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

[+] This command includes features that are part of StataNow.

Description | Quick start | Menu | Syntax
Options | Remarks and examples | Stored results | Methods and formulas
References | Also see

Description

The `h2oml` *gbm* commands implement the gradient boosting machine (GBM) method for regression, binary classification, and multiclass classification. `h2oml gbregress` implements gradient boosting regression for continuous and count responses; `h2oml gbbinclass` implements gradient boosting classification for binary responses; and `h2oml gbmulticlass` implements gradient boosting classification for multiclass responses (categorical responses with more than two categories).

The `h2oml` *gbm* 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 GBM, see [H2OML] **Intro**.

Quick start

Before running the `h2oml` *gbm* 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 gradient boosting regression of response `y1` on predictors `x1` through `x100`

```
h2oml gbregress y1 x1-x100
```

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

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

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

As above, but set the number of trees to 30, the maximum tree depth to 10, the learning rate to 0.01, and the predictor sampling rate to 0.6

```
h2oml gbmulticlass y3 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) lrate(0.01) predsamprate(0.6)
```

As above, but for binary response `y2`, and 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 gbbinclass y2 x1-x100, cv(3) h2orseed(123) lrate(0.01)      ///
    predsamprate(0.6) ntrees(10(5)100) maxdepth(3(1)10)       ///
    tune(metric(logloss))
```

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 gbbinclass y2 x1-x100, cv(3) h2orseed(123) lrate(0.01)     ///
    predsamprate(0.6) ntrees(10(5)100) maxdepth(3(1)10)       ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200))
```

As above, but specify a learning-rate decay of 0.9, and tune the number of bins for the categorical and continuous predictors

```
h2oml gbbinclass y2 x1-x100, cv(3) h2orseed(123) lrate(0.01)     ///
    lratedecay(0.9) predsamprate(0.6) ntrees(10(5)100)       ///
    maxdepth(3(1)10) binscont(15(5)50) binscat(500(50)1100)    ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200))
```

As above, but for continuous response y1, and use the mean squared error (MSE) as the metric for early stopping and grid search

```
h2oml gbregress y1 x1-x100, cv(3) h2orseed(123) lrate(0.01)      ///
    lratedecay(0.9) predsamprate(0.6) ntrees(10(5)100)       ///
    maxdepth(3(1)10) binscont(15(5)50) binscat(500(50)1100)    ///
    tune(metric(mse) grid(random, h2orseed(456)) maxtime(200))   ///
    stop(metric(mse))
```

Menu

Statistics > H2O machine learning

Syntax

Gradient boosting regression

`h2oml gbregress` *response_reg predictors* [, *gbmopts*]

Gradient boosting binary classification for binary response

`h2oml gbbinclass` *response_bin predictors* [, *gbmopts*]

Gradient boosting multiclass classification for categorical response

`h2oml gbmulticlass` *response_mult predictors* [, *gbmopts*]

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

gbmopts	Description
Model	
`loss(`*losstype*`)`	specify the loss function with `h2oml gbregress`; default is `loss(gaussian)`
`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 gbbinclass` or `h2oml gbmulticlass`
`h2orseed(#)`	set H2O random-number seed for GBM
`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 gbmulticlass`
`stop[ (# [ , `*stop_opts*` ]) ]`	specify the number of training iterations and other criteria for stopping GBM training if the stopping metric does not improve
`maxtime(#)`	specify the maximum run time in seconds for GBM; by default, no time restriction is imposed
`scoreevery(#)`	specify that metrics be scored after every # trees during training
`monotone(`*predictors*`[ , `*mon_opts*` ])`	specify monotonicity constraints on the relationship between the response and the specified predictors with `h2oml gbregress` or `h2oml gbbinclass`
Hyperparameter	
`ntrees(#|`*numlist*`)`	specify the number of trees to build the GBM model; default is `ntrees(50)`
`lrate(#|`*numlist*`)`	specify the learning rate of each tree; default is `lrate(0.1)`
`lratedecay(#|`*numlist*`)`	specify the rate by which the learning rate specified in `lrate()` is decaying after adding each tree to the GBM; default is `lratedecay(1)`
`maxdepth(#|`*numlist*`)`	specify the maximum depth of each tree; default is `maxdepth(5)`
`minobsleaf(#|`*numlist*`)`	specify the minimum number of observations per child for splitting a leaf node; default is `minobsleaf(10)`
`predsamprate(#|`*numlist*`)`	specify the sampling rate for randomly selecting a fraction of predictors to build a tree; default is `predsamprate(1)`
`samprate(#|`*numlist*`)`	specify the sampling rate for randomly selecting a fraction of observations to build a tree; default is `samprate(1)`

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.

monotone() can be specified with h2oml gbregress only with loss(gaussian), loss(tweedie), or loss(quantile) and with h2oml gbbinclass.

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.

losstype	Description
gaussian	Gaussian loss; the default
tweedie[, power(#)]	Tweedie loss; response must be nonnegative
poisson	Poisson loss; response must be nonnegative
laplace	Laplace loss
huber[, alpha(#)]	Huber loss
quantile[, alpha(#)]	quantile loss

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 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 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 GBM 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

`loss(`*losstype*`)` specifies the loss function for `h2oml gbregress`; see *Introduction*. For `h2oml gbbinclass`, the Bernoulli loss function is used, and for `h2oml gbmulticlass` the multinomial loss function is used.

`loss(gaussian)` specifies the Gaussian loss function. This is the default with `h2oml gbregress`.

`loss(tweedie[ , power(`#`) ])` specifies the Tweedie loss function. This function is useful for modeling a nonnegative response that has exact zeros. The Tweedie loss function is parameterized by the variance power, specified via option `power(`#`)`. `power()` is a number between 1 and 2, exclusive. The default is `power(1.5)`.

`loss(poisson)` specifies the Poisson loss function for a nonnegative response.

`loss(laplace)` specifies the Laplace loss function, which is an absolute loss function. It is useful for predicting the median percentile.

`loss(huber[ , alpha(`#`) ])` specifies the Huber loss function, which is useful when the response has outliers. For the Huber loss function, `alpha()` is a number between 0 and 1, exclusive, and indicates the top percentiles of residuals that should be considered as outliers. The default is `alpha(0.9)`.

`loss(quantile[ , alpha(`#`) ])` specifies the quantile loss function, which is useful for predicting a specified percentile. For the quantile loss function, `alpha()` is a number between 0 and 1, exclusive, that specifies the desired quantile for quantile regression. For example, to predict the 60th percentile of the response conditional on predictors, use `alpha(0.6)`. The default is `alpha(0.5)`, which corresponds to the median.

`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 *The three-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 gbbinclass` and `h2oml gbmulticlass`. 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 GBM 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 gbmulticlass`. 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 GBM. 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 training iterations specified in `stop(`#`)`, the training stops. The default is `tolerance(1e-3)`.

`maxtime(`#`)` specifies the maximum run time in seconds for the GBM. 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**.

`monotone(`*predictors*[`,` *mon_opts*]) imposes a monotonicity constraint on the specified predictors. The data type of *predictors* should be continuous (H2O type `int` or `real`). *mon_opts* can be one of `increasing` or `decreasing`. The default is `increasing`. `monotone()` may be repeated to specify both increasing constraints for some predictors and decreasing constraints for others. For example, `h2oml gbregress ..., monotone(`*predlist1*`, increasing) monotone(`*predlist2*`, decreasing)` would specify an increasing constraint for the first list of predictors and a decreasing constraint for the second list. The option can be used with `h2oml gbbinclass` and `h2oml gbregress` when the loss function is `loss(gaussian)`, `loss(tweedie)`, or `loss(quantile)`. By default, no constraint is imposed.

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.

`lrate(`#|*numlist*`)` specifies the learning rate of the GBM. The specified number must be in the range $(0, 1]$. The relationship between the learning rate and the number of trees is reciprocal: a lower rate requires a larger number of trees and vice versa. A well-tuned learning rate helps avoid overfitting. The default is `lrate(0.1)`.

`lratedecay(#|`*numlist*`)` specifies the factor by which the learning rate will be reduced after adding each tree. The specified number must be in $(0, 1]$. The default is `lratedecay(1)`. For example, with 10 trees, the GBM starts with the learning rate `lrate()`, and the final 10th tree has a learning rate equal to `lrate()` $\times$ `lratedecay()`^{10}. Iteratively decreasing the learning rate implies that trees contain more information (that is, have higher weights) at the beginning than at the end. When the specified number is less than 1, it is recommended to initialize `lrate()` to a larger value, which leads to faster convergence.

`maxdepth(#|`*numlist*`)` specifies the maximum depth of each tree. The default is `maxdepth(5)`. 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(10)`. 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.

`predsamprate(#|`*numlist*`)` specifies the sampling rate for the predictors. The sampling is without replacement. The sampling rate must be in the range $(0, 1]$. The default is `predsamprate(1)`. The predictor sampling rate reduces the correlation among trees and introduces an additional randomness that might improve generalization of the model to the 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(1)`. 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.

stata.com

Remarks and examples

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

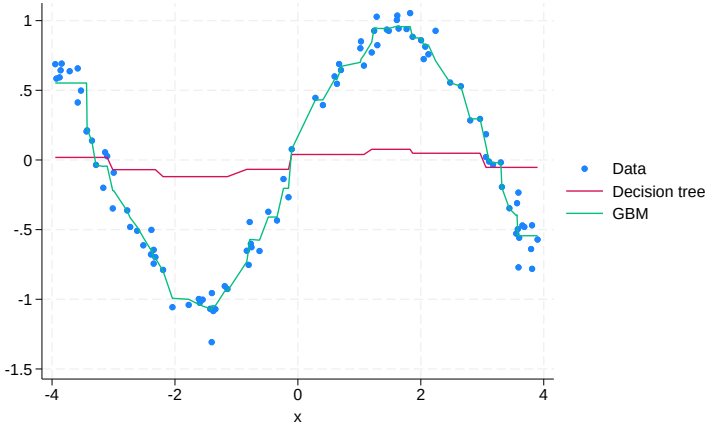
Remarks are presented under the following headings:

Introduction
Tuning hyperparameters
Examples of using GBM
Example 1: Gradient boosting linear regression using default settings
Example 2: Using validation data and early stopping
Example 3: Using cross-validation
Example 4: User-specified hyperparameters
Example 5: Binary classification and hyperparameter tuning
Example 6: Multiclass classification
Example 7: Poisson regression
Example 8: Quantile regression and monotonicity constraint
Example 9: Handling imbalanced data with binary and multiclass classification

Introduction

The GBM (Friedman 2001) is a machine learning method that is useful for prediction, model selection, and explaining the impact of predictors. Even though GBM works with any learner, in H2O it is based on decision trees. A single decision tree is an easily interpretable method for predicting a response; it repeatedly partitions the data into branches based on values of predictors so that responses within each branch are as homogeneous as possible. Despite the advantages, such as interpretability and easy implementation, single decision trees are prone to instability and can struggle to model some types of functions. For example, in the figure below, a single decision tree fails to model simple data generated from the $\sin(x)$ function, where x is generated from a uniform distribution. GBM (Friedman 2001) uses boosting, which fits a series of decision trees that build on each other and gradually increase focus on

observations that are not predicted well by the existing ensemble of decision trees. This boosting process leads to a more stable and better predictive model than a single decision tree. From the figure below, GBM accurately recovers the true data-generation process.



In GBM, boosting can be thought of as a numerical optimization technique that minimizes a given loss function by adding a tree in each stage that best reduces the loss function. The list of loss functions for regression and classification in the `h2oml` *gbm* commands is provided below, where y denotes response and f is a link function.

Loss	$L(y, f)$
Gaussian	$\frac{1}{2}(y-f)^2$
Tweedie(θ)	$2y^{\frac{(2-\theta)}{(1-\theta)(2-\theta)}} - \frac{ye^{f(1-\theta)}}{1-\theta} + \frac{e^{f(2-\theta)}}{2-\theta}$, for $1 < \theta < 2$
Poisson	$-2(yf - e^f)$
Laplace	$\lvert y-f\rvert$
Huber(α)	$(y-f)^2$, for $\lvert y-f\rvert < \alpha$ and $(2\lvert y-f\rvert - \alpha)\alpha$ otherwise
Quantile(α)	$\alpha(y-f)$, for $y > f$ and $(1-\alpha)(f-y)$ otherwise
Bernoulli	$-2(yf - \ln(1+e^f)$
Multinomial	$-\sum_{k=1}^{K} I(y=C_k)f_k + \ln(\sum_{j=1}^{K} e^{f_j})$, where C_k is the kth class

Gaussian, Laplace, Huber, and quantile loss functions use the identity link $E[y|x] = f(x)$. Tweedie, Poisson, and multinomial use the log link function $\log(E[y|x]) = f(x)$. Finally, Bernoulli uses the logit link function $\log(E[y|x]/\{1-E[y|x]\}) = f(x)$. For details about GBM, see *GBM* in [H2OML] **Intro**.

Depending on the type of response, you can use one of the `h2oml gbregress`, `h2oml gbbinclass`, or `h2oml gbmulticlass` commands to perform GBM. `h2oml gbregress` performs gradient boosting regression for continuous and count responses. `h2oml gbbinclass` performs gradient boosting binary classification for binary responses. `h2oml gbmulticlass` performs gradient boosting multiclass classification for categorical responses. In `h2oml gbbinclass` and `h2oml gbmulticlass`, the loss is set to Bernoulli and multinomial, respectively. In `h2oml gbregress`, the `loss()` option is used to specify the loss, which can be one of Gaussian (the default), Tweedie, Poisson, Laplace, Huber, or quantile.

The commands have many common options. To perform GBM using a validation dataset, you can use the `validframe()` option to specify the name of a validation frame. To perform GBM 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 Stata's `rseed()` option 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` *gbm* 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 GBM in the `ntrees()` option or the learning rate of a tree in the `lrate()` option. In practice, however, you would want to *tune* your GBM model, that is, let the GBM 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` *gbm* 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 GBM

In this section, we demonstrate some of the uses of `h2oml` *gbm*. The examples are presented under the following headings.

Example 1: Gradient boosting linear regression using default settings
Example 2: Using validation data and early stopping
Example 3: Using cross-validation
Example 4: User-specified hyperparameters
Example 5: Binary classification and hyperparameter tuning
Example 6: Multiclass classification
Example 7: Poisson regression
Example 8: Quantile regression and monotonicity constraint
Example 9: Handling imbalanced data with binary and multiclass classification

Examples 1 through 4 demonstrate gradient boosting regression, but their discussion applies to all `h2oml` *gbm* commands. Similarly, example 5 demonstrates binary classification, but the steps for tuning hyperparameters are applicable to all commands. Example 6 demonstrates multiclass classification. Examples 7 and 8 show how to specify a different loss function with `h2oml gbregress` to perform Poisson and quantile gradient boosting. Example 8 also shows monotonicity constraints, which can also be accommodated with binary classification. Finally, example 9 shows how to handle imbalanced data during binary classification but is equally applicable to multiclass classification.

▷ Example 1: Gradient boosting linear regression using default settings

For demonstration purposes, we start with gradient boosting linear regression 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.

We start by opening the 1978 automobile data (`auto.dta`) in Stata 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/r18/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

We use gradient boosting linear regression of the response `price` on just a few predictors—`weight`, `length`, and `foreign`—and we specify the `h2orseed(19)` option for reproducibility.

```
. h2oml gbregress price weight length foreign, h2orseed(19)
Progress (%): 0 100
Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training: auto                                  Training =     74

Model parameters

Number of trees      =  50              Learning rate       =     .1
              actual =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   3              No. of bins cat.    =  1,024
                 avg = 3.7              No. of bins root    =  1,024
                 max =   5              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training
Deviance	1692396
MSE	1692396
RMSE	1300.921
RMSLE	.1739734
MAE	893.7925
R-squared	.8027962

The header provides information about the model characteristics and data. Because we used `h2oml gbregress`, the loss is `Gaussian` by default. The `Frame` section contains information about the H2O training frame. In this example, our training frame is `auto` with 74 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 the `Tree depth` there are four values. The `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 5 because `maxdepth()` enforces a possible maximum depth the tree can achieve, but the method can

stop splitting earlier. The `Metric summary` table reports the six regression 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 regression, a similar interpretation applies to binary and multiclass classification using the `h2oml gbbinclass` and `h2oml gbmulticlass` commands, respectively.

◁

▷ Example 2: Using validation data and early stopping

Example 1 illustrates the simple use of the `h2oml gbregress` 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 `auto` frame into a training frame (80% of observations) and validation frame (20% of observations), which we name `train` and `valid`, respectively. We also change the current frame to `train`.

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

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

```
. h2oml gbregress price weight length foreign, h2orseed(19) validframe(valid)
Progress (%): 0 100
Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training:   train                               Training =     63
  Validation: valid                             Validation =     11

Model parameters

Number of trees      =  50              Learning rate       =     .1
             actual  =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   3              No. of bins cat.    =  1,024
                 avg = 3.1              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training	Validation
Deviance	2235364	2391512
MSE	2235364	2391512
RMSE	1495.114	1546.451
RMSLE	.1954448	.2578085
MAE	1013.616	1058.391
R-squared	.7634879	.2253408

Compared with example 1, the output contains additional information about the validation frame. There are 63 training and 11 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 5 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 GBM when the stopping metric does not improve after 5 iterations. By default, the stopping metric is `Deviance`. 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 gbregress price weight length foreign, h2orseed(19) validframe(valid)
> stop(5) scoreevery(1)

Progress (%): 0 100

Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training:   train                               Training =     63
  Validation: valid                             Validation =     11

Model parameters

Number of trees      =  50              Learning rate       =     .1
             actual =  26               Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   3              No. of bins cat.    =  1,024
                 avg = 3.1              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

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

Metric summary

------------------------------------------
    Metric |   Training  Validation
-----------+------------------------------
 Deviance  |    3094539     2288930
      MSE  |    3094539     2288930
     RMSE  |    1759.13    1512.921
    RMSLE  |   .2247564     .251828
      MAE  |   1199.072     1044.42
R-squared  |   .6725832     .2585691
------------------------------------------

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 (26 versus 50). In addition, the RMSE for the training frame increased, and the RMSE for the validation frame decreased from 1546.451 to 1512.921, which means there is less overfitting.

◁

▷ Example 3: Using cross-validation

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

Continuing with example 2, we keep the frame `train` as our current training data. In the `h2oml` *gbm* 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 gbregress price weight length foreign, h2orseed(19) cv(5, modulo)
Progress (%): 0 72.6 99.6 100
Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training: train                                 Training =     63
                                       Cross-validation =     63
Cross-validation: Modulo                Number of folds     =      5

Model parameters

Number of trees      =  50              Learning rate       =     .1
             actual  =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   3              No. of bins cat.    =  1,024
                 avg = 3.1              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training	Cross-validation
Deviance	2235364	3641968
MSE	2235364	3641968
RMSE	1495.114	1908.394
RMSLE	.1954448	.2603751
MAE	1013.616	1391.129
R-squared	.7634879	.6146625

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. GBM 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 gbregress price weight length foreign, h2orseed(19) cv(foldvar)
Progress (%): 0 100
Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training: train                                 Training =     63
Cross-validation: foldvar                    Cross-validation =     63

Model parameters

Number of trees      =  50              Learning rate       =     .1
             actual  =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   3              No. of bins cat.    =  1,024
                 avg = 3.1              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training	Cross-validation
Deviance	2235364	7785926
MSE	2235364	7785926
RMSE	1495.114	2790.327
RMSLE	.1954448	.3791052
MAE	1013.616	1883.424
R-squared	.7634879	.1762122

Example 4: User-specified hyperparameters

In examples 2 and 3, we used validation and cross-validation with default values for all hyperparameters. Continuing with example 3, suppose we now want to try some specific values of several hyperparameters (the number of trees, learning rate, and predictor sampling rate) by including the `ntrees(50)`, `lrate(0.2)`, and `predsamprate(0.7)` options.

```
. h2oml gbregress price weight length foreign, h2orseed(19) cv(5, modulo)
> ntrees(50) lrate(0.2) predsamprate(0.7)
Progress (%): 0 100
Gradient boosting regression using H2O

Response: price
Loss:     Gaussian
Frame:                                  Number of observations:
  Training: train                                 Training =     63
                                             Cross-validation =     63
Cross-validation: Modulo                Number of folds     =      5

Model parameters

Number of trees      =  50              Learning rate       =     .2
              actual =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =     .7
           Input max =   5              Sampling rate       =      1
                 min =   2              No. of bins cat.    =  1,024
                 avg = 3.1              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training	Cross-validation
Deviance	1605800	3398097
MSE	1605800	3398097
RMSE	1267.202	1843.393
RMSLE	.1736271	.2622264
MAE	863.7136	1357.606
R-squared	.8300987	.6404653

The output is similar to previous examples, except that it now reports our specified values of 50 for the number of trees, 0.2 for the learning rate, and 0.7 for the predictor sampling rate.

◁

▷ Example 5: Binary classification and hyperparameter tuning

In example 1 of [H2OML] **h2oml**, we used the `churn` dataset to show steps for building a predictive model to predict whether a customer will churn. In particular, we used a GBM binary classification model with 3-fold stratified cross-validation and the following tuning specification as a baseline model:

```
. h2oml gbbinclass churn $predictors, h2orseed(19) cv(3, stratify)
> ntrees(100) lrate(0.05) predsamprate(0.15)
```
(*output omitted*)

In this example, we demonstrate a process of tuning model parameters to arrive to the model above. As we discussed in *Model selection in machine learning* in [H2OML] **Intro**, the analysis should start by defining the baseline or reference performance. The baseline model has been defined in example 2 of [H2OML] **h2oml**. For simplicity and computational purposes, we will tune only hyperparameters—number of trees and predictor sampling rate—on a small hyperparameter search space. Remember that hyperparameter tuning is an iterative procedure and the considered examples are only for illustration purposes. In practice, you should follow the steps in table 3 in [H2OML] **Intro**.

We read the `churn` dataset as an H2O frame and split it into `train` and `test` H2O frames.

```
. use https://www.stata-press.com/data/r18/churn
(Telco customer churn data)
. h2o init
```
(*output omitted*)
```
. _h2oframe put, into(churn)
Progress (%): 0 100
. _h2oframe change churn
. _h2oframe split churn, into(train test) split(0.8 0.2) rseed(19) replace
. _h2oframe change train
```

Next we create a global macro `predictors` in Stata to store the names of predictors.

```
. global predictors latitude longitude tenuremonths monthlycharges
> totalcharges gender seniorcitizen partner dependents phoneservice
> multiplelines internetserv onlinesecurity onlinebackup streamtv
> techsupport streammovie contract paperlessbill paymethod deviceprotect
```

In the h2oml *gbm* commands, the grid values of a hyperparameter are passed using *numlist* in a hyperparameter option. For example, for the predsamprate() option, we pass a list of numbers $\{0.05, 0.15, 0.25\}$ as *numlist* specification 0.05(0.1)0.25. For the lrate() option, we pass a fixed value of 0.05. As a grid search method for tuning, we use the Cartesian exhaustive search method. We also use the AUCPR metric for tuning.

```
. h2oml gbbinclass churn $predictors, h2orseed(19) cv(3, stratify)
> lrate(0.05) ntrees(50(50)150) predsamprate(0.05(0.1)0.25)
> tune(metric(aucpr))
Progress (%): 0 100

Gradient boosting binary classification using H2O

Response: churn
Loss:     Bernoulli
Frame:                                   Number of observations:
  Training: train                                   Training =  5,643
                                            Cross-validation =  5,643
Cross-validation: Stratify               Number of folds     =      3

Tuning information for hyperparameters

Method: Cartesian
Metric: AUCPR

---------------------------------------------------------------------
                     |                    Grid values
     Hyperparameters |        Minimum         Maximum        Selected
---------------------+-----------------------------------------------
     Number of trees |             50             150             100
 Pred. sampling rate |            .05             .25             .15
---------------------------------------------------------------------

Model parameters

Number of trees      = 100               Learning rate       =    .05
              actual = 100               Learning rate decay =      1
Tree depth:                              Pred. sampling rate =    .15
           Input max =   5               Sampling rate       =      1
                 min =   5               No. of bins cat.    =  1,024
                 avg = 5.0               No. of bins root    =  1,024
                 max =   5               No. of bins cont.   =     20
Min. obs. leaf split =  10               Min. split thresh.  = .00001

Metric summary

------------------------------------------
                  |               Cross-
           Metric |   Training  validation
------------------+-----------------------
         Log loss |   .3531063    .4026141
 Mean class error |   .1784776    .2313897
              AUC |   .8992847    .8565935
            AUCPR |   .7610732     .673929
 Gini coefficient |   .7985693    .7131869
              MSE |   .1126847    .1314475
             RMSE |   .3356854    .3625569
------------------------------------------
```

The output interpretation of h2oml gbbinclass is similar to that of h2oml gbregress. Because we perform binary classification, the Bernoulli loss function is used. Also, the metrics specific to binary classification are reported in the metrics table.

The tuning information is displayed in the header. It includes the tuning method and metric and grid search ranges and the selected values for the hyperparameters. The grid search ranges are the specified minimum and maximum values for hyperparameters. The select values are optimal selected by the algorithm. These are the values we used in our final GBM model in example 3 in [H2OML] **h2oml**.

In this example, we tuned only two hyperparameters and allowed only three possible values for each one, so the grid search was limited to a small space. When the number of hyperparameters and the grid space are large, the grid search can become computationally intensive. You can use the `parallel()` option to specify the number of models to build in parallel during the grid search, thereby improving computational time. However, results for models built in parallel may not be reproducible; see [H2OML] **H2O reproducibility**. By default, the models are built sequentially.

◁

▷ Example 6: Multiclass classification

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/r18/iris
(Iris data)
. _h2oframe put, into(iris)
Progress (%): 0 100
. _h2oframe split iris, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

We use the _h2oframe split command to split the dataset into training and validation frames. Next we run gradient boosting multiclass classification using 500 trees and default values for other hyperparameters.

```
. h2oml gbmulticlass iris seplen sepwid petlen petwid, validframe(valid)
> ntrees(500) h2orseed(19)

Progress (%): 0 9.7 36.8 63.5 90.2 100

Gradient boosting multiclass classification using H2O

Response: iris                          Number of classes   =      3
Loss:     Multinomial
Frame:                                  Number of observations:
  Training:   train                               Training =    125
  Validation: valid                             Validation =     25

Model parameters

Number of trees      = 500              Learning rate       =     .1
              actual = 500              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   1              No. of bins cat.    =  1,024
                 avg = 4.8              No. of bins root    =  1,024
                 max =   5              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training	Validation
Log loss	7.19e-08	1.277958
Mean class error	0	.0740741
MSE	7.52e-14	.0775579
RMSE	2.74e-07	.2784921

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

Two popular metrics to measure the performance after classification are AUC and AUCPR. Their computation may be time consuming, so they are not reported by default. But we can specify the `auc` option to request them.

```
. h2oml gbmulticlass iris seplen sepwid petlen petwid, validframe(valid)
> ntrees(500) h2orseed(19) auc

Progress (%): 0 34.2 43.3 44.6 56.5 100

Gradient boosting multiclass classification using H2O

Response: iris                          Number of classes   =      3
Loss:     Multinomial
Frame:                                  Number of observations:
  Training:   train                               Training =    125
  Validation: valid                             Validation =     25

Model parameters

Number of trees      = 500              Learning rate       =     .1
              actual = 500              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   1              No. of bins cat.    =  1,024
                 avg = 4.8              No. of bins root    =  1,024
                 max =   5              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary

------------------------------------------------
           Metric |   Training  Validation
------------------+-----------------------------
         Log loss |  7.19e-08    1.277958
Mean class error |          0    .0740741
              AUC |          1    .9930556
            AUCPR |          1    .9890377
              MSE |  7.52e-14    .0775579
             RMSE |  2.74e-07    .2784921
------------------------------------------------

Note: AUC and AUCPR computed using macro
      average OVR.
```

The table now reports two additional metrics. From the note, `h2oml gbmulticlass` computes AUC and AUCPR using macro average OVR, which is a uniform weighted average of all AUC scores calculated for each class versus the rest of classes.

With more than two classes, as in this example, you need to decide whether to report AUC and AUCPR based on pairwise combinations of classes or to compare one class with the rest of classes; see [H2OML] ***metric_option*** for definitions of all AUC-based metrics. If you wish to report AUC-based metrics other than the ones reported by `h2oml gbmulticlass`, you can use the `h2omlestat aucmulticlass` postestimation command; see example 1 of [H2OML] **h2omlestat aucmulticlass**.

◁

▷ Example 7: Poisson regression

In example 1, we used the default Gaussian loss function for GBM regression. Depending on the type of response and research problem, we may specify other loss functions. In this example, we consider the data on running shoes for a sample of runners who registered an online running log (Simonoff 1996). Suppose a running-shoe marketing executive is interested in knowing how predictors such as gender, marital status, age, education, income, typical number of runs per week, average miles run per week, and

the preferred type of running explain the number of pairs of running shoes purchased. For this task, we use the GBM with Poisson regression. Because our goal is to simply demonstrate the use of the `loss()` option, we do not tune our model.

We start by initializing the cluster, opening the dataset in Stata, and importing the dataset to an H2O frame.

```
. use https://www.stata-press.com/data/r18/runshoes
(Running shoes)
. h2o init
. _h2oframe put, into(runshoes)
Progress (%): 0 100
. _h2oframe change runshoes
```

To perform a Poisson regression with `h2oml gbregress`, we specify the `loss(poisson)` option.

```
. h2oml gbregress shoes rpweek mpweek male age married trunning, h2orseed(19)
> loss(poisson)
Progress (%): 0 100
Gradient boosting regression using H2O

Response: shoes
Loss:     Poisson
Frame:                                  Number of observations:
  Training: runshoes                              Training =     60

Model parameters

Number of trees      =  50              Learning rate       =     .1
             actual  =  50              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =      1
                 min =   2              No. of bins cat.    =  1,024
                 avg = 2.9              No. of bins root    =  1,024
                 max =   4              No. of bins cont.   =     20
Min. obs. leaf split =  10              Min. split thresh.  = .00001

Metric summary
```

Metric	Training
Deviance	.3649675
MSE	1.064175
RMSE	1.031589
RMSLE	.2691122
MAE	.7149171
R-squared	.4885824

The output is similar to that of `h2oml gbregress` from example 1, but the loss function is `Poisson` here.

For prediction explainability of this model, see example 14 of [H2OML] **h2oml**.

◁

▷ Example 8: Quantile regression and monotonicity constraint

In example 10 of [H2OML] **h2oml**, we used a random forest regression to estimate the conditional mean of house prices. Sometimes, we may be interested in estimating different characteristics of the conditional distribution of house prices other than the mean. Quantile regression, introduced in Koenker and Bassett (1978), predicts conditional quantiles of the response. For an introduction to quantile regression, see Koenker (2005).

In this example, we use GBM quantile regression and the entire `house` dataset without splitting it into training and validation frames. For simplicity, we do not tune hyperparameters and show the model with predetermined values for hyperparameters. These values are borrowed from example 10 of [H2OML] **h2oml**, which are not necessarily optimal for the quantile regression. Before putting the dataset into an H2O frame, we perform some data manipulation in Stata. Because `saleprice` is right-skewed (for example, type `histogram saleprice`), we use its log. We also generate a variable, `houseage`, that calculates the age of the house at the time of a sales transaction.

```
. use https://www.stata-press.com/data/r18/ameshouses
(Ames house data)
. gen logsaleprice = log(saleprice)
. gen houseage = yrsold - yearbuilt
. drop saleprice yearbuilt yrsold
```

The dataset has a total of 46 predictors, but for simplicity we include only 10. We create a global macro, `predictors`, that contains the names of our predictor variables.

```
. global predictors overallqual grlivarea exterqual houseage garagecars
> totalbsmtsf stflrsf garagearea kitchenqual bsmtqual
```

Next we initialize a cluster and put the data into an H2O frame.

```
. h2o init
  (output omitted)
. _h2oframe _put, into(house)
. _h2oframe _change house
```

To perform GBM quantile regression with `h2oml gbmregress`, we specify the `loss(quantile)` option with the `alpha(0.25)` suboption for the desired quantile. We also prespecify values for some hyperparameters.

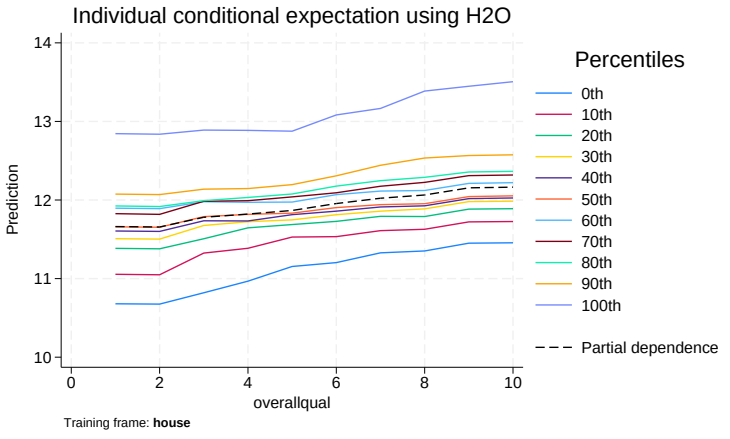
```
. h2oml gbregress logsaleprice $predictors, loss(quantile, alpha(0.25))
> h2orseed(19) ntrees(500) minobsleaf(1) binscat(115) samprate(0.8)

Progress (%): 0 2.4 14.5 34.0 55.1 78.2 100

Gradient boosting regression using H2O

Response: logsaleprice
Loss:     Quantile .25
Frame:                                  Number of observations:
  Training: house                                 Training =  1,460

Model parameters

Number of trees      = 500              Learning rate       =     .1
             actual = 500               Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =     .8
                 min =   5              No. of bins cat.    =    115
                 avg = 5.0              No. of bins root    =  1,024
                 max =   5              No. of bins cont.   =     20
Min. obs. leaf split =   1              Min. split thresh.  = .00001

Metric summary
```

Metric	Training
Deviance	.0256034
MSE	.0145046
RMSE	.1204352
RMSLE	.0092806
MAE	.0773586
R-squared	.9090348

Here, because we estimated the conditional 25th percentile (or 0.25 quantile) of the log price, the header reports the loss as `Quantile .25`.

Sometimes, we may want to impose monotonicity constraints on predictors. For instance, let's use the `h2omlgraph ice` postestimation command to check for monotonicity of the `overallqual` predictor. This command visualizes the relationship between a predictor, `overallqual` in our case, and the predicted response for deciles of the data.

```
. h2omlgraph ice overallqual
```



The relationship between the response and predictor `overallqual` is monotonic for all deciles. Let's impose a monotonicity constraint on this predictor. To apply increasing or decreasing monotonicity constraint, we can use the `monotone()` option.

```
. h2oml gbregress logsaleprice $predictors, loss(quantile, alpha(0.25))
> h2orseed(19) ntrees(500) minobsleaf(1) binscat(155) samprate(0.8)
> monotone(overallqual, increasing)

Gradient boosting regression using H2O

Response: logsaleprice
Loss:     Quantile .25
Frame:                                  Number of observations:
  Training: house                                 Training =  1,460

Model parameters

Number of trees      = 500              Learning rate       =     .1
             actual  = 500              Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   5              Sampling rate       =     .8
                 min =   0              No. of bins cat.    =    155
                 avg = 0.1              No. of bins root    =  1,024
                 max =   5              No. of bins cont.   =     20
Min. obs. leaf split =   1              Min. split thresh.  = .00001

Metric summary

------------------------
    Metric |   Training
-----------+------------
  Deviance |   2.521312
       MSE |   108.0305
      RMSE |   10.39377
     RMSLE |          .
       MAE |   10.08525
 R-squared |  -676.5092
------------------------

Monotone increasing: overallqual
```

The note at the bottom of the table describes specified monotonicity constraints.

The `monotone()` option is available only with `h2oml gbregress` with loss function Gaussian, quantile, or Tweedie and with `h2oml gbbinclass`.

◁

▷ Example 9: Handling imbalanced data with binary and multiclass classification

In this example, we study how to handle imbalanced data in categorical responses such as those having rare events or rare outcomes. We use a popular credit card dataset available in Kaggle (Pozzolo et al. 2015, 2018) to predict whether a given credit card transaction is fraudulent.

The dataset contains 28 predictors `v1` through `v28`, which are obtained after a principal component analysis transformation. Because of confidentiality issues, the original predictors are not available. The response `fraud` is a binary variable that takes value 1 if the transaction is fraudulent and 0 otherwise.

```
. use https://www.stata-press.com/data/r18/creditcard
(Credit card data)
. tabulate fraud
```

Is fraudulent	Freq.	Percent	Cum.
No	284,315	99.83	99.83
Yes	492	0.17	100.00
Total	284,807	100.00	

The data are highly imbalanced. We should practice caution when analyzing such data.

Similar to other examples, we start by converting the dataset in Stata's memory to an H2O frame and splitting it into training and validation frames.

```
. _h2oframe put, into(credit)
Progress (%): 0 2.5 100
. _h2oframe split credit, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

For illustration purposes, we do not implement tuning in this example, but we use 500 trees instead of the default 50. We also specify an H2O random-number seed for reproducibility.

```
. h2oml gbbinclass fraud v1-v28 amount, validframe(valid) h2orseed(19)
> ntrees(500)

Progress (%): 0 0.2 0.4 0.9 4.6 10.0 15.3 21.4 26.6 32.4 38.4 44.4 49.5 56.1
> 62.8 68.4 74.8 81.8 88.5 94.1 100

Gradient boosting binary classification using H2O

Response: fraud
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training:   train                               Training = 228,083
  Validation: valid                             Validation =  56,724

Model parameters

Number of trees      = 500              Learning rate       =      .1
             actual = 500               Learning rate decay =       1
Tree depth:                             Pred. sampling rate =       1
           Input max =   5              Sampling rate       =       1
                 min =   5              No. of bins cat.    =   1,024
                 avg = 5.0              No. of bins root    =   1,024
                 max =   5              No. of bins cont.   =      20
Min. obs. leaf split =  10              Min. split thresh.  =  .00001

Metric summary
```

Metric	Training	Validation
Log loss	.0148732	.0234753
Mean class error	.1043567	.1406525
AUC	.9053009	.8265031
AUCPR	.6773611	.5326735
Gini coefficient	.8106018	.6530063
MSE	.0006575	.0010012
RMSE	.0256412	.0316414

For imbalanced data, the literature (Davis and Goadrich 2006) recommends using AUPCR as the performance metric. For more information about AUCPR and other metrics, see [H2OML] ***metric_option***. The AUCPR on the validation dataset is 0.53. To account for the data imbalance, the `h2oml gbbinclass` and `h2oml gbmulticlass` commands support the `balanceclasses` option, which oversamples the minority class to balance the class distribution. But oversampling may not always be a good solution and may negatively affect machine learning models. You should use the `balanceclasses` option with caution (van den Goorbergh et al. 2022; Sakho, Malherbe, and Scornet 2024).

```
. h2oml gbbinclass fraud v1-v28 amount, validframe(valid) h2orseed(19)
> ntrees(500) balanceclasses
note: balancing distribution of classes per option balanceclasses.

Progress (%): 0 0.4 1.7 2.9 4.8 7.1 9.7 12.2 14.3 16.7 19.4 21.9 23.9 26.6 29.1
> 31.6 33.5 36.1 38.8 41.2 43.2 45.6 48.1 50.5 52.6 55.0 57.5 60.0 62.1 64.6
> 67.1 69.5 72.0 74.4 76.9 79.1 81.5 83.9 86.5 88.8 91.2 93.8 96.2 98.1 100

Gradient boosting binary classification using H2O

Response: fraud
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training:   train                               Training = 455,361
  Validation: valid                             Validation =  56,724

Model parameters

Number of trees      = 500              Learning rate       =      .1
              actual = 500              Learning rate decay =       1
Tree depth:                             Pred. sampling rate =       1
           Input max =   5              Sampling rate       =       1
                 min =   5              No. of bins cat.    =   1,024
                 avg = 5.0              No. of bins root    =   1,024
                 max =   5              No. of bins cont.   =      20
Min. obs. leaf split =  10              Min. split thresh.  =  .00001

Metric summary

------------------------------------------------
          Metric |   Training  Validation
-----------------+------------------------------
        Log loss |  .0108671    .0055343
 Mean class error |        0    .1011677
             AUC |        1    .9716178
           AUCPR |        1    .8094138
 Gini coefficient |        1    .9432356
             MSE |  .0010155    .0004613
            RMSE |  .0318666    .0214785
------------------------------------------------
```

In our case, the AUCPR score improves from 0.53 to 0.81.

◁

Stored results

`h2oml` *gbm* 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 GBM
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(lrate)	learning rate
e(lratedecay)	learning rate decay
e(samprate)	observation sampling rate
e(predsamprate)	predictor sampling rate
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(alpha)	top percentile of residuals if loss(huber); quantile if loss(quantile)
e(power)	variance power if loss(tweedie)
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 gbregress, h2oml gbbinclass, or h2oml gbmulticlass
e(cmdline)	command as typed
e(subcmd)	gbregress, gbbinclass, or gbmulticlass
e(method)	gbm
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(loss)	name of the loss function
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(monotone_inc)	names of predictors with monotone increasing constraints
e(monotone_dec)	names of predictors with monotone decreasing constraints
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 GBM implementation, see https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html. For a mapping of `h2oml` *gbm* 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.

Davis, J., and M. Goadrich. 2006. "The relationship between precision-recall and ROC curves". In *Proceedings of the 23rd International Conference on Machine Learning*, 233–240. New York: Association for Computing Machinery. https://doi.org/10.1145/1143844.1143874.

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.

Friedman, J. H. 2001. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* 29: 1189–1232. https://doi.org/10.1214/aos/1013203451.

Koenker, R. 2005. *Quantile Regression*. New York: Cambridge University Press. https://doi.org/10.1017/CBO9780511754098.

Koenker, R., and G. Bassett, Jr. 1978. Regression quantiles. *Econometrica* 46: 33–50. https://doi.org/10.2307/1913643.

Pozzolo, A. D., G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi. 2018. Credit card fraud detection: A realistic modeling and a novel learning strategy. *IEEE Transactions on Neural Networks and Learning Systems* 29: 3784–3797. https://doi.org/10.1109/tnnls.2017.2736643.

Pozzolo, A. D., O. Caelen, R. A. Johnson, and G. Bontempi. 2015. "Calibrating probability with undersampling for unbalanced classification". In *Proceedings of the IEEE Symposium Series on Computational Intelligence*, 159–166. Piscataway, NJ: IEEE. https://doi.org/10.1109/SSCI.2015.33.

Sakho, A., E. Malherbe, and E. Scornet. 2024. Do we need rebalancing strategies? A theoretical and empirical study around SMOTE and its variants. arXiv:2402.03819 [stat.ML], https://doi.org/10.48550/arXiv.2402.03819.

Simonoff, J. S. 1996. *Smoothing Methods in Statistics*. New York: Springer. https://doi.org/10.1007/978-1-4612-4026-6.

van den Goorbergh, R., M. van Smeden, D. Timmerman, and B. Van Calster. 2022. The harm of class imbalance corrections for risk prediction models: Illustration and simulation using logistic regression. *Journal of the American Medical Informatics Association* 29: 1525–1534. https://doi.org/10.1093/jamia/ocac093.

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 gbbinclass** — Gradient boosting binary classification+

[H2OML] **h2oml gbmulticlass** — Gradient boosting multiclass classification+

[H2OML] **h2oml gbregress** — Gradient boosting regression+

[H2OML] ***h2oml rf*** — Random forest for regression and classification+

[U] 20 Estimation and postestimation commands