H2OML] **Intro**.

Remarks are presented under the following headings:

Introduction

Like gradient boosting machine (GBM, see *Introduction* in [H2OML] ***h2oml gbm***), random forest is a machine learning method used for prediction, model selection, and exploring predictor importance. And just like GBM, random forest uses an ensemble of decision trees to alleviate the pitfalls of using a single decision tree. Whereas GBM uses boosting, random forest uses a variation of the so-called bagging procedure.

The bagging procedure, introduced in [H2OML] **Intro**, averages an ensemble of unstable decision trees to reduce the variance in the predictions. Thus, bagging leads to the improvement of the generalization error (a measure of error in using the model to predict in new data) over using a single decision tree. However, this reduction in variance is not substantial if the trees in the ensemble are correlated with each other. For example, if the training data have one strong and several moderately strong predictors, then in the ensemble of bagged decision trees, the majority of the trees will have this strong predictor as one of the first splits. Therefore, most of the bagged trees will have a similar structure, resulting in predictors that are highly correlated.

Random forest (Breiman 2001) is a modification of the bagging procedure that generates an ensemble of decorrelated trees and then averages them. It generates B bootstrap samples of predictors X^b, where $b = 1, 2, \ldots, B$, from the training data. Random forest recursively grows a tree in which, instead of the full set of p predictors, a random sample of m predictors is selected as potential split candidates to generate decorrelated trees. In `h2oml` *rf*, the value of B can be specified by using the `ntrees()` option, and the value of m can be specified by using the `predsampvalue()` option. In practice, $m = \lfloor\sqrt{p}\rfloor$ is recommended for classification and $m = \lfloor p/3\rfloor$ is recommended for regression, where $\lfloor\cdot\rfloor$ is a floor function that rounds a given number down to the nearest integer. These are the default values of m used by `h2oml` *rf* when the `predsamplevalue()` option is not specified. The size of the bootstrap sample X^b controls the bias-variance tradeoff of the random forest. The size can be controlled by using the `samprate()` option to specify the sampling rate (the fraction of observations to be sampled). By default, `samprate()` is set to 0.632.

Depending on the type of response, you can use one of the `h2oml rfregress`, `h2oml rfbinclass`, or `h2oml rfmulticlass` commands to perform random forest. `h2oml rfregress` performs random forest regression for continuous responses. `h2oml rfbinclass` performs random forest binary classification for binary responses. `h2oml rfmulticlass` performs random forest multiclass classification for categorical responses. The commands have many common options. To perform random forest using a validation dataset, you can use the `validframe()` option to specify the name of a validation frame. To perform random forest using cross-validation, you can use the `cv()` option. You can choose between three cross-validation methods for splitting data among folds by specifying the `random`, `modulo`, or `stratify` suboption within the `cv()` option. Alternatively, you can specify a variable in the `cv()` option that defines how observations are split into different folds.

For reproducibility, you can use the `h2orseed()` option to specify a random-number seed for H2O. This option is different from the `rseed()` option available with other commands and the `set seed` command. For early stopping, you can use the `stop[()]` option. We highly recommend that you always specify the `scoreevery()` option with early stopping to ensure reproducibility. For details, see [H2OML] **H2O reproducibility** and H2O's reproducibility page.

Tuning hyperparameters

All `h2oml` *rf* commands provide default values for hyperparameters, but you can also specify your own in the corresponding options. For instance, you can specify the number of trees for random forest in the `ntrees()` option or the predictor sampling value in the `predsampvalue()` option. In practice, however, you would want to tune your random forest model, that is, let the random forest method select the values of the model parameters that correspond to the best-fitting model according to some metric. You can do this by specifying a possible range of grid values for each hyperparameter you intend to tune and controlling the grid search by using the `tune()` option. Currently, `h2oml` *rf* provides two grid search strategies: an exhaustive (Cartesian) grid search with `tune(grid(cartesian))` and a random grid search with `tune(grid(random))`. And several performance metrics are available in `tune(metric())`.

Tuning hyperparameters of the machine learning method is a complex and iterative procedure. Understanding the steps is important for the correct specification of the options provided. A brief overview of these steps is provided below, and a deeper treatment can be found in *Hyperparameter tuning* in [H2OML] **Intro**.

Step 1: Choose the data-splitting approach

Use either a three-way holdout method in which data are separated into training, validation, and testing datasets or, if the number of observations is low, a two-way holdout method (training and testing) with k-fold cross-validation. Recall that the optimal hyperparameters are selected using the results of the metric on the validation set (`validframe()`) or cross-validation (`cv()`), not on the training set.

Step 2: Select the hyperparameters and performance metric

From the list of hyperparameters such as `ntrees()` or `maxdepth()`, select the ones that require tuning for your application. When *numlist* is specified in one or more of the hyperparameter options, tuning is implemented based on the specified grid search suboptions in the `tune()` option. For instance, you can specify the desired performance metric in the `tune(metric())` option; see [H2OML] ***metric_option*** for supported metrics. The default metric is specific to each command. There is no systematic guidance on how many and which hyperparameters to choose: the inclusion of tuning hyperparameters depends on the data, machine learning method, and prior knowledge of the researcher.

The performance metric should be selected carefully because it may affect the estimation results. For example, for the classification problem, if the data are imbalanced, metric `accuracy` is not recommended and a more appropriate metric, such as `aucpr`, is preferred. For more details, see metric options.

Step 3: Select the grid search strategy and search space

If the number of hyperparameters is large, then a random grid search specified via the `tune(grid(random))` option is a better choice than an exhaustive grid search that is performed by default or when the `tune(grid(cartesian))` option is specified. For the first run, it is recommended that you specify a large search space and try to overfit the model. Then, on subsequent runs, you should narrow the search space on high-performance hyperparameters and apply early-stopping rules by specifying the `tune(stop())` option to avoid overfitting.

Step 4: Use the best-performing hyperparameter configuration

Depending on your research problem, use the best-performing hyperparameter configuration to fit the final model on the testing dataset.

Below, we demonstrate the use of options in various applications. In this entry, we focus on the syntax and output of commands. For a more research-focused exposition, see [H2OML] **h2oml**.

Examples of using random forest

In this section, we demonstrate some of the uses of `h2oml` *rf*. Most of the options available in `h2oml` *rf* are also supported in `h2oml` *gbm*. Currently, the only option that `h2oml` *rf* supports but `h2oml` *gbm* does not is `predsampvalue()`. Conversely, the options `loss()`, `monotone()`, `lrate()`, `lratedecay()`, and `predsamprate()` are supported by `h2oml` *gbm* but not by `h2oml` *rf*. If you have already read the examples presented in [H2OML] ***h2oml gbm***, then the discussions of command syntax in the examples below might seem repetitive because the two commands are similar, but we use `h2oml` *rf* instead of the corresponding `h2oml` *gbm* commands in this entry.

The examples are presented under the following headings.

Examples 1 through 4 demonstrate random forest binary classification, but their discussion applies to all `h2oml` *rf* commands. Example 5 demonstrates random forest multiclass classification. Detailed steps for tuning a random forest model are provided in example 10 in [H2OML] **h2oml**.

▷ Example 1: Random forest binary classification using default settings

For demonstration purposes, we start with random forest binary classification using the default settings. In practice, however, you would rarely use the default settings because the performance of the model is improved during training by specifying options that allow optimization or tuning of hyperparameters.

Consider the social pressure dataset, `socialpressure`, borrowed from Gerber, Green, and Larimer (2008), which examines whether social pressure can be used to increase voter turnout in elections in the United States. The data on voting behavior were gathered from Michigan before the August 2006 primary election using a large mailing campaign.

We start by opening the dataset and then putting the data into an H2O frame, Recall that `h2o init` initiates an H2O cluster, `_h2oframe put` loads the current Stata dataset into an H2O frame, and `_h2oframe change` makes the specified frame the current H2O frame. For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/socialpressure
(Social pressure data)
. h2o init
  (output omitted)
. _h2oframe put, into(social)
Progress (%): 0 100
. _h2oframe change social
```

We use random forest binary classification of the response `voted` on predictors `gender`, `g2000`, `g2002`, `p2000`, `p2004`, `treatment`, and `age`, and we specify the `h2orseed(19)` option for reproducibility. For convenience, we introduce a global macro `predictors` that stores the predictors.

```
. global predictors gender g2000 g2002 p2000 p2002 p2004 treatment age
. h2oml rfbinclass voted $predictors, h2orseed(19)
Progress (%): 0 3.9 7.9 36.0 81.9 100
Random forest binary classification using H2O
Response: voted
Frame:                                  Number of observations:
  Training: social                                Training = 229,461
Model parameters
Number of trees      =   50
             actual =   50
Tree depth:                             Pred. sampling value =      -1
           Input max =   20             Sampling rate        =    .632
                 min =   12             No. of bins cat.     =   1,024
                 avg = 18.2             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001
Metric summary
```

Metric	Training
Log loss	.5740521
Mean class error	.3958885
AUC	.6704081
AUCPR	.4669581
Gini coefficient	.3408163
MSE	.1952073
RMSE	.4418227

The header provides information about the model characteristics and data. The `Frame` section contains information about the H2O training frame. In this example, our training frame is `social` with 229,461 observations. The `Model parameters` portion reports the information about hyperparameters. Multiple values are reported for some hyperparameters. For example, there are two values for the number of trees. One reports the number of trees as specified by the user. In our case, it is the default 50. The `actual` value shows the number of trees actually used during training. These numbers may differ when an early stopping rule is applied such as when the `stop()` option is specified. Similarly, for `Tree depth`, there are four values. `Input max` reports the user-specified value, and `min` and `max` report the actual minimum and maximum depths achieved during training. The last two may be different from the default value of 20

because `maxdepth()` enforces a possible maximum depth the tree can achieve, but the method can stop splitting earlier. The `Metric summary` table reports the seven classification performance metrics for the training frame. In general, metrics values are used to compare different models. Depending on whether the method implements regression, binary classification, or multiclass classification, the reported metrics change. For the definition of metrics, see [H2OML] ***metric_option***.

Even though the above output is for binary classification, a similar interpretation applies for regression and multiclass classification using the `h2oml rfregress` and `h2oml rfmulticlass` commands, respectively.

◁

▷ Example 2: Using validation data and early stopping

Example 1 illustrates the simple use of the `h2oml rfbinclass` command. In practice, we want a model that minimizes overfitting. As we discussed in *Model selection in machine learning* in [H2OML] **Intro**, there are two main approaches to check for overfitting: by using a validation dataset or by cross-validation. The former is recommended when the number of observations is large and the latter otherwise (see example 3).

Continuing with example 1, we use the `_h2oframe split` command to randomly split the `social` frame into a training frame (80% of observations) and validation frame (20% of observations), which we named `train` and `valid`, respectively. We also change the current frame to `train`.

```
. _h2oframe split social, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

We now use the `validframe()` option with `h2oml rfbinclass` to specify the validation frame:

```
. h2oml rfbinclass voted $predictors, validframe(valid) h2orseed(19)
Progress (%): 0 14.0 30.0 43.9 56.0 100
Random forest binary classification using H2O
Response: voted
Frame:                                  Number of observations:
  Training:   train                               Training = 183,607
  Validation: valid                             Validation =  45,854
Model parameters
Number of trees      =   50
              actual =   50
Tree depth:                             Pred. sampling value =      -1
           Input max =   20             Sampling rate        =    .632
                 min =   13             No. of bins cat.     =   1,024
                 avg = 18.0             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001
Metric summary
```

Metric	Training	Validation
Log loss	.5744728	.5723461
Mean class error	.3955656	.3970816
AUC	.6696099	.6725455
AUCPR	.4661055	.4700511
Gini coefficient	.3392199	.345091
MSE	.1954345	.1943139
RMSE	.4420798	.4408105

Compared with example 1, the output contains additional information about the validation frame. There are 183,607 training and 45,854 validation observations. The important information here is the performance metrics for the validation frame, the `Validation` column of the `Metric summary` table. The validation frame is used during tuning to select the best model and control for overfitting. See example 10 in [H2OML] **h2oml** and example 5 in [H2OML] ***h2oml gbm*** for tuning.

In some cases, we can greatly improve the generalization of the model, that is, improve model prediction on the new testing dataset, by using early stopping. Early stopping allows you to stop adding trees when the metric computed on the validation sample (or on the cross-validation sample if the `cv[()]` option was specified) does not improve after a prespecified number of iterations. This prevents overfitting. In this example, we use `stop(5)` to halt the training of random forest when the stopping metric does not improve after 5 iterations. By default, the stopping metric is `Log loss`. For reproducibility, we specify the `scoreevery()` option together with the `stop()` option. The `scoreevery()` option controls how frequently the metric score is updated. For example, `scoreevery(1)` means the score is updated after adding each tree to the ensemble. For details, see [H2OML] **H2O reproducibility**.

```
. h2oml rfbinclass voted $predictors, validframe(valid) h2orseed(19)
> stop(5) scoreevery(1)

Progress (%): 0 21.9 100

Random forest binary classification using H2O

Response: voted
Frame:                                  Number of observations:
  Training:   train                               Training = 182,945
  Validation: valid                             Validation =  45,854

Model parameters

Number of trees      =   50
             actual =   12
Tree depth:                             Pred. sampling value =      -1
           Input max =   20             Sampling rate        =    .632
                 min =   13             No. of bins cat.     =   1,024
                 avg = 16.8             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001

Stopping criteria:                      No. of iterations    =       5
  Metric: Log loss                      Tolerance            =    .001

Metric summary
```

Metric	Training	Validation
Log loss	.5771652	.5735485
Mean class error	.4003924	.398497
AUC	.6640448	.6712069
AUCPR	.4583645	.468647
Gini coefficient	.3280896	.3424138
MSE	.1964515	.1948558
RMSE	.4432285	.4414248

```
Note: Metric is scored after every tree.
```

We see several differences compared with the first output in this example. First, as expected, now the actual number of trees is less than the specified number of trees (12 versus 50). In addition, the log-loss metric for both the training frame and validation frame slightly increased, which means early stopping might not be beneficial for the current model.

◁

▷ Example 3: Using cross-validation

In this example, we illustrate the use of `h2oml rfbinclass` with the default parameters and cross-validation.

Continuing with example 2, we keep the frame `train` as our current training data. In the `h2oml` *rf* commands, cross-validation is performed by specifying the `cv()` option. This option supports three methods for folds assignment: random, modulo, and stratified. The random method is the default and is preferred with large datasets. Here, to demonstrate, we use 5-fold cross-validation with `modulo` fold assignment, which assigns each observation to a fold based on the modulo operation. We type

```
. h2oml rfbinclass voted $predictors, cv(5, modulo) h2orseed(19)
Progress (%): 0 10.6 21.3 30.6 39.3 62.0 83.3 83.3 90.6 98.6 100
Random forest binary classification using H2O
Response: voted
Frame:                                  Number of observations:
  Training: train                                 Training = 183,607
                                       Cross-validation = 183,607
Cross-validation: Modulo                Number of folds      =       5
Model parameters
Number of trees      =   50
             actual =   50
Tree depth:                             Pred. sampling value =      -1
           Input max =   20             Sampling rate        =    .632
                 min =   13             No. of bins cat.     =   1,024
                 avg = 18.0             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.5744728	.5741153
Mean class error	.3955656	.396895
AUC	.6696099	.6706381
AUCPR	.4661055	.4675035
Gini coefficient	.3392199	.3412763
MSE	.1954345	.1953061
RMSE	.4420798	.4419344

The output now provides information about the cross-validation assignment method, the number of folds, and, in the second column of the `Metric summary` table, the cross-validated metrics.

The three fold-assignment methods are useful when the data are i.i.d. If the dataset requires a specific grouping for cross-validation, then a new categorical variable can be created and specified in the `cv(`*colname*`)` option. Random forest then uses those variable values to split the data into folds. To demonstrate, in our H2O frame, we generate a new column named `foldvar`, which contains a hypothetical grouping for the fold assignment.

```
. _h2oframe generate foldvar = 1
. _h2oframe replace foldvar = 2 in 20/35
. _h2oframe replace foldvar = 3 in 36/63
. _h2oframe factor foldvar, replace
```

The last command converts the type of `foldvar` into H2O's `enum` type, which is required by the `cv()` option. Now we can perform cross-validation with the fold assignment determined by `foldvar`.

```
. h2oml rfbinclass voted $predictors, cv(foldvar) h2orseed(19)
Progress (%): 0 4.5 20.9 37.0 56.4 75.0 75.0 85.5 97.0 100
Random forest binary classification using H2O
Response: voted
Frame:                                  Number of observations:
  Training: train                                 Training = 183,607
Cross-validation: foldvar                     Cross-validation = 183,607
Model parameters
Number of trees      =   50
             actual =   50
Tree depth:                             Pred. sampling value =      -1
           Input max =   20             Sampling rate        =    .632
                 min =   13             No. of bins cat.     =   1,024
                 avg = 18.0             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.5744728	.6689446
Mean class error	.3955656	.4134973
AUC	.6696099	.6015317
AUCPR	.4661055	.3785627
Gini coefficient	.3392199	.2030635
MSE	.1954345	.2243841
RMSE	.4420798	.473692

Example 4: User-specified hyperparameters

In examples 2 and 3, we used, respectively, validation and cross-validation with default values for all hyperparameters. Continuing with example 2, suppose we now want to try some specific values of several hyperparameters (the number of trees, predictor sampling value, and predictor sampling rate) by including, respectively, the `ntrees(50)`, `predsampvalue(3)`, and `samprate(0.7)` options.

```
. h2oml rfbinclass voted $predictors, cv(5, modulo) h2orseed(19)
> ntrees(50) predsampvalue(3) samprate(0.7)
Progress (%): 0 6.6 15.0 22.3 28.9 41.9 59.6 77.9 83.3 83.3 89.3 95.3 100
Random forest binary classification using H2O

Response: voted
Frame:                                  Number of observations:
  Training: train                                 Training = 183,607
                                             Cross-validation = 183,607
Cross-validation: Modulo                Number of folds      =       5

Model parameters

Number of trees      =   50
              actual =   50
Tree depth:                             Pred. sampling value =       3
           Input max =   20             Sampling rate        =      .7
                 min =   20             No. of bins cat.     =   1,024
                 avg = 20.0             No. of bins root     =   1,024
                 max =   20             No. of bins cont.    =      20
Min. obs. leaf split =    1             Min. split thresh.   =  .00001

Metric summary
```

Metric	Training	Cross-validation
Log loss	.5763545	.57595
Mean class error	.3967958	.3973574
AUC	.6651064	.6650558
AUCPR	.4577942	.4583547
Gini coefficient	.3302127	.3301117
MSE	.1961533	.1961127
RMSE	.442892	.4428462

The output is similar to previous examples, except that it now reports our specified values of 50 for the number of trees, 3 for the predictor sampling value, and 0.7 for the observation sampling rate. Compared with example 3, all validation metrics improved. Although we specified our own parameter values, in practice, these values are typically chosen by performing tuning. For example, see example 10 in [H2OML] **h2oml**.

▷ Example 5: Multiclass classification and model performance

In this example, we show how to implement multiclass classification and which performance metrics to use to measure the performance of the model. For this example, we will use a well-known iris dataset, where the goal is to predict a class of iris plant. This dataset was used in Fisher (1936) and originally collected by Anderson (1935). We start by initializing a cluster, opening the dataset in Stata, and importing the dataset as an H2O frame.

```
. h2o init
  (output omitted)
. use https://www.stata-press.com/data/r19/iris
(Iris data)
. _h2oframe put, into(iris)
```

We then split the data into training and validation frames, with 80% of observations in the training frame, and use the training frame as our current frame.

```
. _h2oframe split iris, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

For convenience, we define a global macro `predictors` to store the names of the predictors. Next we run random forest multiclass classification using 500 trees and default values for other hyperparameters.

```
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, validframe(valid) h2orseed(19)
> ntrees(500)
Progress (%): 0 28.2 61.1 86.5 100
Random forest multiclass classification using H2O
Response: iris                          Number of classes    =      3
Frame:                                  Number of observations:
  Training:   train                                Training =    125
  Validation: valid                              Validation =     25
Model parameters
Number of trees      = 500
             actual = 500
Tree depth:                             Pred. sampling value =     -1
           Input max =  20              Sampling rate        =   .632
                 min =   1              No. of bins cat.     =  1,024
                 avg = 3.4              No. of bins root     =  1,024
                 max =   9              No. of bins cont.    =     20
Min. obs. leaf split =   1              Min. split thresh.   = .00001
Metric summary
------------------------------------------------
           Metric |   Training  Validation
------------------+-----------------------------
        Log loss  |   .1128858    .0952996
 Mean class error |   .0487805     .037037
              MSE |   .0356783    .0307455
             RMSE |   .1888871    .1753439
------------------------------------------------
```

The output is almost identical to the output for the regression we described in detail in examples 1 and 2, except we have different performance metrics.

For computing and reporting AUC and AUCPR metrics after the multiclass classification, see example 6. Even though the example is for the GBM, similar steps apply for the random forest.

◁

Stored results

`h2oml` *rf* stores the following in `e()`:

Scalars

`e(N_train)`	number of observations in the training frame
`e(N_valid)`	number of observations in the validation frame (with option `validframe()`)
`e(N_cv)`	number of observations in the cross-validation (with option `cv()`)
`e(n_cvfolds)`	number of cross-validation folds (with option `cv()`)
`e(k_predictors)`	number of predictors
`e(n_class)`	number of classes (with classification)
`e(n_trees)`	number of trees
`e(n_trees_a)`	actual number of trees used in random forest
`e(maxdepth)`	maximum specified tree depth
`e(depth_min_a)`	achieved minimum tree depth
`e(depth_avg_a)`	achieved average depth among trees
`e(depth_max_a)`	achieved maximum tree depth
`e(minobsleaf)`	minimum specified number of observations for a child leaf
`e(samprate)`	observation sampling rate
`e(predsampvalue)`	predictor sampling value
`e(minsplitthr)`	minimum split improvement threshold
`e(binscat)`	number of bins for categorical predictors
`e(binsroot)`	number of bins for root node
`e(binscont)`	number of bins for continuous predictors
`e(h2orseed)`	H2O random-number seed
`e(auc)`	`1` if `auc`; `0` otherwise (with multiclass classification)
`e(maxtime)`	maximum run time
`e(balanceclass)`	`1` if classes are balanced; `0` otherwise (with classification)
`e(stop_iter)`	maximum iterations before stopping training without metric improvement
`e(stop_tol)`	tolerance for metric improvement before training stops
`e(scoreevery)`	number of trees before scoring metrics during training
`e(tune_h2orseed)`	random-number seed for tuning (with option `tune()`)
`e(tune_stop_iter)`	maximum iterations before stopping tuning without metric improvement (with option `tune()`)
`e(tune_stop_tol)`	tolerance for metric improvement before tuning stops (with option `tune()`)
`e(tune_maxtime)`	maximum run time for tuning grid search (with option `tune()`)
`e(tune_maxmodels)`	maximum number of models considered in tuning grid search (with option `tune()`)

Macros

`e(cmd)`	`h2oml rfregress`, `h2oml rfbinclass`, or `h2oml rfmulticlass`
`e(cmdline)`	command as typed
`e(subcmd)`	`rfregress`, `rfbinclass`, or `rfmulticlass`
`e(method)`	`randomforest`
`e(method_type)`	`regression` or `classification`
`e(class_type)`	`binary` or `multiclass` (with classification)
`e(method_full_name)`	full method name
`e(response)`	name of response
`e(predictors)`	names of predictors
`e(title)`	title in estimation output
`e(train_frame)`	name of the training frame
`e(valid_frame)`	name of the validation frame (with option `validframe()`)
`e(cv_method)`	fold assignment method (with option `cv()`)
`e(cv_varname)`	name of variable identifying cross-validation folds (with option `cv()`)
`e(encode_type)`	encoding type for categorical predictors

e(stop_metric)	stopping metric for training
e(tune_grid)	grid search method used for tuning (with option tune())
e(tune_metric)	name of the tuning metric (with option tune())
e(tune_stop_metric)	stopping metric for tuning (with option tune())
e(properties)	nob noV
e(estat_cmd)	program used to implement h2omlestat
e(predict)	program used to implement h2omlpredict
e(marginsnotok)	predictions disallowed by margins

Matrices

e(metrics)	training, validation, and cross-validation metrics
e(hyperparam_table)	minimum, maximum, and selected hyperparameter values

Methods and formulas

For methods and formulas for random forest implementation, see https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html. For a mapping of h2oml *rf* option names to the H2O options, see [H2OML] **H2O option mapping**.

References

Anderson, E. 1935. The irises of the Gaspé Peninsula. *Bulletin of the American Iris Society* 59: 2–5.

Breiman, L. 2001. Random forests. *Machine Learning* 45: 5–32. https://doi.org/10.1023/A:1010933404324.

Fisher, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals of Eugenics* 7: 179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x.

Gerber, A. S., D. P. Green, and C. W. Larimer. 2008. Social pressure and voter turnout: Evidence from a large-scale field experiment. *American Political Science Review* 102: 33–48. https://doi.org/10.1017/S000305540808009X.

Also see

[H2OML] **h2oml postestimation** — Postestimation tools for h2oml gbm and h2oml rf

[H2OML] **h2oml** — Introduction to commands for Stata integration with H2O machine learning

[H2OML] **h2oml rfbinclass** — Random forest binary classification

[H2OML] **h2oml rfmulticlass** — Random forest multiclass classification

[H2OML] **h2oml rfregress** — Random forest regression

[H2OML] ***h2oml gbm*** — Gradient boosting machine for regression and classification

[U] 20 Estimation and postestimation commands