MACHINE LEARNING IN STATA USING H2O: ENSEMBLE DECISION TREES REFERENCE MANUAL

RELEASE 19

A Stata Press Publication
StataCorp LLC
College Station, Texas

Copyright © 1985–2025 StataCorp LLC
All rights reserved
Version 19

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845

ISBN-10: 1-59718-451-9
ISBN-13: 978-1-59718-451-9

This manual is protected by copyright. All rights are reserved. No part of this manual may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means—electronic, mechanical, photocopy, recording, or otherwise—without the prior written permission of StataCorp LLC unless permitted subject to the terms and conditions of a license granted to you by StataCorp LLC to use the software and documentation. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document.

StataCorp provides this manual "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. StataCorp may make improvements and/or changes in the product(s) and the program(s) described in this manual at any time and without notice.

The software described in this manual is furnished under a license agreement or nondisclosure agreement. The software may be copied only in accordance with the terms of the agreement. It is against the law to copy the software onto DVD, CD, disk, diskette, tape, or any other medium for any purpose other than backup or archival purposes.

The automobile dataset appearing on the accompanying media is Copyright © 1979 by Consumers Union of U.S., Inc., Yonkers, NY 10703-1057 and is reproduced by permission from CONSUMER REPORTS, April 1979.

Stata, STATA, Stata Press, Mata, MATA, and NetCourse are registered trademarks of StataCorp LLC.

Stata and Stata Press are registered trademarks with the World Intellectual Property Organization of the United Nations.

StataNow and NetCourseNow are trademarks of StataCorp LLC.

Other brand and product names are registered trademarks or trademarks of their respective companies.

For copyright information about the software, type `help copyright` within Stata.

The suggested citation for this software is

StataCorp. 2025. *Stata 19*. Statistical software. StataCorp LLC.

The suggested citation for this manual is

StataCorp. 2025. *Stata 19 Machine Learning in Stata Using H2O: Ensemble Decision Trees Reference Manual* . College Station, TX: Stata Press.

www.stata.com

Contents

Intro . . . . Introduction to machine learning and ensemble decision trees 1
h2oml . . . . Introduction to commands for Stata integration with H2O machine learning 31

H2O setup . . . . Prepare data for H2O analysis in Stata 71

h2oml gbm . . . . Gradient boosting machine for regression and classification 76
h2oml gbbinclass . . . . Gradient boosting binary classification 111
h2oml gbmulticlass . . . . Gradient boosting multiclass classification 117
h2oml gbregress . . . . Gradient boosting regression 123

h2oml rf . . . . Random forest for regression and classification 130
h2oml rfbinclass . . . . Random forest binary classification 150
h2oml rfmulticlass . . . . Random forest multiclass classification 156
h2oml rfregress . . . . Random forest regression 162

h2oml postestimation . . . . Postestimation tools for h2oml gbm and h2oml rf 168

h2omlest . . . . Store and restore H2OML estimation results 177

h2omlestat aucmulticlass . . . . Display AUC and AUCPR after multiclass classification 180
h2omlestat confmatrix . . . . Display confusion matrix 188
h2omlestat cvsummary . . . . Display cross-validation summary 196
h2omlestat gridsummary . . . . Display grid-search summary 202
h2omlestat hitratio . . . . Display hit-ratio table 208
h2omlestat metrics . . . . Display performance metrics 214
h2omlestat threshmetric . . . . Display threshold-based metrics for binary classification 218

h2omlexplore . . . . Explore models after grid search 227

h2omlgof . . . . Compare goodness of fit for machine learning models 232

h2omlgraph ice . . . . Produce individual conditional expectation plot 239
h2omlgraph pdp . . . . Produce partial dependence plot 247
h2omlgraph prcurve . . . . Produce precision–recall curve plot 262
h2omlgraph roc . . . . Produce ROC curve plot 270
h2omlgraph scorehistory . . . . Produce score history plot 279
h2omlgraph shapsummary . . . . Produce SHAP beeswarm plot 286
h2omlgraph shapvalues . . . . Produce SHAP values plot for individual observations 292
h2omlgraph varimp . . . . Produce variable importance plot 303

h2omlpostestframe . . . . Specify frame for postestimation analysis 310

h2omlselect . . . . Select model after grid search 316

h2omltree . . . . Save decision tree DOT file and display rule set 321
DOT extension . . . . Handling DOT files 330

encode_option . . . . Encoding schemes for categorical predictors 335
metric_option . . . . Classification and regression metrics 338

H2O option mapping . . . . Mapping of H2OML estimation options to H2O 349
H2O reproducibility . . . . Reproducibility in H2O 351

Glossary 353
Subject and author index 356

Cross-referencing the documentation

When reading this manual, you will find references to other Stata manuals, for example, **[U] 27 Overview of Stata estimation commands**; [R] **regress**; and [D] **reshape**. The first example is a reference to chapter 27, *Overview of Stata estimation commands*, in the *User's Guide*; the second is a reference to the `regress` entry in the *Base Reference Manual*; and the third is a reference to the `reshape` entry in the *Data Management Reference Manual*.

All the manuals in the Stata Documentation have a shorthand notation:

[GSM]	*Getting Started with Stata for Mac*
[GSU]	*Getting Started with Stata for Unix*
[GSW]	*Getting Started with Stata for Windows*
[U]	*Stata User's Guide*
[R]	*Stata Base Reference Manual*
[ADAPT]	*Stata Adaptive Designs: Group Sequential Trials Reference Manual*
[BAYES]	*Stata Bayesian Analysis Reference Manual*
[BMA]	*Stata Bayesian Model Averaging Reference Manual*
[CAUSAL]	*Stata Causal Inference and Treatment-Effects Estimation Reference Manual*
[CM]	*Stata Choice Models Reference Manual*
[D]	*Stata Data Management Reference Manual*
[DSGE]	*Stata Dynamic Stochastic General Equilibrium Models Reference Manual*
[ERM]	*Stata Extended Regression Models Reference Manual*
[FMM]	*Stata Finite Mixture Models Reference Manual*
[FN]	*Stata Functions Reference Manual*
[G]	*Stata Graphics Reference Manual*
[H2OML]	*Machine Learning in Stata Using H2O: Ensemble Decision Trees Reference Manual*
[IRT]	*Stata Item Response Theory Reference Manual*
[LASSO]	*Stata Lasso Reference Manual*
[XT]	*Stata Longitudinal-Data/Panel-Data Reference Manual*
[META]	*Stata Meta-Analysis Reference Manual*
[ME]	*Stata Multilevel Mixed-Effects Reference Manual*
[MI]	*Stata Multiple-Imputation Reference Manual*
[MV]	*Stata Multivariate Statistics Reference Manual*
[PSS]	*Stata Power, Precision, and Sample-Size Reference Manual*
[P]	*Stata Programming Reference Manual*
[RPT]	*Stata Reporting Reference Manual*
[SP]	*Stata Spatial Autoregressive Models Reference Manual*
[SEM]	*Stata Structural Equation Modeling Reference Manual*
[SVY]	*Stata Survey Data Reference Manual*
[ST]	*Stata Survival Analysis Reference Manual*
[TABLES]	*Stata Customizable Tables and Collected Results Reference Manual*
[TS]	*Stata Time-Series Reference Manual*
[I]	*Stata Index*
[M]	*Mata Reference Manual*

Description Remarks and examples References Also see

Description

Machine learning methods are commonly used to solve various research and business problems. These methods can be used to predict the probability of a patient having a disease based on their symptoms, forecast customer churn for the coming year, determine whether a customer is likely to default on a loan based on their background characteristics, predict changes in house prices in the coming month, and identify important factors in predicting the outcome of an election. And these are just a few examples. These types of problems often require more sophisticated modeling approaches than, for instance, a linear regression or generalized linear models. Ensemble decision tree methods, which combine multiple decision trees to improve model predictive performance, have emerged as some of the more popular and more effective methods for solving such problems because they perform well in practice (Shmuel, Glickman, and Lazebnik 2024; Shwartz-Ziv and Armon 2022; and Borisov et al. 2024).

This entry provides a software-free introduction to ensemble decision tree methods. In particular, we focus on two popular methods: gradient boosting machine (GBM) and random forest. See [H2OML] **h2oml** for the Stata implementation.

Remarks and examples

Remarks are presented under the following headings:

Why machine learning?
Preliminaries
Fundamentals of machine learning
Decision trees
  Classification trees
  Regression trees
  Pros and cons of decision trees
Ensemble methods
  Bagging
  Random forest
  Boosting
  GBM
  Trees with monotonicity constraints
Model selection in machine learning
  Three-way and two-way holdout methods
  k-fold cross-validation
  Hyperparameter tuning
  Method comparison
Interpretation and explanation
  Global surrogate models

Why machine learning?

Linear and generalized linear models are among the most widely used models in various fields. However, they may not always capture more complex patterns in the data well and thus may lead to poor prediction. As an example, consider a fictional dataset used to predict employee attrition based on salary and performance. Figure 1 provides the scatterplot of the data, with blue dots representing employees who stayed with the company and red dots representing those who left.

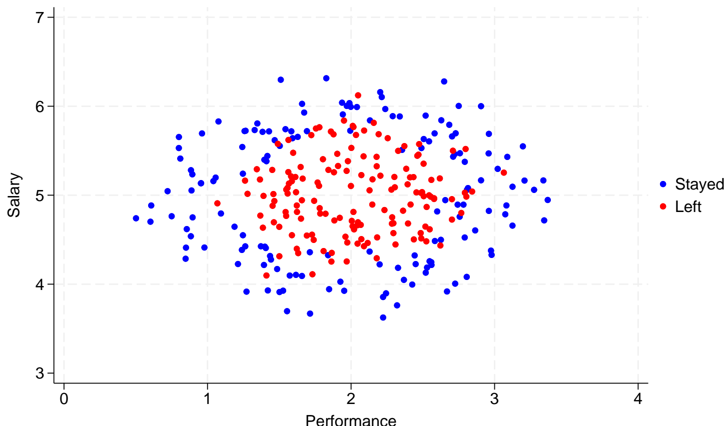


Figure 1.

The data-generating mechanism is complex, and there is no one line that can separate the blue and red dots. That is, the dataset is not linearly separable. To illustrate this point further, figure 2 shows the decision surface, the predicted attrition based on performance and salary, for the logistic regression. It predicts that an employee will leave (`attrition` = 1) for observations on the orange surface and that an employee will stay for observations on the light-blue surface. As we can see, the linear decision boundary misclassifies many blue dots as red and vice versa.

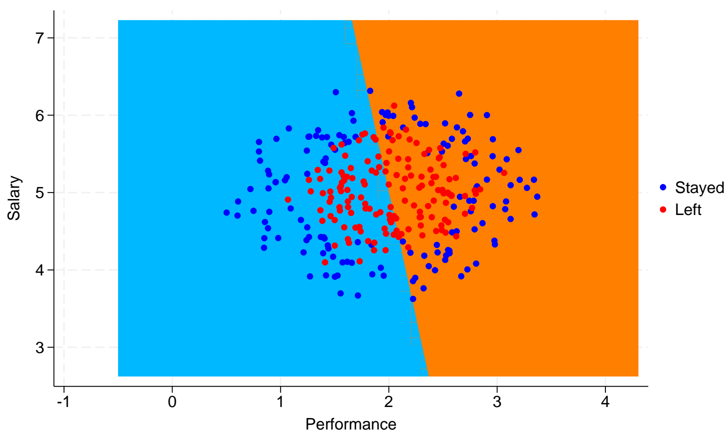


Figure 2. Logistic regression decision surface

On the other hand, machine learning methods can capture the complex structure better. Figure 3 displays the decision surface for the random forest. Here we can easily see that the random forest performs much better, with predictions more closely matching the observed attrition values.

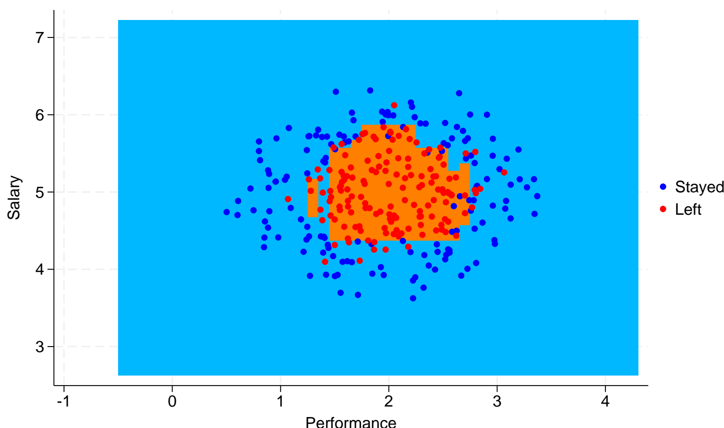


Figure 3. Random forest decision surface

Preliminaries

Before describing ensemble decision trees, we introduce the machine learning terminology that we will use throughout this manual.

Predictors. The inputs for a machine learning model. In classical statistics, these may be referred to as independent variables, covariates, x variables, or predictors. In the machine learning literature, they are also referred to as features.

Responses. The outputs for a machine learning model. In classical statistics, these may be referred to as dependent variables, y variables, or outcomes. In the machine learning literature, they are also referred to as targets.

Learning, **training**. In the machine learning context, learning refers to the process when a model uses data to adjust its parameters to increase prediction accuracy.

Learner. A model that is used for learning.

Supervised learning. A type of machine learning in which a method is trained on data where there is an associated response for each observation.

Unsupervised learning. A type of machine learning where there is no response variable.

Hyperparameter. A parameter whose value is adjusted to control and improve a training process.

Tuning. A process where the hyperparameters of a model are optimized to improve model performance.

Training data. A subset of the data that a model uses to learn.

Validation data. A subset of the data used to evaluate model performance during training as hyperparameters change.

Testing data. A subset of the data that is used to evaluate the performance of a trained model.

Performance metric. A quantitative measure or metric used to evaluate model performance.

Hyperparameter space. Possible values and ranges of the hyperparameters.

Grid search. A process of evaluating different hyperparameter configurations in the hyperparameter space to find the best configuration that improves model performance.

Generalization. A concept that a model performs well not only on the training data but also on the new (testing) data.

Generalization error, **test error**. A quantitative measure of how well a machine learning model can predict responses for new (testing) data.

Overfitting. Fitting a model too well to the training data.

Metric scoring. A process of evaluating the performance of a machine learning method by using a specified performance metric.

In a typical machine learning scenario, the goal is to predict a response based on a set of predictors. To achieve this goal, a researcher uses training data to build (or train) a prediction model. A good model, or learner, is one that accurately predicts the response for new or testing data and minimizes a generalization error or test error. A generalization error of a learning model is a quantitative measure of how well a machine learning model can predict responses for new data or, more formally, an expected error on any testing data sampled from the data-generating distribution. In other words, the focus is on predictive modeling, which is the process of "developing a mathematical tool or model that generates accurate prediction" (Kuhn and Johnson 2013). Intuitively, success in predictive modeling depends on finding a model that 1) has low generalization error, 2) is simple, and 3) can be used on a sufficiently large training dataset.

Most machine learning problems can be divided into two categories: supervised learning and unsupervised learning. In supervised learning, there is an associated response for each observation of the predictors. Most types of regression and many tree-based methods are examples of supervised learning. In contrast, in unsupervised learning, there is no response variable, and only the predictors are observed. Cluster analysis is an example of unsupervised learning.

In what follows, we provide a more technical introduction to machine learning, including decision trees and ensemble decision trees. For a brief and more gentle exposition of a machine learning workflow by using the `h2oml` command, see *h2oml in a nutshell* in [H2OML] **h2oml**.

Fundamentals of machine learning

One of the main issues in machine learning, also known as a fundamental problem of machine learning (Chollet 2021), is balancing learning and generalization. Recall that learning refers to the process of adjusting a model to achieve the best performance on the training data, whereas generalization refers to evaluating the performance of the model on the data it has never seen before such as the testing data. Unfortunately, generalization cannot be fully controlled by a researcher because we observe only the training data, and overfitting (fitting a model too well on the training data) can hurt the generalization of the model. This is why it is important to "mimic" the presence of testing data by splitting the observed training data, as we discuss in *Three-way and two-way holdout methods*.

The tradeoff between learning and generalization is related to the well-known bias–variance tradeoff, where the aim is to lower the generalization error by reducing the bias and variance of the proposed method. Suppose we have a supervised learning problem, where the relationship between predictors and the response is described by some unknown function $f(\cdot)$ plus an additive error,

$$y_i = f(\mathbf{x}_i) + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where $E(\varepsilon_i) = 0$ and $\mathrm{Var}(\varepsilon_i) = \sigma^2$.

The goal is to estimate $f(\cdot)$ by $\widehat{f}(\cdot)$ using a specific machine learning method on training data. However, if we use different training data, the learned $\widehat{f}(\cdot)$ is likely to be different. The amount by which $\widehat{f}(\cdot)$ changes as we use different training data is the variance. Machine learning methods, like other statistical estimation methods, often introduce bias because they typically impose simplifying assumptions during the estimation of $f(\cdot)$.

The generalization error for training data $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ and test observation $(\mathbf{x}, y)$, sampled from the data-generating distribution, can be written as the sum of the error variance and the squared bias and the variance of the estimate:

$$E_{(\mathbf{x},y,D)}\left[\left\{\widehat{f}(\mathbf{x}) - f(\mathbf{x})\right\}^2\right] = \sigma^2 + \mathrm{Bias}^2\{\widehat{f}(\mathbf{x})\} + \mathrm{Var}\{\widehat{f}(\mathbf{x})\}$$

The error variance σ^2 is inherited from the data and cannot be reduced. However, the bias, which is the average difference between $\widehat{f}(\cdot)$ and $f(\cdot)$, is a result of underfitting and can be reduced. And the variance, which is inextricably linked to overfitting, where the model fits the training data too well and thus the variance of the model increases for new data, can also be reduced. Thus, an ideal machine learning method reduces the bias without increasing the variance or reduces the variance without increasing the bias. In practice, decreasing one will necessarily increase the other, so the preferred method strives to achieve the best tradeoff between the bias and the variance.

Consider a hypothetical example below that shows two methods, `Method 1` and `Method 2`. The red points correspond to the training data and blue points to the testing data. From the left graph, `Method 2` predicts the training points very well with possibly small bias and mean squared error (MSE). However, compared with `Method 1`, the prediction of `Method 2` deteriorates on the testing data because of the high variance. `Method 2` predicts the testing data poorly because it overfits the training data.

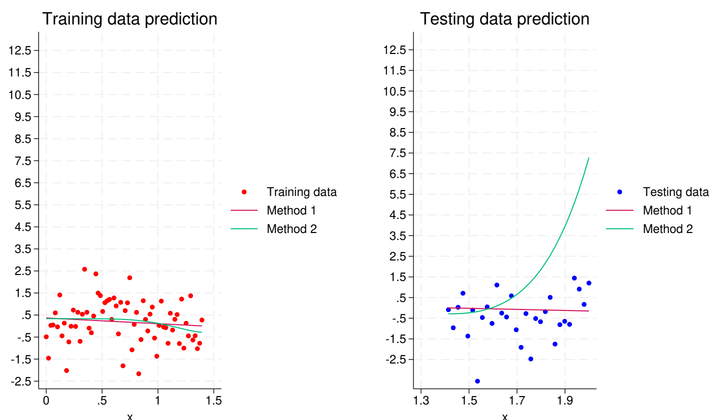


Figure 4.

The above example demonstrated the generalization of machine learning methods in just one dimension. In general, the ability of these methods, such as ensemble decision trees, to generalize well to high-dimensional data can be explained by the so-called manifold hypothesis (Chollet 2021; Wyner et al. 2017 ; and Belkin et al. 2019). According to this hypothesis, the observed high-dimensional data can be approximated by a low-dimensional manifold, or subspace. Informally, this means that a complex structure of the high-dimensional data can be represented by a simpler, lower-dimensional structure, which machine learning methods tend to capture well.

Decision trees

Decision trees are versatile and powerful supervised machine learning methods that can be used for both regression and classification. Decision trees repeatedly partition the data based on values of the predictors by asking a series of Boolean-type (“yes” or “no”) questions. For each question, the data are partitioned into two branches such that the response observations in each branch are more homogeneous. Then a simple regression model is fit to each partition. Such repeated partitioning creates a treelike structure with the branches based on the values of the predictors. Some popular methods for building decision trees are CART (Breiman et al. 1984) and C4.5 (Quinlan 1993).

The hierarchical structure of a tree is inherently designed to capture and represent the interactions between predictors. Decision trees are insensitive to outliers and can easily handle missing data in predictors. In practice, decision trees are grown using greedy-type methods that make locally optimal splits at each step, instead of finding the globally optimal tree. Even though this can potentially lead to suboptimal trees, decision trees are effective in many applications. Decision trees are fast to train and can handle high-dimensional data with many predictors. They are also easy to interpret and visualize, making them a popular choice for many machine learning tasks. Decision trees have been widely used in scientific fields such as biomedicine, genetics, and marketing, among many other fields.

We first focus on introducing decision trees for classification, where the dependent variable is categorical. Then we describe decision trees for regression, where the dependent variable is continuous.

Classification trees

To motivate the concept of a decision tree, we consider a toy dataset where the goal is to predict whether a mushroom is edible or poisonous, coded as e and p, respectively, based on two predictors: cap diameter and season. The cap diameter is a continuous variable and season is categorical, where s and w denote summer and winter, respectively.

	capdiam	season	class
1.	7.3	s	e
2.	7.68	s	e
3.	8.4	s	e
4.	8.86	w	p
5.	9.03	s	e
6.	9.1	s	e
7.	9.59	w	p
8.	9.59	s	e
9.	10.42	w	e
10.	10.5	s	e
11.	12.85	s	e
12.	13.55	w	p
13.	14.07	w	p
14.	14.17	s	p
15.	14.64	s	p
16.	14.85	s	p
17.	14.86	s	p
18.	15.26	w	p
19.	15.34	s	p
20.	16.6	w	p

Based on the training data, a classification tree learns an ordered sequence of questions, where the answer to each question in the sequence affects the type of question asked in the next step. The tree diagram below shows the decision tree for our toy example. The method starts at the top of the tree, called the root node, and uses the entire training dataset. In this example, the root node splits the dataset into two parts based on the cap diameter predictor. By convention, the “yes” answer to the question at the node splits to the left, and the “no” answer splits to the right. A node is a subset of predictors. It can be classified as a terminal or nonterminal. A nonterminal node or parent node splits the data into two regions using the predictor that results in the best fit. (We will describe later how such a predictor is selected.) A terminal node or leaf node does not split the data further.

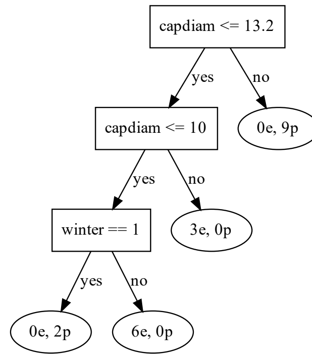


Figure 5.

For example, at the root node, the best split occurs for the predictor x_i = `capdiam` at the split point $t_1 = 13.2$. This split partitions the data into the $\{\mathbf{x}|x_i \leq t_1\}$ and $\{\mathbf{x}|x_i > t_1\}$ regions. Throughout this entry, we will denote the split points by t_s, where s denotes the number of the split, counted from top to bottom and left to right on the above tree. The partition of the predictor space continues recursively until some stopping criterion is applied or there are no more splits. The set of all terminal nodes is called a partition of the data. Each observation from the training data falls into one of the terminal nodes.

Below, we show the partition of the predictor space into the regions that correspond to the above tree diagram. The red and yellow vertical lines correspond to the `capdiam` ≤ 13.2 and `capdiam` ≤ 10 conditions, and the horizontal line depicts the `winter` $= 1$ condition. The green and blue dots correspond to the observations with classes `p` and `e`, respectively.

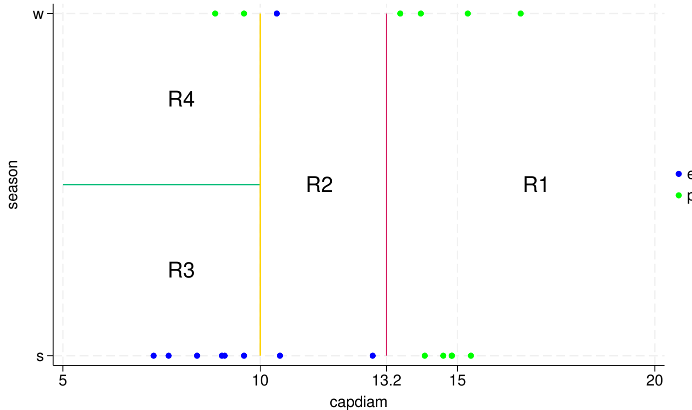


Figure 6.

We can now classify observations by first determining to which terminal node they belong based on their predictor values and then finding the most common class in that terminal node. Thus, for an observation in the terminal node j with the corresponding region R_j, an observation is predicted to be in the class with the largest proportion of observations from the training data, $\max_k p_{jk}$, where p_{jk} is the proportion of training observations in R_j belonging to class k and $k = 1, 2, \ldots, K$. Suppose we have

a new observation for which `capdiam` = 8.32 and `season` = `winter`. If we "put" this observation in the classification tree above, it will end up in the terminal node 4 in the region R_4 with 0 edible and 2 poisonous mushrooms. Therefore, our tree will classify the new observation as a poisonous mushroom.

We now discuss how to choose which predictor to split on and how to determine the best split in each nonterminal node in a decision tree. To choose the predictor and split point, we need to introduce impurity measures that quantify the splitting criteria. One such measure is the misclassification error rate. For a terminal node j with the corresponding region R_j, the misclassification error rate is the fraction of training observations that do not belong to the most common class, that is, $1 - \max_k \widehat{p}_{jk}$, where $\widehat{p}_{jk}$ is an estimate of p_{jk}. Unfortunately, the misclassification error rate is not very sensitive to changes in the class probabilities of each node, meaning that multiple splits may correspond to the same class probabilities, making it difficult to select the best splits. Thus, the misclassification error rate is not recommended for growing a classification tree.

Instead, the following measures are used: The Gini index,

$$\sum_{k=1}^{K} \widehat{p}_{jk}(1 - \widehat{p}_{jk})$$

and cross-entropy,

$$-\sum_{k=1}^{K} \widehat{p}_{jk} \ln \widehat{p}_{jk}$$

The Gini index and cross-entropy are close to zero when all proportions $\widehat{p}_{jk}$'s are close to zero or one. This explains the name "impurity measure"—a small value indicates that the node contains many observations from the same class.

Here we focus on cross-entropy. When the number of groups $K = 2$, cross-entropy is

$$\imath_j = -\widehat{p}_{j1} \ln \widehat{p}_{j1} - (1 - \widehat{p}_{j1}) \ln(1 - \widehat{p}_{j1})$$

The goal of classification trees is to partition the predictor space into regions $R_1, R_2, \ldots, R_J$ that minimize cross-entropy. In practice, the consideration of every possible partition of the predictor space into J rectangles is computationally infeasible. A typical remedy for such problems is to use a greedy approach and successively split the predictor space into two new regions through binary splitting. The binary splitting is performed by first selecting the predictor x_i and the split point t such that it leads to the greatest possible reduction in cross-entropy. In other words, the method examines all predictors x_1 through x_p and considers all possible values of the split point t such that the selected predictor x_i and cutpoint t result in the lowest cross-entropy. Once we have determined the best split point for a given predictor, we can use this information to split the data into two sets and repeat the process for each of the two new sets, continuing until we reach a terminal node or until a stopping criterion is reached.

We start by considering a possible split for the root node. Because the variable season is binary, we can tabulate it to determine the possible split point t.

```
. tabulate class season, column
```

Key
frequency *column percentage*

class	season s	w	Total
e	8 61.54	1 14.29	9 45.00
p	5 38.46	6 85.71	11 55.00
Total	13 100.00	7 100.00	20 100.00

From the above table, season splits the dataset into two nodes: summer, s, and winter, w. The summer node contains 8 edible and 5 poisonous mushrooms, and the winter node contains 1 edible and 6 poisonous mushrooms, respectively. The cross-entropy for the summer and winter nodes can be computed as

$$\imath(\text{summer}) = -\frac{8}{13}\ln\frac{8}{13} - \frac{5}{13}\ln\frac{5}{13} \approx 0.666$$

and

$$\imath(\text{winter}) = -\frac{1}{7}\ln\frac{1}{7} - \frac{6}{7}\ln\frac{6}{7} \approx 0.410$$

The summer and winter nodes contain different numbers of observations. Thus, to find the cross-entropy for the split, we take the weighted average of the entropies in each region:

$$\imath(\text{season}) = -\frac{13}{20}0.666 - \frac{7}{20}0.410 \approx 0.576$$

We can also find the importance or the goodness of fit of the split by measuring the improvement of the impurity measure gained from splitting the parent node into the summer and winter children nodes,

$$\imath(\text{summer}, \text{winter}) = \imath(\text{season}_b) - \imath(\text{season}) \tag{1}$$

where season_b indicates the cross-entropy before the split. Here

$$\imath(\text{season}_b) = -\frac{9}{20}\ln\frac{9}{20} - \frac{11}{20}\ln\frac{11}{20} \approx 0.688$$

Therefore, $\imath(\text{summer}, \text{winter}) = 0.112$. This value indicates the improvement attributed to this split and can be used as a measure of the predictor's importance.

Next we consider splits for the cap diameter predictor. Conventionally, to estimate the cross-entropy for a continuous variable, we first need to sort the data and consider all possible cutpoints (Breiman et al. 1984). For example, for the cap diameter, a possible cutpoint t between the respective 1st and 2nd values of 7.3 and 7.68 is selected as $t = (7.3 + 7.68)/2$, between the 2nd and 3rd values of 7.68 and 8.4, $t = (7.68 + 8.4)/2$, and so on. However, for high-dimensional data such an approach is computationally

expensive. To overcome this, some software packages, such as H2O, divide the data into discrete equal-size sections by using histogram bins and then estimate the best split among those sections (Ben-Haim and Tom-Tov 2010; Chen and Guestrin 2016; and Ke et al. 2017).

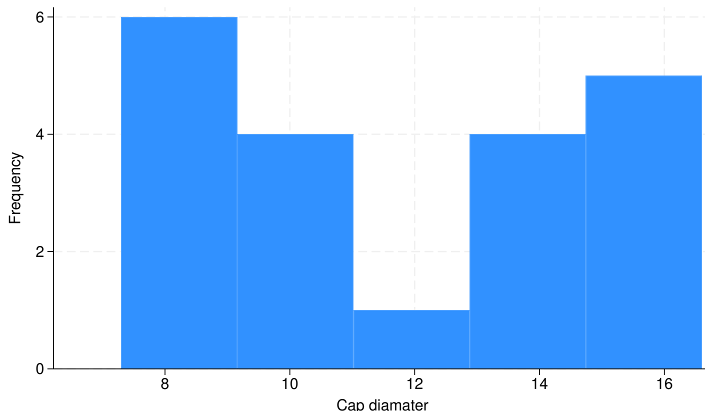


Figure 7.

For illustration purposes, we considered five bins for the histogram of `capdiam`. The number of splits to be evaluated is then determined by the number of bins in the histogram. In practice, the number of bins is a hyperparameter, that is, a parameter that we learn or tune using the training data such that the tuned parameters minimize the generalization error; see *Hyperparameter tuning*. After binning, the number of possible split points reduces to five. For example, because the 1st bin contains 6 observations, a potential split point can be computed by averaging the 6th and 7th observations for `capdiam` in the dataset: $t = (9.1 + 9.59)/2 = 9.345$. Similarly, we can compute all 5 split points, which are $\{9.345, 11.68, 13.2, 14.75, 16.6\}$.

We show the calculation of the cross-entropy only for the split point $t = 13.2$, which is the best split point. You can calculate the cross-entropy for the other split points similarly. The criterion ($\texttt{capdiam} \leq 13.2$) splits the data into two regions, where the left region contains 9 edible and 2 poisonous mushrooms and the right region contains 0 edible and 9 poisonous mushrooms. The right region, which contains observations for which ($\texttt{capdiam} > 13.2$), is called pure because it is homogeneous and is a terminal node. Analogously to the splits for the season predictor, we can compute the cross-entropy for the left and right regions as

$$\imath(\text{left}) = -\frac{9}{11}\ln\frac{9}{11} - \frac{2}{11}\ln\frac{2}{11} \approx 0.474$$

and

$$\imath(\text{right}) = 0$$

Therefore, the cross-entropy for the split is equal to

$$\imath(\texttt{capdiam} \leq 13.2) = \frac{11}{20}0.474 + \frac{9}{20}0 \approx 0.261$$

The cross-entropy before the split can be computed by using the actual class distribution of `class`:

$$\imath(\texttt{capdiam}_b) = -\frac{11}{20}\ln\frac{11}{20} - \frac{9}{20}\ln\frac{9}{20} \approx 0.688$$

From the above, the importance of the capdiam split is

$$\imath(\texttt{capdiam} \leq 13.2, \texttt{capdiam} > 13.2) \approx 0.688 - 0.261 = 0.427$$

Thus, in the root node we select the cap diameter with the best split $t = 13.2$, because the gain from the cap diameter split (0.427) is larger than the gain from the season split (0.112). The next best split is found following the same steps but by considering only the subset of the dataset that satisfies the criterion ($\texttt{capdiam} \leq 13.2$). The tree grows recursively until all observations are classified.

In the last recursive split ($\texttt{winter} = 1$), the left region contains only two observations. Splits with few observations may lead to overfitting. To avoid overfitting, we recommend to limit the minimum number of observations that a leaf node may have for the node to be considered for splitting. For example, if we limit the minimum number of observations in the leaf nodes to three, then the last split ($\texttt{winter} = 1$) will not occur because this criterion requires that both branches have at least three observations.

In general, each split increases the depth of the decision tree, and large trees usually overfit the data. On the other hand, small trees may not capture a complex structure hidden in the data. Thus, the tree size is treated as a hyperparameter, and its optimal value is chosen from the data.

For the multiclass classification with K classes, the preferred approach is to compare each class k with the rest (Rifkin and Klautau 2004). That is, we grow K different trees and for each k find the probability of class k, p_k. Then the final class prediction is computed as $\max_k p_k$.

Regression trees

The general idea for growing a regression tree is similar to a classification tree. The main goal is to partition the predictor space into distinct and nonoverlapping regions by using binary splits. However, because in regression trees the response is continuous, we use the residual sum of squares RSS $= \sum_{i=1}^{N}(y_i - \widehat{y}_i)^2$ as an impurity measure instead of the cross-entropy to determine the best split at each node. Then, for each terminal node, the prediction is computed as the mean of the response values $\mathbf{y}$ in the region corresponding to the terminal node. For example, if the mean response of the training observations in the first region R_1 is $\widehat{c}_1 = 5$, then for a given observation $\mathbf{x}_i \in R_1$, the regression tree will predict a value of $\widehat{c}_1 = 5$. Thus, the regression model prediction for J distinct and nonoverlapping regions, which correspond to J terminal nodes, can be represented as

$$\widehat{f}(\mathbf{x}) = \sum_{j=1}^{J} \widehat{c}_j I\{\mathbf{x} \in R_j\}$$

where $\widehat{c}_j = \text{Mean}(y_i | \mathbf{x}_i \in R_j)$.

In general, growing a regression tree can be summarized by the following two steps (James et al. 2021):

1. Partition the predictor space into J distinct and nonoverlapping regions $R_1, R_2, \ldots, R_J$.
2. For each observation that belongs to the region R_j, predict the response as the mean of the response values for the training observations in R_j.

Therefore, the goal of a regression tree is to partition the predictor space into rectangles $R_1, R_2, \ldots, R_J$ that minimize the RSS:

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \widehat{c}_j)^2$$

Similar to a classification tree, the binary splitting is performed by first selecting the predictor x_i and the cutpoint t such that it leads to the greatest possible reduction in RSS. Mathematically, in each nonterminal node, a regression tree tries to select the predictor x_i and cutpoint t such that the following expression is minimized,

$$\min_{i,t}\left\{\sum_{\mathbf{x}_i\in R_1(i,t)}(y_i-\hat{c}_1)^2+\sum_{\mathbf{x}_i\in R_2(i,t)}(y_i-\hat{c}_2)^2\right\}$$

where $R_1(i,t)=\{\mathbf{x}|x_i\leq t\}$ and $R_2(i,t)=\{\mathbf{x}|x_i>t\}$. Then the above process is repeated recursively to minimize the RSS within each region. As for a classification tree, the importance of the split $\imath(\cdot)$ is defined as the difference between the RSS before and after the split.

It is recommended to apply a stopping criterion to avoid overfitting. For example, the node splitting may be terminated if the method reaches some predetermined tree depth or the terminal regions contain no more than a prespecified number of observations.

After the terminal nodes and the corresponding regions are determined, we obtain predictions for the test observations by first identifying to which terminal nodes the test observations belong. Then the predicted response is computed as the mean of the training observations in the corresponding terminal node. This is in contrast with classification trees, where the predicted response is determined by the most common class among the training observations in the terminal node.

One issue with decision trees is that the partitioning of a categorical predictor can take different but equally justifiable paths. For example, we can decompose categories into binary predictors and include them individually in the model (also known as one-hot encoding) or implement more dynamic splits, such as groups of two or more categories. The best approach depends on the specific data and model. In general, the partitioning algorithm tends to favor categorical predictors with many levels, leading to severe overfitting when the number of categories is large; see, for instance, *Effect of categorical predictors* in [H2OML] **h2oml**. Therefore, it is recommended to avoid such predictors.

Pros and cons of decision trees

One of the key advantages of decision trees is that they represent information in an intuitive and easy-to-visualize way. In a decision tree, predictors can be of any type: numeric, binary, categorical, etc. A monotone transformation or different scales of measurements among predictors do not change the model outcome.

Another advantage of decision trees is that they can handle missing data. For instance, missing values are often treated as containing information, which does not require the common missing-at-random assumption. For categorical predictors, missing values are treated as a separate category that can split left or right; for other types of predictors, the missing values split to the left. Then, for the testing or validation data, the missing values follow the path on the tree that was determined during training. If there are no missing values in the training data, then missing values in the testing or validation data follow the path of the most training observations. Missing values in the response are also allowed, but nothing will be learned from observations containing those missing values.

Despite their advantages, decision trees are notoriously unstable and have a high variance. Even though a deep tree (with many terminal nodes) has a small bias, a small change in the data can lead to a completely different set of splits and obscure its interpretation. Moreover, decision trees have difficulties with modeling simple smooth functions; see, for instance, *Introduction* in [H2OML] ***h2oml gbm***.

One solution is to use ensemble methods, which we introduce next.

Ensemble methods

The basis for ensemble methods can be summarized as a mechanism that forms a smart committee of incompetent but carefully selected members to solve a machine learning problem. As we discussed in the previous section, despite their advantages such as efficiency and interpretability, decision trees suffer from high variance and instability. Specifically, if we slightly modify the data by splitting them or introducing nuisance predictors, the new results may differ substantially from the original results. In contrast, the low-variance methods are more robust to small changes and tend to yield similar results.

Bagging and boosting are two methods used to improve the accuracy of a machine learning method by combining unstable learners. Using unstable learners is important because they provide more variable outcomes than stable learners and thus aid in generalization. Both methods perturb the original dataset to generate an ensemble of various base learners and combine them into one method. The usefulness of ensemble methods is established for unstable base learners, but these methods may produce contradictory results for stable base learners such as a linear regression.

Both bagging and boosting methods are general-purpose procedures and are not tied to a specific learning estimation method, but in this entry, our main focus is on bagging and boosting for decision trees. The main difference between bagging and boosting is in how they perturb and generate new datasets. Bagging, which was first introduced in Breiman (1996), generates the perturbations by random and independent drawings (bootstrap samples) from the training data. In contrast, boosting, introduced by Freund and Schapire (1997) to solve classification problems, has a deterministic approach and generates perturbations by sequentially reweighting the dataset. In particular, at any step, the weights of the observations that were misclassified in the previous step increase, whereas the weights for the correctly classified observations decrease. Thus, boosting forces each successive classifier to focus on those observations that were missed by the previous ones in the sequence. By design, bagging reduces variance, whereas boosting tends to control the generalization error by reducing bias. The difference is summarized in the figure below.

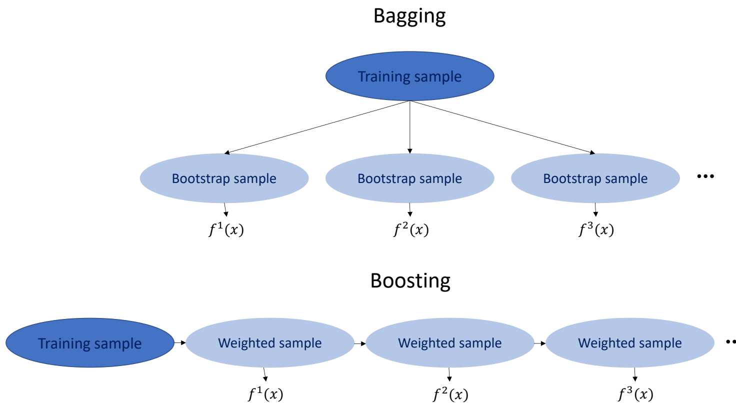


Figure 8.

Bagging

Bagging or bootstrap aggregation relies on a bootstrap procedure (Efron 1979) that combines an ensemble of learners to improve the performance of the prediction. The main idea of bagging can be motivated by the fact that the variance of the mean of n independent observations $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ with

variance σ^2 is σ^2/n. Consequently, averaging a set of independent observations reduces the variance. A natural extension of this idea to the machine learning is to independently sample many training datasets from the population, build a separate prediction model $\widehat{f}^b(\mathbf{x})$ for each sample, and take the average. Unfortunately, this approach is not viable because, in practice, we observe only one training dataset. However, we can use bootstrap to generate samples from the training dataset. Thus, after building the $\{\widehat{f}^b(\mathbf{x}), b = 1, 2, \ldots, B\}$ learners from the bootstrap samples, for the observation $\mathbf{x}$, the bagging procedure returns

$$\widehat{f}_{\text{bag}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \widehat{f}^b(\mathbf{x})$$

The bias of a bagged tree is the same as that of a single tree, because each tree generated from the bootstrapped data is identically distributed and has the same expected value.

To apply bagging to regression trees, we grow B deep regression trees using B bootstrap samples and take the average of the resulting predictions. Each deep regression tree has a high variance and low bias. Therefore, averaging these B trees substantially reduces the variance and improves the prediction accuracy; see *Fundamentals of machine learning* for details about the bias–variance tradeoff.

There are several approaches for extending bagging to classification trees. The most common one is the majority-vote rule. For the ith observation of the testing data, we can record the predicted class for each of the B classification trees. The majority-vote rule returns the most frequent class among these B predictions.

A salient feature of bagging is its ability to estimate the test error of a bagged model. This feature helps avoid arduous computations and is especially useful for large datasets. Bagging repeatedly builds trees on bootstrap samples, and about 37% of the observations in the training data will not be selected for each bootstrap sample (Izenman 2008, chap. 5). Therefore, each bagged tree is grown only on the remaining two-thirds of observations. The 37% of observations that are not used to grow the tree serve as an independent testing set. Such observations are called out-of-bag observations. Now, to predict the response for the ith observation, we use each of the trees for which the ith observation was out of bag. The average (or the majority vote in the case of classification) of those predicted responses yields a single prediction for the ith observation. The estimated generalization error from the out-of-bag approach is a valid estimate of the test error and is equivalent to using an independent testing set of the same size.

Random forest

Recall that bagging averages an ensemble of unstable decision trees to reduce the variance, which leads to the improvement of the generalization error. However, this reduction may not be sufficient 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 the top split. Therefore, most of the bagged trees will have a similar structure, resulting in predictors that are highly correlated.

Although historically a variety of tree ensembles have been referred to as a random forest (Lin and Jeon 2006), nowadays, a random forest is associated with the random forest proposed in Breiman (2001), which is a tree ensemble that uses both bagging and subsampling of predictors. It is a modification of the bagging procedure that generates an ensemble of decorrelated trees and then averages them. To overcome the shortcomings of the bagging procedure and achieve decorrelation, for each split in the tree, instead of the full set of p predictors, random forest selects a random sample of m predictors as potential split candidates. With this strategy, the strong predictors, on average, $(p-m)/p$ times are not considered

as potentially the best predictors to split on, which increases the chance that other predictors can be considered for splitting. Below, we summarize the main steps of a random forest. For $b = 1, 2, \ldots, B$, do the following:

1. Generate a bootstrap sample D^b from the training data.

2. Until the stopping criterion is reached, recursively grow a tree T_b by implementing the following steps:

   i. Randomly choose $m \leq p$ predictors.
   ii. Select the predictor with the best split point from m potential predictors.
   iii. Split the selected node.

Similar to bagging, to make a prediction for a new test point $\mathbf{x}$, random forest estimates $\widehat{f}_{\text{rf}}(x) = (1/B)\sum_{b=1}^{B} \widehat{f}^b(x)$ for regression, where $\widehat{f}^b(\cdot)$ is a prediction model from the tree T_b, and uses the majority-vote rule for classification. In practice, it is recommended to select $m = \lfloor \sqrt{p} \rfloor$ for classification and $m = \lfloor p/3 \rfloor$ for regression, where $\lfloor \cdot \rfloor$ is a floor function. The size of the bootstrap sample D^b controls the bias–variance tradeoff of the random forest.

A smaller bootstrap sample size lowers the probability of a particular training observation to be included in the bootstrap sample, which decreases similarity among the individual trees. The latter helps reduce overfitting. Analogously, a larger bootstrap sample size increases the degree of overfitting.

The above approach describes a random forest as a complex black-box model. We find it helpful to also describe a random forest from a different perspective that connects it to the existing well-understood statistical methods. Specifically, the prediction from a random forest can be viewed as an adaptive neighborhood classification or regression procedure (Lin and Jeon 2006). Recall from decision trees that every terminal node $j = 1, 2, \ldots, J$ of a tree corresponds to a rectangular subspace R_j of a predictor space such that for every observation $\mathbf{x}_i$, there is only one terminal node j such that $\mathbf{x}_i \in R_j$. Let's focus on a prediction from a single tree T_b at a new data point $\mathbf{x}_0$. Suppose that in the tree T_b, $\mathbf{x}_0$ belongs to the terminal node j with the corresponding region $R_j(\mathbf{x}_0, b)$, where we make the dependence of the region on $\mathbf{x}_0$ and tree T_b explicit. Then the prediction is obtained by averaging the observed values y_i's in the region $R_j(\mathbf{x}_0, b)$. Let's assign the weight $w_i(\mathbf{x}_0, b)$ a positive constant if the observation $\mathbf{x}_i$ is in the region $R_j(\mathbf{x}_0, b)$ and 0 otherwise, such that

$$w_i(\mathbf{x}_0, b) = \frac{1\{\mathbf{x}_i \in R_j(\mathbf{x}_0, b)\}}{|\{k\colon \mathbf{x}_k \in R_j(\mathbf{x}_0, b)\}|}$$

where $|\cdot|$ denotes the number of observations in the region $R_j(\mathbf{x}_0, b)$ and $1(A)$ is the identity function, which is equal to 1 if the condition A holds and 0 otherwise. Note that the weights sum to one: $\sum_{i=1}^{n} w_i(\mathbf{x}_0, b) = 1$. Thus, the prediction from a single tree given a new point $\mathbf{x}_0$ is the weighted average of the original observations y_i's for $i = 1, 2, \ldots, n$:

$$\widehat{f}^b(\mathbf{x}_0) = \sum_{i=1}^{n} w_i(\mathbf{x}_0, b) y_i$$

For a random forest, where B trees are ensembled, the prediction at observation $\mathbf{x}_0$ can be written as

$$\hat{f}_{\text{rf}}(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B}\sum_{i=1}^{n} w_i(\mathbf{x}_0, b)y_i = \sum_{i=1}^{n}\overline{W}_i(\mathbf{x}_0)y_i$$

where $\overline{W}_i(\mathbf{x}_0)$ is the average of the weights w_i's over B trees:

$$\overline{W}_i(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B} w_i(\mathbf{x}_0, b)$$

Consequently, a random forest prediction can be viewed as a weighted average of the observations y_i's because $\sum_{i=1}^{n}\overline{W}_i(\mathbf{x}_0) = 1$, which makes a random forest an adaptive smoother (Curth, Jeffares, and van der Schaar 2024). For most observations, the weight $\overline{W}_i$ will be zero; see Lin and Jeon (2006), Meinshausen (2006), and Biau and Scornet (2016).

Wager and Athey (2018) rely on the above approach to prove the consistency of the random forest estimator. In figure 9, we use a toy example to visualize this approach. Here, for a new data point $\mathbf{x}_0$ (denoted by $+$), each tree assigns a positive weight to the observations in the same terminal node (denoted in red) and zero weight to the rest of the observations. The random forest prediction averages the weights from the three trees and measures how frequent each observation falls into the same terminal node as $\mathbf{x}_0$.

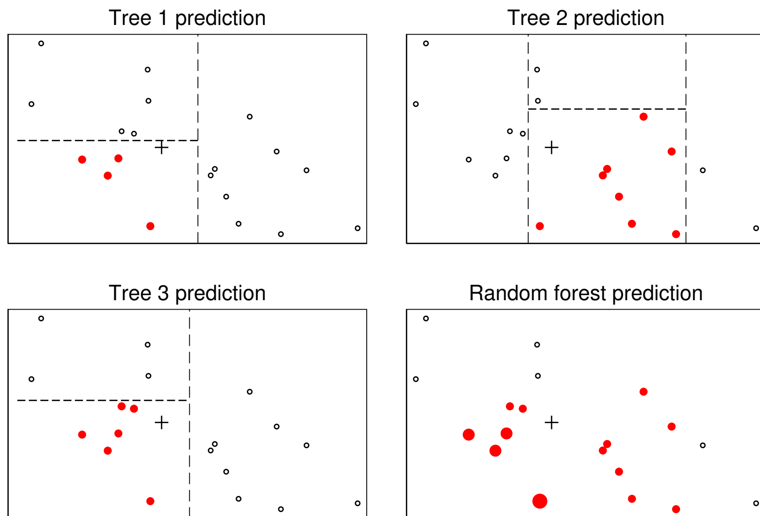


Figure 9.

Boosting

Boosting is a powerful idea that can be applied to any regression or classification problem. In contrast to bagging, where each tree in an ensemble is built on a bootstrap training dataset and independent of the other trees, boosting grows trees sequentially. One of the first boosting methods, AdaBoost (Freund and Schapire 1997), was introduced to solve classification problems. AdaBoost repeatedly applies weights to the observations to produce a sequence of classifiers. The observations that are poorly modeled get higher weights and vice versa. This way, each successive classifier is focused on those observations that received higher weights in the previous iteration. The figure below summarizes the steps of AdaBoost.

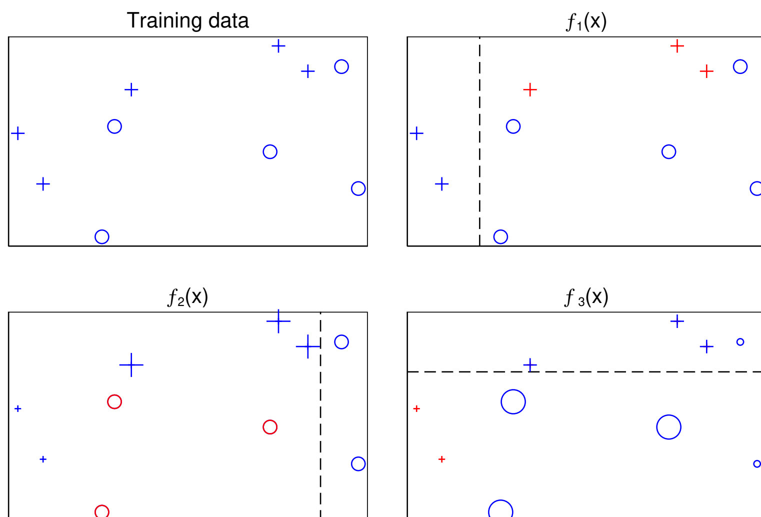


Figure 10.

Here we have three classifiers or base learners, $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, and $f_3(\mathbf{x})$, which can be classification trees. The observations are classified based on $+$'s and o's. AdaBoost starts by assigning the same weight $1/n$ to all observations, where n is the number of observations. $f_1(\mathbf{x})$ incorrectly classified three $+$ observations, which are displayed in red. In the next iteration, those three observations were assigned higher weights, and $f_2(\mathbf{x})$ classified those observations correctly. Similarly, $f_3(\mathbf{x})$ assigned more weight to the three previously misclassified o observations and classified them correctly. The final ensemble or boosted classifier is obtained based on those three classifiers as $F(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x})$, where α_m measures the importance of the classifier $f_m(\cdot)$ and M is the number of classifiers.

This approach tends to explain boosting in terms of updating weights, which makes it difficult to evaluate its performance (Schapire 2003). To establish a connection with the statistical framework, in their seminal paper, Friedman, Hastie, and Tibshirani (2000) propose a different view of AdaBoost. In particular, the authors use a gradient-descent-based formulation to reformulate AdaBoost as an optimization problem and show that it is a greedy procedure that minimizes the exponential loss,

$$L\{y_i, F(\mathbf{x}_i)\} = \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F(\mathbf{x}_i)}$$

where $F(\mathbf{x}_i) = \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x}_i)$. They proposed the following coordinate descent algorithm to achieve the minimization.

1. Initialize: $F_0(\mathbf{x}) = 0$.
2. For $m = 1, 2, \ldots, M$:
   i. Choose a classifier $f_m(\cdot)$ and α_m to minimize
   $$\frac{1}{n}\sum_{i=1}^{n} \exp[-y_i\{F_{m-1}(\mathbf{x}_i) + \alpha_m f_m(\mathbf{x}_i)\}]$$
   ii. Update: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \alpha_m f_m(\mathbf{x})$.
3. Output: $F_M(\mathbf{x})$.

Thus, AdaBoost minimizes its loss function by iteratively descending toward one coordinate direction at each iteration.

The important feature of this loss-function formulation is that, instead of the exponential loss, one can use any other loss function and extend AdaBoost from solving a classification problem to solving a regression problem. For details, see Friedman, Hastie, and Tibshirani (2000), Schapire and Freund (2012), and Hastie, Tibshirani, and Friedman (2009).

GBM

The formulation discussed in the previous section and the corresponding models are called GBMs. GBM is one of the popular methods to implement boosting. Although the original method, proposed in Friedman, Hastie, and Tibshirani (2000), can work with any base learner, in practice, decision trees are some of the main choices.

In the previous section, we viewed AdaBoost as an optimization problem with some loss function $L(F)$. In *Decision trees*, we parameterized a decision tree as a model $f(\mathbf{x}) = \sum_{j=1}^{J} c_j I\{\mathbf{x} \in R_j\}$, where J is the number of terminal nodes, R_j's are nonoverlapping regions of the predictor space, and c_j is the prediction (the mean for regression and the most probable class for classification) in the terminal node j.

The main idea behind GBM is to parameterize the estimate of the ensemble function $F(\mathbf{x})$ as

$$\widehat{F}(\mathbf{x}) = \sum_{i=0}^{M} \widehat{F}_m(\mathbf{x})$$

where M is the number of iterations, $\widehat{F}_0(\cdot)$ is an initial guess, and $\{\widehat{F}_m(\cdot)\}_{m=1}^{M}$ are the function increments, also known as boosts.

Parameterizing the tree by $\Theta = \{R_j, c_j\}_{j=1}^{J}$ and following the coordinatewise approach presented in the previous section, for some loss function $L(\cdot)$, in the stage m, we can write the minimization of the tree-boosting method as

$$(\alpha_m, \Theta_m) = \text{argmin}_{\alpha,\Theta} \sum_{i=1}^{n} L\{y_i, \widehat{F}_{m-1}(\mathbf{x}_i) + \alpha f(\mathbf{x}_i, \Theta)\}$$

where n is the number of observations in the training dataset, α is a learning rate, and

$$\widehat{F}_m(\mathbf{x}) = \widehat{F}_{m-1}(\mathbf{x}) + \alpha f(\mathbf{x}, \Theta_m)$$

Unfortunately, such minimization is practically infeasible to solve. To alleviate the issue, it was proposed, at stage m, to choose a new function $f(\mathbf{x}, \theta)$ to be the most correlated with the negative gradient

$$g_m(\mathbf{x}_i) = \left[\frac{\partial L\{y_i, F(\mathbf{x}_i)\}}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x}_i)=\widehat{F}_{m-1}(\mathbf{x}_i)}$$

by solving a classical least-squares minimization problem:

$$(\alpha_m, \Theta_m) = \text{argmin}_{\alpha,\Theta} \sum_{i=1}^{n} \{-g_m(\mathbf{x}_i) + \alpha f(\mathbf{x}_i, \Theta)\}$$

For example, if the loss function is the squared error loss $L\{y_i, F(\mathbf{x}_i)\} = (1/2)\{y_i - F(\mathbf{x}_i)\}^2$, then the gradient $g_m(\mathbf{x}_i) = -\{y_i - F(\mathbf{x}_i)\}$.

Below, we summarize the gradient-tree boosting method for the squared error loss $L(\cdot)$ and fixed learning rate α, with the number of iterations, that is, the number of trees in this context, equal to M.

1. Initialize: $F_0(\mathbf{x})$ and $g_i = y_i$ for all $1 \le i \le n$.
2. For $m = 1, 2, \dots, M$:
   i. Compute $g_m(\mathbf{x}_i) = -\{y_i - F_{m-1}(\mathbf{x}_i)\}$ for all $1 \le i \le n$.
   ii. Fit a tree $\widehat{f}_m(\cdot)$ with J splits to the training data $\{\mathbf{x}_i, -g_m(\mathbf{x}_i)\}$ for $i = 1, 2, \dots, n$.
   iii. Update $\widehat{F}$: $\widehat{F}_m(\mathbf{x}) = \widehat{F}_{t-1}(\mathbf{x}) + \alpha \widehat{f}_m(\mathbf{x})$.
3. Output: $\widehat{F}(\mathbf{x}) = \sum_{m=1}^{M} \widehat{F}_m(\mathbf{x}) = \sum_{m=1}^{M} \alpha f_m(\mathbf{x})$.

The learning rate α reduces the contribution of each tree as it is added to the model, which prevents overfitting. The simulation studies indicate that a smaller α reduces overfitting and provides a lower generalization error. The relationship between the learning rate and the number of trees M is reciprocal. That is, decreasing the learning rate increases the required number of trees.

Historically, researchers suggested using a stump (decision tree with depth equal to one) as a base learner in each iteration. However, current research on ensemble methods suggests that if the noise in the data is small, it is preferable to use deeper trees as base learners to improve generalization (Wyner et al. 2017). This is related to the idea that the ensemble methods are local interpolators. The depth of a tree affects the selection of the optimal number of trees. For a given learning rate, fitting more complex (deeper) trees results in a fewer number of trees being selected. Typically, the learning rate and tree complexity are inversely related: doubling the tree depth should be matched with halving the learning rate to provide roughly the same number of trees (Elith, Leathwick, and Hastie 2008).

Trees with monotonicity constraints

In some applications, it is reasonable to assume that the response is a monotone function of the predictors. For example, in economic theory the price elasticity of the normal good is assumed to be positive, or in hedonic price analysis, in which price is a function of the characteristics of the product, it is expected that some of the characteristics will always have a positive or negative effect on the price. The original decision trees and ensemble decision tree methods, described above, do not support such a constraint and may violate the monotonicity assumption. However, there are modifications to the above methods that incorporate the monotonicity constraints (Potharst and Feelders 2002).

Model selection in machine learning

Most machine learning models are defined by a set of model parameters and hyperparameters. A model parameter is initialized and computed during the learning process. A hyperparameter cannot be directly estimated from the learning process and must be prespecified before training a machine learning model (Kuhn and Johnson 2013). For example, in decision trees, the parameters correspond to the split decisions and regions, and the hyperparameters include the tree depth, impurity measures, the minimum number of observations in each terminal node, and more. The goal of machine learning models is to make accurate predictions on future data. To build an optimal model, we need to explore a wide range of values for hyperparameters and select the ones that improve the model performance the most. This process is also known as model selection. So we are interested in selecting the best-performing model from the set of potential models. That is, we want to evaluate the performances of the models and compare them with each other. The process of designing an effective machine learning model with an optimal hyperparameter configuration is called hyperparameter tuning. The material in this section closely follows Raschka (2020) and Yang and Shami (2020).

The steps for selecting the best-performing model are summarized in table 1 below.

Table 1. Steps for selecting the best-performing model

To minimize the generalization error, which measures the predictive model performance on new data, do the following:

1. Split the data for training and evaluating a model; see *Three-way and two-way holdout methods*.
2. Optimize hyperparameters to select the best-performing model; see *Hyperparameter tuning*.
3. Compare different machine learning methods and select the one that performs the best; see *Method comparison*.

In the rest of this section, we will discuss different approaches to accomplish the above steps.

Three-way and two-way holdout methods

The simplest approach to evaluating a model is the two-way holdout method, in which we take the observed data and split them into two parts: training data and testing data. A model is fit to the training data, and the prediction is obtained on the testing data. It is important to perform the training and evaluation steps using different data. Otherwise, if a sufficiently complex model fits the training data too well, it will be difficult to distinguish whether the model is memorizing the training data or generalizing well to the "new" data. Thus, the model performance will suffer from the optimism bias. Even after we randomly sample and split the data, it is essential to prevent the leakage of information from the testing data into the training process (Raschka 2020 and Lones 2021). Common, seemingly innocuous mistakes include using the information about the means and ranges of the predictors from the entire dataset to scale the predictors or performing predictor selection before partitioning the data and using the same data as testing data to evaluate the generality of multiple models. The best practical way to prevent information leakage is to partition the data at the beginning of the analysis and use the testing data only once to measure the generality of a final model at the end of the analysis (Cawley and Talbot 2010).

The two-way holdout method addresses only the first generalization step from table 1 and cannot be used to sequentially train multiple models for hyperparameter optimization, which we discuss later. In contrast, the three-way holdout method partitions the dataset into training, validation, and testing data. Model selection and hyperparameter tuning are performed on training and validation data and model evaluation on testing data. This procedure avoids repeated use of the testing data and prevents information leakage. Another advantage of including validation data is that we can impose early stopping rules, in which the model performance is measured against validation data at each iteration, and stop training when the performance score starts deteriorating or does not change over a sequence of iterations. In general, to obtain a generalization error, which is independent from how we split the data into training, validation, and testing, we recommend to repeat the holdout method multiple times with different random-number seeds and report the average performance over these repetitions. Alternatively, one can use the leave-one-out bootstrap technique and evaluate the generalization error by using the out-of-bag samples instead of the training data (Efron and Tibshirani 1993).

The steps for selecting the best-performing model with the three-way holdout method are summarized in table 2.

Table 2. Steps for selecting the best-performing model with the three-way holdout method

1. Randomly partition the data into three parts: training for model fitting, validation for model selection, and testing for the final evaluation of the selected model.
2. Hyperparameter tuning: define a grid of various hyperparameter configurations to fit models to the training data; see *Hyperparameter tuning*.
3. Model selection: evaluate and compare the estimated performance metrics on the validation data, and choose hyperparameter values that provide the best-performing metrics.
4. Use independent testing data to estimate the generalization error by comparing various metrics of the best-performing model.

In step 2, tuning can be performed by using either a Cartesian grid search (as described in table 4) or a random grid search. We treat the splitting of a dataset into training, validation, and testing data as random subsampling and assume that each observation has been drawn from the same probability distribution. However, when the dataset is imbalanced, random subsampling is not recommended. A better approach is to divide the dataset in a way that preserves the original class proportions in the resulting subsets (training, validation, and testing). This approach is called stratification.

k-fold cross-validation

For small datasets, the three-way holdout method of splitting the data is not recommended because the validation and testing data may not be representative. In such cases, k-fold cross-validation is the most common model evaluation and selection technique. It starts by splitting the data into training and testing data. For the training data, k-fold cross-validation splits them into k parts or folds. In each kth iteration, it uses one part for validation and the remaining $k-1$ parts as a training subset for model fitting. The figure below illustrates 3-fold cross-validation for a toy example. The dataset is randomly split into three folds, and red, blue, and green observations correspond to observations in folds 1, 2, and 3, respectively. In the first cross-validation iteration, the method uses observations in folds 2 and 3 as a training set and

observations in fold 1 as a validation set. The next two iterations follow a similar procedure but use observations from folds 2 and 3, respectively, as validation sets. For example, for $k = 3$, four models are fit. The first three cross-validation models are fit using 2/3 of the training data, as described above, and a different 1/3 of the training data is held out for validation for each of the three models. Then the main fourth model is fit using the entire training data, and the cross-validation metrics are reported. Also see [H2OML] **h2omlestat cvsummary**.

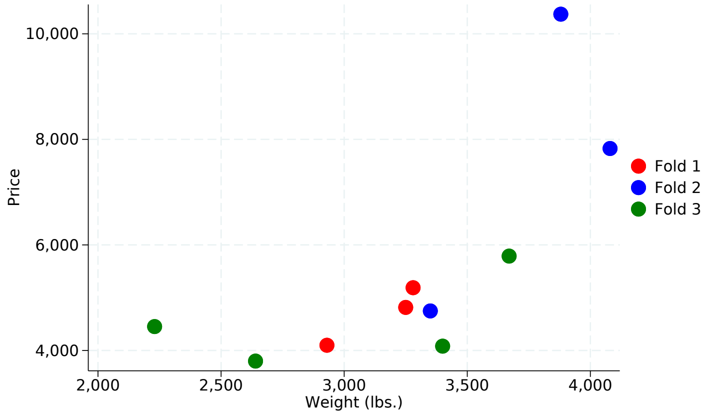


Figure 11.

Hyperparameter tuning

A typical process to build an effective machine learning model is complicated and time consuming. It involves choosing an appropriate method and selecting a model by tuning hyperparameters (see step 2 in table 1). The choice of optimal hyperparameters directly affects the model performance on the testing data. The hyperparameter tuning depends on a machine learning method and the type of hyperparameter, such as continuous, discrete, or categorical. Setting and testing hyperparameters manually is time consuming and inefficient. Therefore, there exist automatic optimization techniques for hyperparameter tuning.

The main goal of hyperparameter optimization is to achieve optimal model performance within a given budget, where budget refers to computational resources or the time allocated to tuning. We summarize the hyperparameter optimization process following Yang and Shami (2020) in table 3.

Table 3. Steps for hyperparameter optimization

1. Select the machine learning method and the performance metrics.
2. Select the hyperparameters that require tuning.
3. Determine the baseline or reference model by training the machine learning method using the default hyperparameter configuration.
4. Start with a large search space such as the hyperparameter feasible domain.
5. Refine the search space using well-performing hyperparameter values, or explore new areas if needed.
6. Select the best-performing hyperparameter configuration as the final result.

Some researchers often neglect the baseline determination step 3 and spend most of their time developing complex models, which may not outperform the simplest model. For example, if the task is binary classification or regression, then the baseline method can be the simplest known method such as logistic or linear regression. Or if our data are highly imbalanced with one of the classes containing 95% of observations, then this 95% can serve as our baseline, because the method that always predicts this class already has 95% accuracy and the preferred machine learning model should outperform this baseline.

The simplest hyperparameter tuning method is a so-called babysitting or trial and error approach, where a researcher manually experiments with various hyperparameter values using experience, intuition, or prior knowledge (Abreu 2019 and Elsken, Metzen, and Hutter 2019). Manual tuning is infeasible for most machine learning methods because they are complex and require many hyperparameters. The methods we describe next are more suitable for complex machine learning methods.

Decision-theoretical methods are one of the common techniques for hyperparameter optimization. The most popular ones are a Cartesian grid search (Bergstra et al. 2011) and a random grid search (Bergstra and Bengio 2012). A Cartesian grid search performs an exhaustive grid search of hyperparameter configurations and evaluates the Cartesian product of possible hyperparameter combinations. Its search is limited to the grid specified by the user and cannot explore other regions. To achieve good results, Yang and Shami (2020) suggest the steps that we summarize in table 4.

Table 4. Steps for Cartesian grid search

1. Choose a broad search space and a large step size.
2. Based on the results from step 1, refine the search space and step size using well-performing hyperparameter configurations.
3. Repeat step 2 until there is no substantive improvement in the performance metric.

A Cartesian grid search is exhaustive, which makes it infeasible for a high-dimensional hyperparameter configuration space. A random grid search overcomes this drawback by randomly choosing a set number of samples within the upper and lower bounds as candidate hyperparameter values. Those values are used to evaluate the model. The rest of the steps are the same as in table 4. Moreover, if the configuration space is large enough, then the global optimum of the tuning metric can be achieved. On a limited budget, a random grid search explores a larger search space than a Cartesian grid search. However, both Cartesian and random grid search methods share the same drawback: each hyperparameter evaluation is independent of the others, leading to wasted computational time and resources on poorly performing areas of the search space. For a review of hyperparameter optimization techniques, see Yang and Shami (2020).

Method comparison

Comparing evaluation results for different machine learning methods is fundamental to model selection (step 3 in table 1). This process typically includes a comparison of different performance metrics, visualization, and statistical analysis. The performance metrics of various machine learning methods are compared using testing data, and the best method is chosen based on the results. Visualization, such as receiver operating characteristics curves and precision–recall curves, are commonly used for comparison during binary classification. For details, see [H2OML] **h2omlgraph roc** and [H2OML] **h2omlgraph prcurve** and, more generally, [H2OML] **h2oml postestimation**. Depending on the research question, in addition to performance metrics, it may be important to also explore the explainability of the method. See the next section for details.

Interpretation and explanation

Machine learning models are ubiquitous in many fields. Despite their widespread use, they are often treated as black boxes that do not explain their predictions in a way that practitioners can understand. The misuse of black-box predictive models can lead to serious consequences, for instance, incorrectly denying parole, releasing dangerous criminals because of inadequate bail decisions, mispredicting air pollution level, and more (Rudin 2019). One of the concerns with deploying machine learning methods is whether their models and predictions can be trusted. And it is difficult to trust something that cannot be interpreted or explained. Traditionally, machine learning models are evaluated by comparing performance metrics using validation data. This may be unreliable because validation data may not always be fully representative of real-world data.

The use of interpretable models and explainable methods sheds light on model performance and encourages a transparent usage of black-box models. In machine learning, an interpretable model has the ability to explain its results in an understandable and transparent way without the need for additional methods (Doshi-Velez and Kim 2017). Commonly used interpretable models are linear and logistic regressions, decision trees, decision-set and rule-based methods and their extensions (Friedman and Popescu 2008; Letham et al. 2015 ; Lakkaraju, Bach, and Leskovec 2016; Rudin and Ustun 2018; and Chen et al. 2018). An interpretable model is judged based on several criteria, including interpretability and accuracy (Guidotti et al. 2018).

In contrast with interpretable models, explainable methods rely on external models and methods to make their predictions presentable and understandable to a human. In general, they do not create models that are inherently interpretable, but provide post hoc models that explain the prediction of the original black-box models (Goldstein et al. 2015 ; Ribeiro, Singh, and Guestrin 2016; Bastani, Kim, and Bastani 2017; and Lundberg and Lee 2017). It is not recommended to heavily rely on explainable models for high-stake decisions, such as in medicine, criminal justice, social bias, and other fields (Rudin 2019

and Ghassemi, Oakden-Rayner, and Beam 2021), but to use those techniques as a tool for analysis and algorithmic audit (Raji et al. 2020). For more information, see Slack et al. (2020), Lakkaraju and Bastani (2020), and Krishna et al. (2022).

In machine learning literature, explainable methods are divided into model specific and model agnostic. A model-specific explainable method is inherently connected to the used machine learning model such as a random forest or a deep neural network and cannot be used for other models. With a model-agnostic explainable method, a user is free to use any black-box model for data analysis, and the explainable method can be applied to that model. There are two types of model-agnostic methods: local and global. Local methods explain individual predictions and approximate a black-box model in the vicinity of an individual observation. The popular methods include local surrogate models (Ribeiro, Singh, and Guestrin 2016), individual conditional expectation curves (Goldstein et al. 2015), and Shapley values (Lundberg and Lee 2017). A global method describes the average behavior of a black-box model. Partial dependence plots (Friedman 2001), variable importance plots (Breiman 2001; Fisher, Rudin, and Dominici 2019), and global surrogate models (Bastani, Kim, and Bastani 2017) are some of the popular choices.

See [H2OML] **h2omlgraph ice**, [H2OML] **h2omlgraph shapvalues**, and [H2OML] **h2omlgraph shapsummary** for a few local model-agnostic methods and [H2OML] **h2omlgraph pdp** and [H2OML] **h2omlgraph varimp** for global model-agnostic methods. We also describe the global surrogate models in the next section.

Global surrogate models

Global surrogate models (Bastani, Kim, and Bastani 2017 and Craven and Shavlik 1995) are explainable models that approximate the predictions of a black-box model. In other words, a surrogate model uses an interpretable model to explain a black-box model. The steps for obtaining a global surrogate model are straightforward:

1. Obtain predictions from a well-tuned black-box model fit to the testing data.
2. Select and train an interpretable model (for example, a decision tree) for predictions on the testing data.
3. Measure the goodness of fit of the surrogate model for the predictions, and interpret the model.

One way to measure the goodness of fit of a surrogate model for predictions is by using the R^2 for regression and accuracy or log loss for classification,

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\{\widehat{g}(\mathbf{x}_i) - \widehat{f}(\mathbf{x}_i)\}^2}{\sum_{i=1}^{n}\{\widehat{f}(\mathbf{x}_i) - \overline{f}\}^2}$$

where $\widehat{g}(\cdot)$ and $\widehat{f}(\cdot)$ are the respective predictions from the surrogate and black-box models and $\overline{f}$ is the mean of the black-box predictions. The larger the R^2, the better the surrogate model replicates the black-box model.

For example, suppose we used a GBM to obtain predictions of housing prices. We could then apply the above method to explain its predictions by using a decision tree as a surrogate model. We show one such tree below. We can easily see how the predictors explain the predicted log sales prices. The terminal nodes of the tree show the predicted logarithm of the sales prices. For example, the houses with overall quality (`overallqual`) greater than 7.5 and with the lot area (`lotarea`) greater than 12,332.5 square feet have the highest predicted price of 12.74.

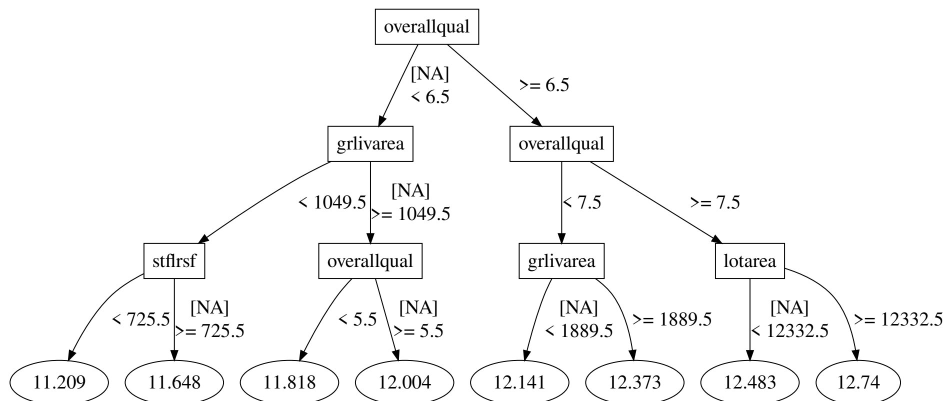


Figure 12.

References

Abreu, S. 2019. Automated architecture design for deep neural networks. arXiv:1908.10714 [cs.LG], https://doi.org/10.48550/arXiv.1908.10714.

Bastani, O., C. Kim, and H. Bastani. 2017. Interpreting blackbox models via model extraction. arXiv:1705.08504 [cs.LG], https://doi.org/10.48550/arXiv.1705.08504.

Belkin, M., D. Hsu, S. Ma, and S. Mandal. 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences* 116: 15849–15854. https://doi.org/10.1073/pnas.1903070116.

Ben-Haim, Y., and E. Tom-Tov. 2010. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research* 11: 849–872.

Bergstra, J., R. Bardenet, Y. Bengio, and B. Kégl. 2011. “Algorithms for hyper-parameter optimization”. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, edited by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, vol. 24: 2546–2554. Red Hook, NY: Curran Associates.

Bergstra, J., and Y. Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13: 281–305.

Biau, G., and E. Scornet. 2016. A random forest guided tour. *TEST* 25: 197–227. https://doi.org/10.1007/s11749-016-0481-7.

Borisov, V., T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci. 2024. Deep neural networks and tabular data: A survey. *IEEE Transactions on Neural Networks and Learning Systems* 35: 7499–7519. https://doi.org/10.1109/tnnls.2022.3229161.

Breiman, L. 1996. Bagging predictors. *Machine Learning* 24: 123–140. https://doi.org/10.1007/BF00058655.

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

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Boca Raton, FL: Chapman and Hall/CRC.

Cawley, G. C., and N. L. C. Talbot. 2010. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research* 11: 2079–2107.

Chen, C., K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang. 2018. An interpretable model with globally consistent explanations for credit risk. arXiv:1811.12615 [cs.LG], https://doi.org/10.48550/arXiv.1811.12615.

Chen, T., and C. Guestrin. 2016. “XGBoost: A scalable tree boosting system”. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939785.

Chollet, F. 2021. *Deep Learning with Python.* 2nd ed. Shelter Island, NY: Manning Publications.

Craven, M. W., and J. W. Shavlik. 1995. "Extracting tree-structured representations of trained networks". In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, edited by D. Touretzky, M. C. Mozer, and M. Hasselmo, 24–30. Cambridge, MA: MIT Press.

Curth, A., A. Jeffares, and M. van der Schaar. 2024. Why do random forests work? Understanding tree ensembles as self-regularizing adaptive smoothers. arXiv:2402.01502 [stat.ML], https://doi.org/10.48550/arXiv.2402.01502.

Doshi-Velez, F., and B. Kim. 2017. Towards a rigorous science of interpretable machine learning. arXiv:1702.08608 [stat.ML], https://doi.org/10.48550/arXiv.1702.08608.

Efron, B. 1979. Bootstrap methods: Another look at the jackknife. *Annals of Statistics* 7: 1–26. https://doi.org/10.1214/aos/1176344552.

Efron, B., and R. J. Tibshirani. 1993. *An Introduction to the Bootstrap*. New York: Chapman and Hall/CRC. https://doi.org/10.1201/9780429246593.

Elith, J., J. R. Leathwick, and T. J. Hastie. 2008. A working guide to boosted regression trees. *Journal of Animal Ecology* 77: 802–813. https://doi.org/10.1111/j.1365-2656.2008.01390.x.

Elsken, T., J. H. Metzen, and F. Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* 20: 1–21.

Fisher, A., C. Rudin, and F. Dominici. 2019. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research* 20: 1–81.

Freund, Y., and R. E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences* 55: 119–139. https://doi.org/10.1006/jcss.1997.1504.

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

Friedman, J. H., T. J. Hastie, and R. J. Tibshirani. 2000. Additive logistic regression: A statistical view of boosting. *Annals of Statistics* 28: 337–407. https://doi.org/10.1214/aos/1016218223.

Friedman, J. H., and B. E. Popescu. 2008. Predictive learning via rule ensembles. *Annals of Applied Statistics* 2: 916–954. https://doi.org/10.1214/07-AOAS148.

Ghassemi, M., L. Oakden-Rayner, and A. L. Beam. 2021. The false hope of current approaches to explainable artificial intelligence in health care. *Lancet* 3: E745–E750.

Goldstein, A., A. Kapelner, J. Bleich, and E. Pitkin. 2015. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics* 24: 44–65. https://doi.org/10.1080/10618600.2014.907095.

Guidotti, R., A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti. 2018. A survey of methods for explaining black box models. arXiv:1802.01933 [cs.CY], https://doi.org/10.48550/arXiv.1802.01933.

Hastie, T. J., R. J. Tibshirani, and J. H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer. https://doi.org/10.1007/978-0-387-84858-7.

Izenman, A. J. 2008. *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning*. New York: Springer. https://doi.org/10.1007/978-0-387-78189-1.

James, G., D. Witten, T. J. Hastie, and R. J. Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in R*. 2nd ed. New York: Springer. https://doi.org/10.1007/978-1-0716-1418-1.

Ke, G., Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. 2017. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, vol. 30: 3149–3157. Red Hook, NY: Curran Associates.

Krishna, S., T. Han, A. Gu, S. Wu, S. Jabbari, and H. Lakkaraju. 2022. The disagreement problem in explainable machine learning: A practitioner's perspective. arXiv:2202.01602 [cs.LG], https://doi.org/10.48550/arXiv.2202.01602.

Kuhn, M., and K. Johnson. 2013. *Applied Predictive Modeling*. New York: Springer. https://doi.org/10.1007/978-1-4614-6849-3.

Lakkaraju, H., S. H. Bach, and J. Leskovec. 2016. "Interpretable decision sets: A joint framework for description and prediction". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1675–1684. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939874.

Lakkaraju, H., and O. Bastani. 2020. "“How do I fool you?”: Manipulating user trust via misleading black box explanations". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 79–85. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375833.

Letham, B., C. Rudin, T. H. McCormick, and D. Madigan. 2015. Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model. *Annals of Applied Statistics* 9: 1350–1371. http://doi.org/10.1214/15-AOAS848.

Lin, Y., and Y. Jeon. 2006. Random forests and adaptive nearest neighbors. *Journal of the American Statistical Association* 101: 578–590. https://doi.org/10.1198/016214505000001230.

Lones, M. A. 2021. How to avoid machine learning pitfalls: A guide for academic researchers. arXiv:2108.02497 [cs.LG], https://doi.org/10.48550/arXiv.2108.02497.

Lundberg, S. M., and S. Lee. 2017. "A unified approach to interpreting model predictions". In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, vol. 30: 4768–4777. Red Hook, NY: Curran Associates.

Meinshausen, N. 2006. Quantile regression forests. *Journal of Machine Learning Research* 7: 983–999.

Potharst, R., and A. J. Feelders. 2002. Classification trees for problems with monotonicity constraints. *ACM SIGKDD Explorations Newsletter* 4: 1–10. https://doi.org/10.1145/568574.568577.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann.

Raji, I. D., A. Smart, R. N. White, M. Mitchell, T. Gebru, B. Hutchinson, J. Smith-Loud, D. Theron, and P. Barnes. 2020. "Closing the AI accountability gap: Defining an end-to-end framework for internal algorithmic auditing". In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, 33–44. New York: Association for Computing Machinery. https://doi.org/10.1145/3351095.3372873.

Raschka, S. 2020. Model evaluation, model selection, and algorithm selection in machine learning. arXiv:1811.12808 [cs.LG], https://doi.org/10.48550/arXiv.1811.12808.

Ribeiro, M. T., S. Singh, and C. Guestrin. 2016. "“Why should I trust you?”: Explaining the predictions of any classifier". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144. New York: Association for Computing Machinery. https://doi.org/10.1145/2939672.2939778.

Rifkin, R., and A. Klautau. 2004. In defense of one-vs-all classification. *Journal of Machine Learning Research* 5: 101–141.

Rudin, C. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1: 206–215. https://doi.org/10.1038/s42256-019-0048-x.

Rudin, C., and B. Ustun. 2018. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *INFORMS Journal on Applied Analytics* 48: 399–486. https://doi.org/10.1287/inte.2018.0957.

Schapire, R. E. 2003. "The boosting approach to machine learning: An overview". In *Nonlinear Estimation and Classification*. Lecture Notes in Statistics, edited by D. D. Denison, M. H. Hansen, C. C. Holmes, B. Mallick, and B. Yu, vol. 171: 149–171. New York: Springer. https://doi.org/10.1007/978-0-387-21579-2_9.

Schapire, R. E., and Y. Freund. 2012. *Boosting: Foundations and Algorithms*. Cambridge, MA: MIT Press.

Shmuel, A., O. Glickman, and T. Lazebnik. 2024. A comprehensive benchmark of machine and deep learning across diverse tabular datasets. arXiv:2408.14817 [cs.LG], https://doi.org/10.48550/arXiv.2408.14817.

Shwartz-Ziv, R., and A. Armon. 2022. Tabular data: Deep learning is not all you need. *Information Fusion* 81: 84–90. https://doi.org/10.1016/j.inffus.2021.11.011.

Slack, D., S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju. 2020. "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 180–186. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375830.

Wager, S., and S. Athey. 2018. Estimation and inference of heterogeneous treatment effects using random forests. *Journal of the American Statistical Association* 113: 1228–1242. https://doi.org/10.1080/01621459.2017.1319839.

Wyner, A. J., M. Olson, J. Bleich, and D. Mease. 2017. Explaining the success of AdaBoost and random forests as interpolating classifiers. *Journal of Machine Learning Research* 18: 1–33.

Yang, L., and A. Shami. 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing* 415: 295–316. https://doi.org/10.1016/j.neucom.2020.07.061.

Also see

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

[H2OML] **Glossary**

h2oml — Introduction to commands for Stata integration with H2O machine learning

Description   Remarks and examples   References   Also see

Description

This entry describes commands for performing predictive analysis using H2O machine learning methods, specifically ensemble decision tree methods, in Stata. H2O is a scalable and distributed machine learning and predictive analytics platform that allows you to perform data analysis and machine learning. It provides parallelized implementations of many widely used supervised and unsupervised machine learning methods. For more details, see [H2OML] **H2O setup**, [P] **H2O intro**, and https://www.stata.com/h2o/h2o18/h2o_intro.html#what-is-h2o. For a software-free introduction to machine learning, see [H2OML] **Intro**.

Supervised learning

h2oml gbm	gradient boosting machine
`h2oml gbregress`	gradient boosting regression
`h2oml gbbinclass`	gradient boosting binary classification
`h2oml gbmulticlass`	gradient boosting multiclass classification
h2oml rf	random forest
`h2oml rfregress`	random forest regression
`h2oml rfbinclass`	random forest binary classification
`h2oml rfmulticlass`	random forest multiclass classification

Estimation results and postestimation frame

`h2omlest store`	catalog H2O estimation results
`h2omlpostestframe`	specify frame for postestimation analysis

Tuning and estimation summaries

`h2omlestat metrics`	display performance metrics
`h2omlgof`	goodness of fit for machine learning methods
`h2omlestat cvsummary`	display cross-validation summary
`h2omlestat gridsummary`	display grid-search summary
`h2omlexplore`	explore models after grid search
`h2omlselect`	select model after grid search
`h2omlgraph scorehistory`	produce score history plot

Performance after binary classification	
h2omlestat threshmetric	display threshold-based metrics
h2omlestat confmatrix	display confusion matrix
h2omlgraph prcurve	produce precision–recall curve plot
h2omlgraph roc	produce ROC curve plot

Performance after multiclass classification	
h2omlestat aucmulticlass	display AUC and AUCPR summary
h2omlestat confmatrix	display confusion matrix
h2omlestat hitratio	display hit-ratio table

Prediction	
h2omlpredict	prediction of continuous responses, probabilities, and classes

Machine learning explainability	
h2omlgraph varimp	produce variable importance plot
h2omlgraph pdp	produce partial dependence plot
h2omlgraph ice	produce individual conditional expectation plot
h2omlgraph shapvalues	produce SHAP values plot for individual observations after regression and binary classification
h2omlgraph shapsummary	produce SHAP beeswarm plot after regression and binary classification

Save decision tree	
h2omltree	save decision tree DOT file and display rule set

Remarks and examples

This entry describes Stata commands to perform predictive analysis using H2O machine learning ensemble decision tree methods.

Remarks and examples are presented under the following headings:

Brief overview
h2oml in a nutshell
Tour of machine learning commands
- *Prepare your data for H2O machine learning in Stata*
- *End-to-end binary classification analysis*
- *Regression analysis*
- *Effect of categorical predictors*
- *Detecting nuisance predictors*
- *Gradient boosting Poisson regression*

Brief overview

The h2oml suite of Stata commands provides end-to-end support for H2O machine learning analysis using ensemble decision tree methods. In addition to h2oml, the _h2oframe command provides several key subcommands that connect Stata to an H2O cluster, import a Stata dataset into an H2O frame, and provide various H2O data management; see [H2OML] **H2O setup**.

h2oml *gbm* and h2oml *rf* provide the suite of estimation commands that implement gradient boosting and random forest regression, binary classification, and multiclass classification. h2oml gbregress and h2oml rfregress perform respective gradient boosting and random forest regressions for continuous and count responses, h2oml gbbinclass and h2oml rfbinclass perform gradient boosting and random forest classifications for binary responses, and h2oml gbmulticlass and h2oml rfmulticlass perform gradient boosting and random forest classifications for categorical responses (with more than two categories).

All commands provide the validframe() and cv() options to specify a validation frame and to perform cross-validation to control for overfitting, the tune() and stop() options to tune hyperparameters and stop early for better model performance, the h2orseed() option to reproduce results, and many more. Many commands also offer specialized options such as the loss() option of h2oml gbregress, which specifies various loss functions, including quantile, Huber, and Tweedie. See [H2OML] ***h2oml gbm*** and [H2OML] ***h2oml rf*** for details.

After estimation, the h2omlest suite of commands can be used to manage estimation results. For instance, h2omlest store can be used to store the current estimation results for later use.

Several postestimation commands are available to obtain tuning and estimation summaries. For instance, h2omlestat gridsummary is useful to view the results after tuning and select an alternative model that is more parsimonious. And h2omlgraph scorehistory can be used to display various validation curves to help monitor overfitting.

For binary and multiclass classifications, several commands can be used to explore model performance such as the h2omlestat confmatrix command, which displays the confusion matrix. Additionally, h2omlgraph prcurve and h2omlgraph roc can be used to plot precision–recall and receiver operating characteristic (ROC) curves after binary classification, and h2omlestat hitratio can be used to produce a hit-ratio table after multiclass classification.

The ultimate goal of machine learning is to obtain accurate prediction of the response on the new data. To achieve this goal, the model predictive performance is often evaluated by using an external, testing dataset. The h2omlpostestframe command provides a convenient way to specify the desired testing frame to be used in all subsequent postestimation analyses.

Depending on the estimation method, regression or classification, the h2omlpredict command produces predictions of continuous and count responses or class probabilities and classes.

Machine learning methods are often treated as a black box, meaning that little attempt is made to understand the obtained predictions. To rectify this, h2oml provides several postestimation commands to help explain predictions. The h2omlgraph varimp command can be used to assess the overall importance of predictors in the model, whereas the h2omlgraph shapvalues and h2omlgraph shapsummary commands can be used to explore the impact of predictors on individual predictions.

Finally, the h2omltree command can be used to save a specific decision tree in a DOT file and plot it by using the open source software Graphviz; see [H2OML] **DOT extension**.

For more details about postestimation commands, see [H2OML] **h2oml postestimation**.

h2oml in a nutshell

In the previous section, we briefly described the functionality of the `h2oml` command. Here we will provide a quick overview of some of the more common usages of this command in practice.

As we mentioned earlier, machine learning is primarily used to develop a model that accurately predicts a response of interest on the new data. In practice, several general steps are often performed to build such a model.

At the beginning of the analysis, the data are often split into training data used for estimation and validation data used for evaluating the model performance. Additionally, external testing data are also available for assessing the model final predictive performance and comparing it with other models that use a different machine learning method such as gradient boosting machine (GBM) or random forest. For each method, models with different sets of hyperparameters are evaluated using a validation dataset (or cross-validation), and the best model is chosen. The chosen models are further evaluated based on their predictive performance on the testing data, and the final model is selected for later prediction on the future new data.

Below, we describe several `h2oml` commands that can be used to perform the above steps.

Setup. To use the `h2oml` command, we must first initialize an H2O cluster and import our data to an H2O frame; see *Prepare your data for* H2O *machine learning in Stata* and [H2OML] **H2O setup**. Here we load the current Stata dataset into the H2O `data` frame and make it the current H2O frame.

```
. h2o init
. _h2oframe put, into(data)
. _h2oframe change data
```

Alternatively, we could replace the last two commands with `_h2oframe put, into(data) current` to put the dataset into an H2O frame and make this frame current in a single step.

Next we split the `data` frame into training and validation with, say, 80% of observations in the training sample. We also specify the random-number seed for reproducibility and make the `train` frame be the current H2O frame for estimation.

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

Depending on the type of a response and the desired machine learning method, we can choose one of the six `h2oml` commands to perform estimation: `h2oml gbregress`, `h2oml gbbinclass`, `h2oml gbmulticlass`, `h2oml rfregress`, `h2oml rfbinclass`, and `h2oml rfmulticlass`.

Reference or baseline model. Suppose we have a binary response and we want to use GBM. We can start with a simple reference model with default hyperparameters:

```
. h2oml gbbinclass response predictors, h2orseed(19) validframe(valid)
```

We specified the `h2orseed(19)` option to ensure H2O reproducibility; see [H2OML] **H2O reproducibility**.

If we do not have sufficient observations to split the data into training and validation, we can use cross-validation instead such as a 3-fold cross-validation with the default random splitting of the data below:

```
. h2oml gbbinclass response predictors, h2orseed(19) cv(3)
```

We store the current estimation results to use as a benchmark later.

```
. h2omlest store gbm_ref
```

User-specified hyperparameters and tuning. Next we can explore models with values of hyperparameters other than the default ones. For instance, we can specify 200 trees instead of the default 50 and a 0.2 learning rate instead of the default 0.1. And we can specify different values for any of the other nine hyperparameters; see *Options* in [H2OML] ***h2oml gbm***.

```
. h2oml gbbinclass response predictors, h2orseed(19) cv(3)
> ntrees(200) lrate(0.2) ...
```

We store this model as

```
. h2omlest store gbm_user
```

In practice, it is difficult to know the actual hyperparameter values that provide the best model performance, so an iterative procedure known as hyperparameter tuning is used to explore different ranges of various hyperparameters to select the best set of values. To incorporate tuning, the `h2oml` estimation commands allow you to specify the ranges (*numlist*) in options for hyperparameters and provide the `tune()` option to control the tuning procedure.

Which hyperparameters should be tuned and what ranges should be explored will be specific to each application. Here, for illustration purposes and continuing with our example, we will tune the number of trees and the learning rate:

```
. h2oml gbbinclass response predictors, h2omlrseed(19) cv(3)
> ntrees(20(10)200) lrate(0.1(0.1)1)
```

We store this tuned model as

```
. h2omlest store gbm_tuned
```

If desired, we can change the default tuning metric (from log loss to, say, accuracy) and grid-search method (from Cartesian to random) as well as specify other suboptions in the `tune()` option:

```
. h2oml gbbinclass response predictors, h2omlrseed(19) cv(3)
> ntrees(20(10)200) lrate(0.1(0.1)1)
> tune(metric(accuracy) grid(random) ...)
```

Checking for overfitting or underfitting. Before we proceed with model selection, we can check for model overfitting or underfitting. We can use the `h2omlgraph scorehistory` command to plot the metric values against the number of trees to compare the training and validation or cross-validation curves:

```
. h2omlgraph scorehistory
```

The number of trees at which the two curves start noticeably diverging provides a tradeoff between underfitting and overfitting.

Because we performed cross-validation, it is also useful to evaluate its performance. We can check the variability of the metric values across the folds with

```
. h2omlestat cvsummary
```

High variation may indicate overfitting.

Our current model is `gbm_tuned`, but we can repeat the above steps for the other two models by first using the `h2omlest restore` command to restore their estimation results.

Selecting the "best" model. Our current gbm_tuned model uses the hyperparameter values that resulted in the smallest value of the default log loss metric. We can evaluate alternative models that may be more parsimonious and thus may run faster:

```
. h2omlestat gridsummary
```

We can also explore the performance of additional metrics for different models before deciding on a model. For instance, we can explore the top 10 models:

```
. h2omlexplore id = 1(1)10
```

If we find an alternative model that we think is best, we can switch to it by using

```
. h2omlselect id = #
```

where # is an index of the corresponding model from h2omlestat gridsummary.

To select between all the considered models with different hyperparameters such as gbm_tuned and gbm_user, we select the one with the most optimal metric value, which is reported in the output of the h2oml estimation commands. We can also use

```
. h2omlestat metrics
```

to report the performance metrics for the current estimation model.

And we can compare different metrics side by side for all models more easily by using

```
. h2omlgof gbm_tuned gbm_user gbm_ref
```

Evaluate predictive performance and compare different methods. Predictive performance of a model is typically evaluated on an external testing dataset. The h2omlpostestframe command provides a convenient way of specifying a testing frame for all postestimation analyses:

```
. h2omlpostestframe test
```

Here test is our H2O testing frame. This command does not physically change the current frame from train to test. It instead specifies that all relevant postestimation commands use the test frame in the computations instead of their specific default frames, which may be training, validation, or cross-validation depending on the estimation.

After binary or multiclass classification, we can evaluate model predictive performance by using the confusion matrix:

```
. h2omlestat confmatrix
```

After binary classification, we can also explore thresholds that are optimal for various metrics

```
. h2omlestat threshmetric
```

Here we chose to use a GBM method. We can also consider using a random forest method. We would repeat all the above steps but now using the rfbinclass command for estimation to select the best random forest model, say rf_tuned. We would then use the above commands to compare the predictive performances of the two models or use

```
. h2omlgof gbm_tuned rf_tuned
```

to compare different performance metrics side by side. We can compare different methods using precision–recall and ROC curves:

```
. h2omlgraph prcurve, models(gbm_tuned rf_tuned)
. h2omlgraph roc, models(gbm_tuned rf_tuned)
```

Obtain predictions. Once the best model is chosen, we can use it to compute predictions. Depending on the research question, we can compute predictions for an entirely new dataset, or we can use the original data. Here we obtain predictions for our original `data` frame.

```
. _h2oframe change data
. h2omlpredict
```

Explain predictions. The `h2oml` suite provides several commands for explaining predictions. We can evaluate overall predictors' importance that quantifies the effect of each predictor on the model's predictions:

```
. h2omlgraph varimp
```

We can also use the partial dependence plot (PDP) and the individual conditional expectation (ICE) plot to visually explore predictor dependence on the response:

```
. h2omlgraph pdp  predictors
. h2omlgraph ice  predictor
```

And, after regression and binary classification, we can use Shapley additive explanations (SHAP) values to explore predictor contributions to the prediction of the response:

```
. h2omlgraph shapvalues
. h2omlgraph shapsummary
```

Tour of machine learning commands

In this section, we illustrate the usage of the `h2oml` command with applications to several real-world datasets. We start by showing how to start an H2O cluster and convert your Stata dataset into an H2O frame. We then illustrate the basic steps for training machine learning methods and provide predictions for binary classification and for regression. We also explore the effect of categorical predictors on the performance of ensemble decision tree methods and demonstrate how to use these methods to detect important predictors. We also show a quick analysis of a count response by using a gradient boosting Poisson regression.

Examples are presented under the following headings:

Prepare your data for H2O machine learning in Stata
End-to-end binary classification analysis
  Example 1: Data setup
  Example 2: Reference binary classification using GBM
  Example 3: Model selection and hyperparameter tuning
  Example 4: Method selection and prediction
  Example 5: Classification prediction on new data
  Example 6: Explaining classification prediction
  Example 7: Shutting down the H2O cluster
Regression analysis
  Example 8: Data setup
  Example 9: Regression using random forest
  Example 10: Hyperparameter tuning using random forest
Effect of categorical predictors
  Example 11: Data setup
  Example 12: Effect of categorical predictors on ensemble decision tree methods
Detecting nuisance predictors
  Example 13: Detecting nuisance predictors with ensemble decision tree methods
Gradient boosting Poisson regression
  Example 14: Explaining Poisson regression predictions

Prepare your data for H2O machine learning in Stata

Before using any of the H2O machine learning methods in Stata, you need to connect to or initialize an H2O server by using the h2o init command. The command first checks whether an H2O cluster is already running on the local machine and uses that cluster if so; otherwise, it attempts to start a new cluster. For details, see [H2OML] **H2O setup**.

We first use the h2o init command to start an H2O cluster.

```
. h2o init
```

Suppose we have an external data.csv file saved in Stata's current directory. We can import it as an H2O frame by typing

```
. _h2oframe import data.csv, into(data)
```

or if we already have our data loaded into Stata, we can store it as an H2O frame by typing

```
. _h2oframe put, into(data)
```

In the above, we put our data into the H2O cluster as an H2O frame and called it data. To be able to work with the data frame, we need to change it to be the current working frame:

```
. _h2oframe change data
```

Before starting any H2O analysis, we recommend that you describe the data to ensure that the H2O variable types are as expected. This is important because the implementation of H2O machine learning methods can vary depending on the types of the response and predictors.

```
. _h2oframe describe
```

Suppose our data have two variables: y and x. To run a regression for y on x using GBM with default settings, we can now type

```
. h2oml gbregress y x
```

Or we can use random forest with default settings by typing

```
. h2oml rfregress y x
```

After estimation, we can use any postestimation command from [H2OML] **h2oml postestimation**.

End-to-end binary classification analysis

In this section, we provide an end-to-end analysis for a binary classification problem using gradient boosting binary classification. The examples comprise tuning, performance analysis, and prediction explainability.

▷ Example 1: Data setup

Consider data from a fictional company, Telco, that provides home phone and internet services in California. The data have been made available by IBM. We want to build a predictive model to predict the behavior of a customer who is more likely to churn. churn.dta contains 7,043 observations and 26 variables. The binary response churn indicates whether a customer left within the last month or is still using Telco's services. The predictors include customers' demographic information such as gender and age, customers' account information such as payment period and duration of services, customers' service types such as whether a customer signed up for internet, phone, device protection, etc.

The goal of this example is to build a predictive model that will predict the behavior of a customer who is more likely to churn or retain the company's services.

As we described in *Prepare your data for H2O machine learning in Stata*, we start by reading the dataset as an H2O frame. We then describe the frame to make sure that variables (H2O columns) have the intended data types by using the `_h2oframe describe` command. Recall that `h2o init` initiates an H2O cluster and `_h2oframe put` loads the current Stata dataset into an H2O frame. For details, see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/churn
(Telco customer churn data)
. h2o init
  (output omitted)
. _h2oframe put, into(churn)
Progress (%): 0 100
. _h2oframe change churn
. _h2oframe describe
          Rows:      7043
          Cols:        26
```

Column	Type	Missing	Zeros	+Inf	-Inf	Cardinality
zipcode	int	0	0	0	0	
latitude	real	0	0	0	0	
longitude	real	0	0	0	0	
tenuremonths	int	0	11	0	0	
monthlycharges	real	0	0	0	0	
totalcharges	real	11	0	0	0	
country	enum	0	7043	0	0	1
state	enum	0	7043	0	0	1
city	enum	0	4	0	0	1129
gender	enum	0	3488	0	0	2
seniorcitizen	enum	0	5901	0	0	2
partner	enum	0	3641	0	0	2
dependents	enum	0	5416	0	0	2
phoneservice	enum	0	682	0	0	2
multiplelines	enum	0	3390	0	0	3
internetserv	enum	0	2421	0	0	3
onlinesecurity	enum	0	3498	0	0	3
onlinebackup	enum	0	3088	0	0	3
deviceprotect	enum	0	3095	0	0	3
techsupport	enum	0	3473	0	0	3
streamtv	enum	0	2810	0	0	3
streammovie	enum	0	2785	0	0	3
contract	enum	0	3875	0	0	3
paperlessbill	enum	0	2872	0	0	2
paymethod	enum	0	1544	0	0	4
churn	enum	0	5174	0	0	2

For definitions of data types in H2O, see https:/www.stata.com/h2o/h2oframe_intro.html. Specifically, `enum` refers to categorical or factor columns in an H2O frame, `real` to numeric columns with `float` or `double` values, and `int` to numeric columns with integer values. For example, here `churn` has the expected type `enum`. If the data types are incorrect, `_h2oframe` provides commands to convert an H2O frame column to the desired data type; see https://www.stata.com/h2o/h2oframe.html. You may notice that the predictor `totalcharges` has 11 missing values. As we discussed in *Decision trees* of [H2OML] **Intro**, tree-based methods naturally handle missing values.

Next we split our data into training and testing frames with 80% of observations in the training sample. We will use cross-validation on training data during estimation to control for overfitting.

```
. _h2oframe split churn, into(train test) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

◁

▷ Example 2: Reference binary classification using GBM

As we discussed in *Model selection in machine learning* of [H2OML] **Intro**, the analysis should start by defining a baseline or reference performance.

For classification problems, it is recommended to first check whether the dataset is imbalanced.

```
. tabulate churn

   Churning |
     status |      Freq.     Percent        Cum.
------------+-----------------------------------
         No |      5,174       73.46       73.46
        Yes |      1,869       26.54      100.00
------------+-----------------------------------
      Total |      7,043      100.00
```

Our dataset suffers from imbalance. Therefore, we will use the stratification method for cross-validation to ensure that the cross-validation samples maintain the same data imbalance. Following the literature on measuring performance for imbalanced data (Davis and Goadrich 2006), we will use area under the precision–recall curve (AUCPR) as a performance metric in our analysis.

Next, for convenience, let's 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
```

As a reference model, we fit a GBM model with a 3-fold stratified cross-validation and default values for other settings. We specify the h2orseed(19) option for reproducibility; see [H2OML] **H2O reproducibility**.

```
. h2oml gbbinclass churn $predictors, h2orseed(19) cv(3, stratify)
Progress (%): 0 42.5 87.0 100
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
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 =   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	Cross-validation
Log loss	.3293387	.411338
Mean class error	.1603572	.2338787
AUC	.9163226	.8500772
AUCPR	.8023966	.6584908
Gini coefficient	.8326452	.7001545
MSE	.1034999	.1350446
RMSE	.321714	.3674841

For detailed interpretation of the output, see example 1 of [H2OML] ***h2oml gbm***.

Although we are mainly interested in cross-validation metrics, we still need to examine the training metrics to make sure that we slightly overfit the training data to avoid underfitting. The latter can be checked by exploring the difference between training and cross-validation metrics, which should be positive for the AUCPR metric. However, if the difference between the validation and training metrics is large, it indicates that the model is too tailored to the training data and may not generalize well to new data. In the literature, there is no clear recommendation on how large the difference between training and validation metrics should be to indicate severe overfitting. Each case should be evaluated individually and with caution. For details, see Valdenegro-Toro and Sabatelli (2023). In our example, the positive difference between the training and cross-validation AUCPR values suggests that our model does overfit the training data. The cross-validation AUCPR for the reference model is approximately 0.658.

We store the reference estimation results for later comparison using the h2omlest store command.

```
. h2omlest store gbm_default
```

It is helpful to assess the variance of each metric over the folds to ensure that the model performance does not depend on the specific split of the data. Large variation of the cross-validation metrics over the folds may lead to poor generalization of the model to new data. In such cases, it is recommended to adjust the number of folds or examine the data to identify the sources of variability. We can use `h2omlestat cvsummary` to display cross-validation summary.

```
. h2omlestat cvsummary
```

Cross-validation summary using H2O

Metric	Mean	Std. dev.	Fold 1	Fold 2
Log loss	.4113427	.0038855	.4085804	.4157856
F1	.6401071	.0044256	.6358885	.6397188
F2	.6954293	.0055981	.6891994	.6970509
F0.5	.5929428	.0039657	.5902329	.591101
Accuracy	.7806169	.0012531	.7793031	.7817988
Precision	.5651822	.0039084	.5632716	.5625966
Recall	.7379531	.0069124	.73	.7413442
Specificity	.7959458	.0011321	.7969871	.7961095
Misclassification	.2193831	.0012531	.2206969	.2182012
Mean class error	.2330506	.0029933	.2365065	.2312731
Max. class error	.2620469	.0069124	.27	.2586558
Mean class accuracy	.7669494	.0029933	.7634935	.7687268
Misclassification count	412.6667	4.618802	418	410
AUC	.8505131	.0040418	.8526636	.8458507
AUCPR	.6597555	.0045358	.6628664	.654551
MSE	.1350454	.0017733	.1340862	.1370917
RMSE	.3674799	.0024083	.3661779	.370259

Metric	Fold 3
Log loss	.4096621
F1	.6447141
F2	.7000377
F0.5	.5974944
Accuracy	.7807487
Precision	.5696784
Recall	.742515
Specificity	.7947407
Misclassification	.2192513
Mean class error	.2313722
Max. class error	.257485
Mean class accuracy	.7686278
Misclassification count	410
AUC	.8530251
AUCPR	.6618491
MSE	.1339582
RMSE	.3660029

In our example, the variation of the cross-validation metrics across folds, that is, AUCPR, is small. The mean value of the cross-validation AUCPR is around 0.660, which is slightly different from the cross-validation AUCPR of 0.658 reported by `h2oml gbbinclass`. This difference is expected because of how

the two commands compute cross-validation metrics. `h2omlestat cvsummary` computes metrics separately for each fold and reports their average value, whereas `h2oml gbbinclass` combines all folds into one and computes a single AUCPR value.

▷ Example 3: Model selection and hyperparameter tuning

Hyperparameters, such as the number of trees and learning rate, control the performance of a machine learning model. Choosing the "right" hyperparameters can substantively improve both the model performance and its ability to be generalized to new data. Poorly selected hyperparameters, on the other hand, can lead to underfitting or overfitting. The process of selecting hyperparameters to achieve optimal model performance is known as hyperparameter tuning.

In example 5 of [H2OML] ***h2oml gbm***, we demonstrated the detailed steps of hyperparameter tuning for this example. Here we use the final selected model:

```
. h2oml gbbinclass churn $predictors, h2orseed(19) cv(3, stratify)
> ntrees(100) lrate(0.05) predsamprate(0.15)
Progress (%): 0 36.0 71.2 89.4 100
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
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
```

Metric	Training	Cross-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

By tuning, we increased the cross-validation AUCPR from 0.658 to 0.674. The improvement is small, because we explored only a small portion of the hyperparameter space in this example. Hyperparameter tuning is an iterative process that requires many iterations to sufficiently explore the hyperparameter space.

Let's compare the best model, which we store as gbm_tuned, with the reference model from the previous example based on other metrics by using the h2omlgof command.

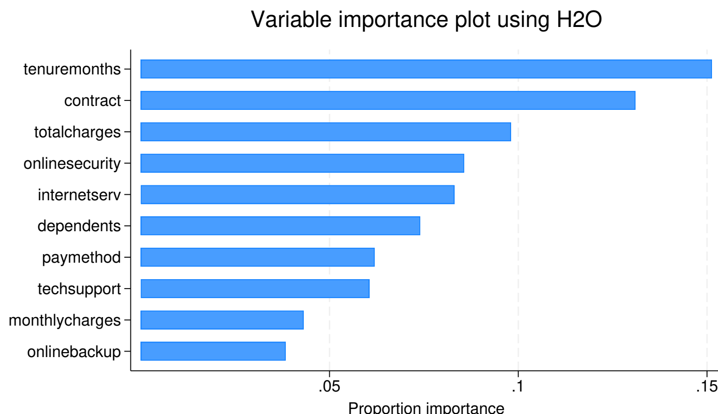
```
. h2omlest store gbm_tuned
. h2omlgof gbm_default gbm_tuned
Performance metrics for model comparison using H2O
Training frame: train
```

	gbm_def~t	gbm_tuned
Training		
No. of observations	5,643	5,643
Log loss	.3293387	.3531063
Mean class error	.1603572	.1784776
AUC	.9163226	.8992847
AUCPR	.8023966	.7610732
Gini coefficient	.8326452	.7985693
MSE	.1034999	.1126847
RMSE	.321714	.3356854
Cross-validation		
No. of observations	5,643	5,643
Log loss	.411338	.4026141
Mean class error	.2338787	.2313897
AUC	.8500772	.8565935
AUCPR	.6584908	.673929
Gini coefficient	.7001545	.7131869
MSE	.1350446	.1314475
RMSE	.3674841	.3625569

In the output, the first section reports the training results, and the second section reports the cross-validation results. Looking at the cross-validation results, we see that tuning improved the model performance for all metrics. The log loss, mean of per-class error rates, mean squared error (MSE), and root mean squared error (RMSE) are all smaller for the tuned model, whereas area under the curve (AUC), AUCPR, and the Gini coefficient are larger for the tuned model, all of which indicate better performance.

In addition to tuning, we may also refine the list of predictors based on variable importance.

```
. h2omlgraph varimp
```



Based on the above graph, we may decide to drop the predictor onlinebackup.

Variable selection with cross-validation requires careful implementation to avoid so-called data leakage, where the training data contain information that would not be available during prediction on the testing data; see Raschka (2020) for details.

◁

▷ Example 4: Method selection and prediction

In example 5 of [H2OML] ***h2oml gbm***, we used hyperparameter tuning to select the best GBM model. Instead of GBM, we may consider other methods such as random forest or logistic regression. In this example, we compare GBM and random forest.

Instead of tuning the random forest model following similar steps from example 5 of [H2OML] ***h2oml gbm***, for simplicity, we pretend that the following model is our tuned model for random forest. We also store the working model as `rf_tuned` by using the `_h2omlest store` command.

```
. h2oml rfbinclass churn $predictors, h2orseed(19) cv(3, stratify)
> ntrees(200) minobsleaf(2)
Progress (%): 0 7.1 14.1 19.8 24.8 56.2 75.0 79.8 84.7 89.4 93.9 100
Random forest binary classification using H2O
Response: churn
Frame:                                           Number of observations:
  Training: train                                            Training =  5,643
                                                     Cross-validation =  5,643
Cross-validation: Stratify                       Number of folds       =      3
Model parameters
Number of trees       =  200
               actual =  200
Tree depth:                                      Pred. sampling value =     -1
           Input max =   20                      Sampling rate        =   .632
                 min =   16                      No. of bins cat.     =  1,024
                 avg = 19.6                      No. of bins root     =  1,024
                 max =   20                      No. of bins cont.    =     20
Min. obs. leaf split =    2                      Min. split thresh.   = .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.4153088	.416142
Mean class error	.2396365	.230295
AUC	.8507327	.8453018
AUCPR	.6526923	.6452846
Gini coefficient	.7014654	.6906036
MSE	.1335578	.1358418
RMSE	.3654556	.3685673

```
. h2omlest store rf_tuned
```

To choose the best method, we compute performance metrics using the testing frame. To compute AUCPR for the testing frame, we use the `h2omlpostestframe` command to specify the name of the frame, `test` in our case, to be used by a subset of postestimation commands for computations.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

By default, the specified frame is considered to be a testing frame and is labeled as "Testing" in the output, but you can specify your own label by using the `framelabel()` option. To report the metrics for the selected testing frame, we use the `h2omlestat metrics` command.

```
. h2omlestat metrics

Performance metrics using H2O
Random forest binary classification

Response:      churn
Testing frame: test

Number of observations = 1,400

─────────────────────────────────
             Metric │   Testing
────────────────────┼────────────
           Log loss │   .4101135
   Mean class error │   .2241742
                AUC │     .85292
              AUCPR │   .6847162
   Gini coefficient │     .70584
                MSE │   .1328891
               RMSE │   .3645396
─────────────────────────────────
```

We next compute the metrics for the testing frame for the GBM model after restoring its estimation results.

```
. h2omlest restore gbm_tuned
(results gbm_tuned are active now)

. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)

. h2omlestat metrics

Performance metrics using H2O
Gradient boosting binary classification

Response:      churn
Loss:          Bernoulli
Testing frame: test

Number of observations = 1,400

─────────────────────────────────
             Metric │   Testing
────────────────────┼────────────
           Log loss │   .3964014
   Mean class error │   .2030941
                AUC │   .8649185
              AUCPR │   .6963289
   Gini coefficient │   .7298371
                MSE │   .1284349
               RMSE │   .3583782
─────────────────────────────────
```

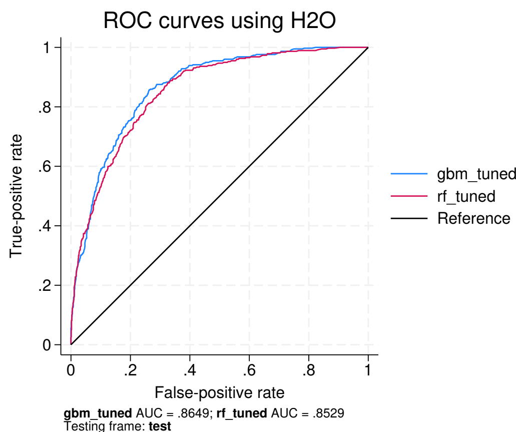
We can compare the results side by side more easily by using the h2omlgof command.

```
. h2omlgof rf_tuned gbm_tuned

Performance metrics for model comparison using H2O
Testing frame: test
```

	rf_tuned	gbm_tuned
Testing		
No. of observations	1,400	1,400
Log loss	.4101135	.3964014
Mean class error	.2241742	.2030941
AUC	.85292	.8649185
AUCPR	.6847162	.6963289
Gini coefficient	.70584	.7298371
MSE	.1328891	.1284349
RMSE	.3645396	.3583782

Based on this example, GBM outperforms random forest because AUCPR for GBM is higher. Thus, we choose GBM as our selected best method. We can also compare methods (or models) based on ROC curves, which plots the true-positive rate versus false-positive rate for different thresholds. The closer the curve to the upper left corner, the better the model fit. Because the test frame has been set for both models, the reported results correspond to the testing frame. For details, see [H2OML] **h2omlgraph roc**.

```
. h2omlgraph roc, models(gbm_tuned rf_tuned)
```



Based on the ROC results, as we expected, the GBM method slightly outperforms the random forest method.

Another popular approach to compare classification predictions between different methods and models is by using a confusion matrix, which reports the numbers of correctly and incorrectly predicted outcomes. Below, we use `h2omlestat confmatrix` to produce the confusion matrix after the GBM estimation for the testing frame we selected earlier with `h2omlpostestframe`.

```
. h2omlestat confmatrix

Confusion matrix using H2O
Testing frame: test

             |      Predicted
       churn |        No         Yes |   Total   Error     Rate
-------------+-----------------------+--------------------------
          No |       754         269 |   1,023     269     .263
         Yes |        54         323 |     377      54     .143
-------------+-----------------------+--------------------------
       Total |       808         592 |   1,400     323     .231
Note: Probability threshold .2378 that maximizes F1
      metric used for classification.
```

In H2O, the “positive” class corresponds to the second label in lexicographical order, which in our case is `Yes`. To see the levels of the categorical variable, type

```
. _h2oframe levelsof churn
`"No"' `"Yes"'
```

From the output, 323 and 754 correspond to true-positive and true-negative responses, respectively, and the misclassification error rate is 0.231. By default, the threshold for binary classification of 0.2378 is selected based on maximizing the F1 metric. Observations with predicted values above this threshold will be classified as “Yes”, and the remaining observations will be classified as “No”. You may want to see the results based on a different metric. For instance, consider a scenario where a company uses predictions to offer additional discounts or free services to customers who are likely to churn. If these benefits are costly, the company would prioritize predictions that maximize precision. To report the confusion matrix using a different metric, use the `metric()` option.

We encourage you to perform the same analysis for the `rf_tuned` model to verify that GBM indeed outperforms random forest on the testing frame.

◁

▷ Example 5: Classification prediction on new data

Continuing with example 4, suppose the company collected new data stored in `newchurn.dta`. It wants to predict the probability of churn for these new customers based on the GBM model `gbm_tuned`.

Let's read the new dataset as an H2O frame and list the first two observations to see some of the new data by using the `_h2oframe list` command.

```
. use https://www.stata-press.com/data/r19/newchurn
(Telco customer churn new data)
. _h2oframe put, into(newchurn) replace
Progress (%): 0 100
. _h2oframe change newchurn
. _h2oframe list in 1/2
  zipcode    latitude     longitude  tenure~s  monthlyc~s  totalcharges
1   95670  38.6027222  -121.2799149        49  75.1999969  3678.3000488
2   91737  34.2452888  -117.6425018         4  88.8499985   372.4500122

        country       state                city  gender  senior~n  partner
1 United States  California     Rancho Cordova    Male        No       No
2 United States  California  Rancho Cucamonga  Female       Yes       No

  depend~s  phones~e  multip~s  internets~v  online~y  online~p  device~t
1       No       Yes       Yes  Fiber optic        No        No        No
2       No       Yes       Yes  Fiber optic        No        No       Yes

  techsu~t  streamtv  stream~e        contract  paperl~l         paymethod
1       No        No        No  Month to month        No       Credit card
2       No        No       Yes  Month to month       Yes  Electronic check

[2 rows x 25 columns]
```

The probabilities of churning and the corresponding classes can be predicted by using the `h2omlpredict` command. By default, this command predicts classes after classification. To predict probabilities instead, we need to specify the `pr` option with `h2omlpredict`. In example 4, we used `h2omlpostestframe` to set the postestimation frame to test for the `gbm_tuned` model. To obtain predictions for the new dataset, specify the `frame(newchurn)` option with `h2omlpredict`. Below, we predict both classes and probabilities for the new dataset using the `gbm_tuned` model.

```
. h2omlest restore gbm_tuned
(results gbm_tuned are active now)
. h2omlpredict churnhat, frame(newchurn)
(option class assumed; predicted class)
Progress (%): 0 100
. h2omlpredict churnprob*, frame(newchurn) pr
Progress (%): 0 100
```

By default, the threshold that maximizes the F1 metric is used to predict classes based on the predicted probabilities. You can specify a different value for the threshold using the threshold() option. To display the threshold values that maximize or minimize different classification metrics, we type

```
. h2omlestat threshmetric

Maximum or minimum metrics using H2O
Testing frame: test
```

Metric	Max/Min	Threshold
F1	.6667	.2378
F2	.7816	.1496
F0.5	.6659	.5142
Accuracy	.8171	.5142
Precision	1	.9081
Recall	1	.0236
Specificity	1	.9081
Min. class accuracy	.7849	.2905
Mean class accuracy	.7969	.2378
True negatives	1023	.9081
False negatives	0	.0236 +
True positives	377	.0236
False positives	0	.9081 +
True-negative rate	1	.9081
False-negative rate	0	.0236 +
True-positive rate	1	.0236
False-positive rate	0	.9081 +
MCC	.5332	.2378

```
+ identifies minimum metrics.
```

The table above displays the set of classification metrics with the corresponding best thresholds; see [H2OML] **h2omlestat threshmetric**. In the reported table, the thresholds provide the best cutpoints for the classification based on the predicted probabilities such that the corresponding metric is optimal. For example, for Precision, the best threshold is 0.9081. For the definition of metrics, see [H2OML] ***metric_option***.

The generated variables for the classes and class probabilities are available in the newchurn frame, because we specified frame(newchurn). Let's list a few values for the predicted classes and probabilities.

```
. _h2oframe list churnhat churnprob*

  churnhat  churnp~1  churnp~2
1       No  .7780746  .2219254
2      Yes  .2161581  .7838419
3       No  .9001728  .0998272
4       No  .8937768  .1062232
5       No  .8101463  .1898537
6      Yes  .2203342  .7796658
7       No  .8987335  .1012665
8      Yes  .4977883  .5022117

[8 rows x 3 columns]
```

The variables (H2O columns) churnhat, churnprob1, and churnprob2 contain the predicted classes and the corresponding predicted probabilities of not churning or churning. In our example, for instance, there is only a 22% chance that the first customer will churn compared with a 78% chance of churning for the second customer.

▷ Example 6: Explaining classification prediction

In this example, we try to answer one of the fundamental questions of machine learning: Why does my model predict what it predicts? In machine learning, explainability refers to the ability of the method to describe how a model arrives at a specific prediction in a way that is understandable to humans. This is important to ensure that, under certain conditions, predictions are not only accurate but also understandable and justifiable.

From *Interpretation and explanation* in [H2OML] **Intro**, there are two types of explainability methods: local and global. Local models explain individual predictions and approximate the machine learning model in the vicinity of one observation. The popular methods include ICE curves and SHAP values, which can be obtained by using the h2omlgraph ice and h2omlgraph shapvalues commands. A global model describes an average behavior of a machine learning model. PDPs, variable importance, and global surrogate models are some of the popular choices.

We start with global methods and then switch to local methods. In example 4, we selected gbm_tuned as the best model. In this example, we want to explore predictions for the original churn dataset (without splitting it into training and testing frames). We start by restoring the gbm_tuned model:

```
. h2omlest restore gbm_tuned
(results gbm_tuned are active now)
```

Now we use h2omlpredict to predict classes for the entire churn dataset. We specify the frame() option to obtain predictions for the churn frame instead of the test frame we selected with h2omlpostestframe earlier in example 4.

```
. h2omlpredict churnhat, frame(churn)
(option class assumed; predicted class)
```

We use these predictions to build global surrogate models, which are some of the simplest global explainable methods. They approximate the prediction of a machine learning model, churnhat in our case, using a model that is easier to interpret such as a decision tree. See *Global surrogate models* in [H2OML] **Intro**.

To demonstrate, we use a classification tree with maximum depth equal to, say, 3 and other parameters at their default values as a global surrogate model. In practice, the depth of the tree and other parameters should be treated as hyperparameters and learned from data. To obtain one classification tree, we use the ntrees(1) option with h2oml rfbinclass.

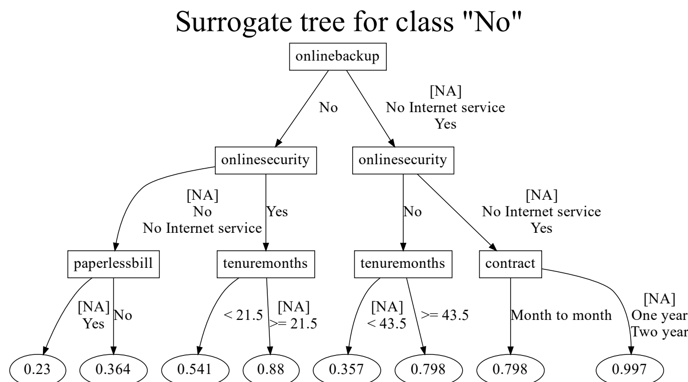
In example 1, we set our working frame as `train`. Thus, before running the estimation command `h2oml rfbinclass` on the `churn` dataset, we need to physically change the working frame to `churn` by using the `_h2oframe change` command.

```
. _h2oframe change churn
. h2oml rfbinclass churnhat $predictors, h2orseed(19) ntrees(1) maxdepth(3)
Progress (%): 0 100
Random forest binary classification using H2O
Response: churnhat
Frame:                                     Number of observations:
  Training: churn                                      Training =  2,523
Model parameters
Number of trees      =   1
              actual =   1
Tree depth:                                Pred. sampling value =     -1
           Input max =   3                 Sampling rate        =   .632
                 min =   3                 No. of bins cat.     =  1,024
                 avg = 3.0                 No. of bins root     =  1,024
                 max =   3                 No. of bins cont.    =     20
Min. obs. leaf split =   1                 Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Log loss	.4182261
Mean class error	.1828537
AUC	.8678704
AUCPR	.727738
Gini coefficient	.7357409
MSE	.1378874
RMSE	.3713319

It is easier to interpret the results from a classification tree visually. The steps on how to obtain an image from the DOT file are provided in [H2OML] **DOT extension**. We follow those steps to display the classification tree below; see [H2OML] **h2omltree**. The `dotsaving()` option of the `h2omltree` command generates and saves a DOT file, which can be used to plot the classification tree using the Graphviz software, see https://graphviz.org.

```
. h2omltree, dotsaving(churntree.dot, replace
> title(Surrogate tree for class "No"))
```

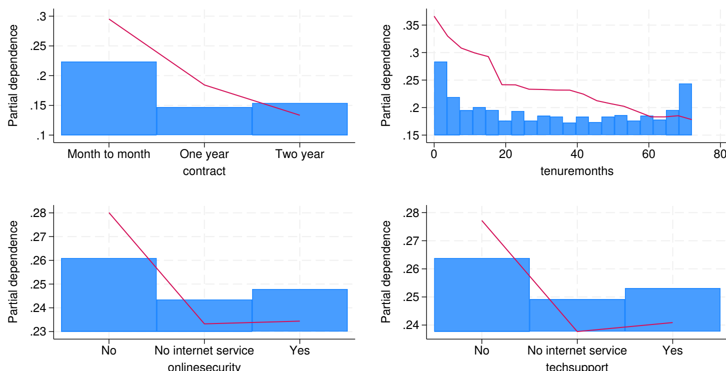


The NA's on the tree indicate the split for the missing values, if any. The values of the terminal nodes can be interpreted as probabilities of class `No`. For example, the highest-predicted probability of not churning (0.997) or the lowest probability of churning ($1 - 0.997 = 0.003$) occurs for the customers who have a one- or two-year contract with the company and are either not subscribed to any internet services or use online backup and online security services.

In example 3, we used `h2omlgraph varimp` to display important predictors for the `gbm_tuned` model. We use some of these important predictors to produce PDP. PDP is a global explainable method that shows the marginal effect that the specified predictors have on the predicted outcome of a machine learning model (`gbm_tuned` here); see [H2OML] **h2omlgraph pdp**.

Our current estimation results are from the `h2oml rfbinclass` command, so we first use `h2omlest restore` to restore the `gbm_tuned` estimation results. Next we use `h2omlpostestframe` with the `notest` option to specify that the `churn` frame be used by the subsequent postestimation commands but not considered a testing frame.

```
. h2omlest restore gbm_tuned
(results gbm_tuned are active now)
. h2omlpostestframe churn, notest
(frame churn is now active for h2oml postestimation)
. h2omlgraph pdp contract tenuremonths onlinesecurity techsupport, combine
Progress (%): 0 75.0 100
```



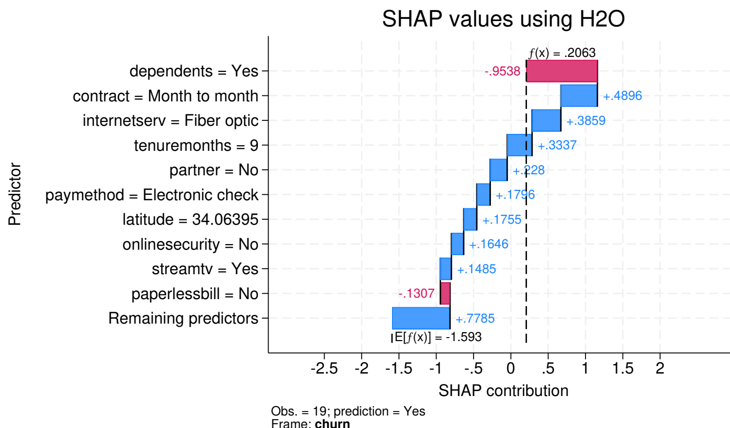
The PDP pattern (red line in the plot) agrees with the results from the surrogate tree. For instance, the probability of churning (shown on the y axis) decreases for customers with a one- or two-year contract (`contract`) and for customers who use the company's services longer (`tenuremonths`).

For local explainability, we can use SHAP values. A SHAP value estimates the contribution of each predictor to the prediction for an individual observation. Let's consider observation 19 and explain its prediction from the `gbm_tuned` model. Below, we list some of the predictors for this observation, which corresponds to a female customer who used a month-to-month contract service for 9 months and has both the observed `churn` and predicted `churnhat` values of `Yes`.

```
. _h2oframe list churn churnhat contract totalcharges onlinesecurity
> tenuremonths gender in 19
  churn  churnhat        contract  totalc~s  online~y  tenure~s  gender
1   Yes       Yes  Month to month    857.25        No         9  Female
[1 row x 7 columns]
```

We now use `h2omlgraph shapvalues` to produce SHAP values for observation 19 for the top 10 SHAP-important predictors.

```
. h2omlgraph shapvalues, obs(19) top(10) xlabel(-2.5(0.5)2)
```

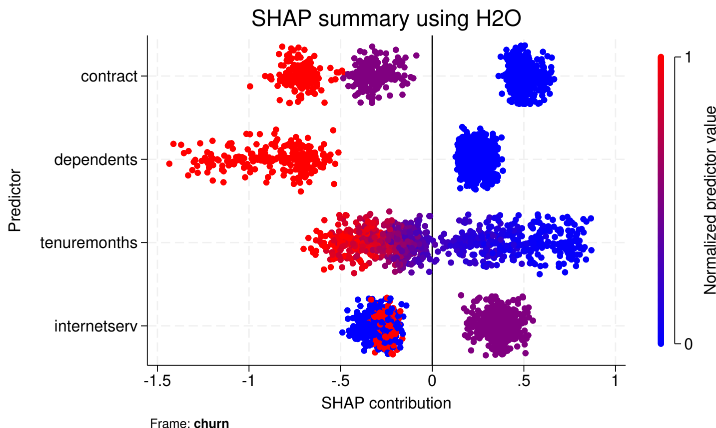


The blue bars show predictors that increase probability of churn, and red bars indicate the opposite. The SHAP values agree with previous findings. Month-to-month `contract`, small `tenuremonths`, and not using online security services contribute positively to this particular customers' churning. On the other hand, having a dependent contributes to retaining this particular customer to continue using the company's services.

We can also display the SHAP summary plot, also known as a beeswarm plot, for all observations and predictors. The beeswarm plot shows both the magnitudes of SHAP values, which represent the contribution of a predictor to a particular prediction, and the SHAP-value distribution across many observations. This allows you to quickly see which predictors are most important and how they influence the response.

For illustration purposes, we plot SHAP values for the top 4 SHAP-important predictors.

```
. h2omlgraph shapsummary, top(4) rseed(19)
```



In the figure, the color map, titled as "Normalized predictor value", indicates colors of the normalized values of the predictors. For example, if a variable is not of the data type enum, such as tenuremonths, then the smallest normalized variable value will be given a lighter blue color, and, as the values increase, the color gradient will change from blue to red for the largest value of 1. Similarly, for a categorical variable (enum), such as contract, the base level of the predictor will be given a lighter blue color, and the color will change from blue to red according to the categories. Within each level, the observations are jittered for presentational purposes. To check the levels of a categorical variable (for example, contract), type

```
. _h2oframe levelsof contract
`"Month to month"' `"One year"' `"Two year"'
```

The predictors displayed on the y axis are ranked based on SHAP predictor importance: predictors with large absolute SHAP values are listed in descending order. From the SHAP summary plot, for the contract predictor, a smaller value, which corresponds to the month-to-month option, increases the probability of churn, and this probability decreases for the other contract options. Similarly, smaller values of tenuremonths increase the probability of churn and vice versa.

◁

▷ Example 7: Shutting down the H2O cluster

Once you are finished with your analysis, you can disconnect from the H2O cluster by using

```
. h2o disconnect
```

This command closes the H2O session between Stata and the cluster. However, the H2O cluster continues running in the background. Later in the same Stata session, you can type h2o connect to rebuild the connection to it and reaccess the resources it contains. If you want to force shutting down the cluster, you can type

```
. h2o shutdown, force
```

The above completely shuts down the cluster, and all resources within the cluster are lost, including any data (H2O frames) it contained.

If you want the H2O cluster to remain connected but would like to clear everything in memory, including all data in H2O frames, you can type

```
. h2o clear
```

◁

Regression analysis

In this section, we demonstrate analysis for the regression problem using random forest.

▷ Example 8: Data setup

Consider the Ames housing dataset (De Cock 2011), ameshouses.dta, also used in a Kaggle competition, which describes residential houses sold in Ames, Iowa, between 2006 and 2010. It contains about 80 housing (and related) characteristics such as home size, amenities, and location. This dataset is often used for building predictive models for home sale price, saleprice. We will use random forest to model home sale price and evaluate its predictive performance. Here we will use just a few predictors to demonstrate some of the h2oml features.

Before putting the dataset into an H2O frame, we do several data transformations in Stata. In particular, because saleprice is right-skewed (type histogram saleprice), we perform logarithmic transformation. We also generate the houseage variable, which records the age of the house at the time of a sales transaction.

```
. use https://www.stata-press.com/data/r19/ameshouses
(Ames house data)
. generate logsaleprice = log(saleprice)
. generate houseage = yrsold - yearbuilt
. drop saleprice yearbuilt yrsold
```

We put the dataset into an H2O frame by using the _h2oframe put command. We split the data into training and validation frames (without a testing frame) with 75% of observations in the training frame.

```
. h2o init
  (output omitted)
. _h2oframe put, into(house)
Progress (%): 0 100
. _h2oframe change house
. _h2oframe split house, into(train valid) split(0.75 0.25) rseed(19)
. _h2oframe change train
```

The steps of method selection and prediction for the regression are the same as for binary classification, discussed in example 3 and example 4. Therefore, in this example, we focus only on tuning.

▷ Example 9: Regression using random forest

As we discussed in *Model selection in machine learning* of [H2OML] **Intro**, we start by defining a reference model, which in our case is a random forest with default parameters. We use the MSE metric, computed on validation frame, to evaluate the performance of the model.

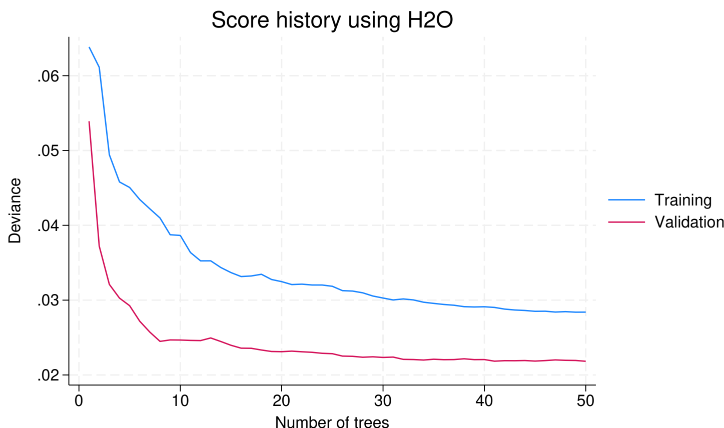
The dataset has a total of 46 predictors, but for simplicity, we include only 10 and create a global macro, `predictors`, in Stata to store the names of these predictors.

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

. h2oml rfregress logsaleprice $predictors, h2orseed(19) validframe(valid)

Progress (%): 0 54.0 100

Random forest regression using H2O

Response: logsaleprice
Frame:                                 Number of observations:
  Training:   train                               Training =  1,099
  Validation: valid                             Validation =    361

Model parameters

Number of trees      =   50
              actual =   50
Tree depth:                            Pred. sampling value =     -1
           Input max =   20            Sampling rate        =   .632
                 min =   18            No. of bins cat.     =  1,024
                 avg = 19.9            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

    Deviance      .0283991    .0218303
         MSE      .0283991    .0218303
        RMSE      .1685202    .1477508
       RMSLE      .0130751    .0114914
         MAE      .1163998    .1042066
   R-squared      .8240197    .8577693
```

The description and interpretation of the output of random forest is provided in example 1 of [H2OML] ***h2oml rf***. The definitions of metrics can be found in [H2OML] ***metric_option***.

The MSE for the validation frame is 0.022, which is our reference value for later. We also need to make sure that we are slightly overfitting the training dataset. The above model does not overfit the training dataset, because the training MSE is larger than the validation MSE. To visualize this, we plot the validation curve using the `h2omlgraph scorehistory` command.

```
. h2omlgraph scorehistory
Training frame:   train
Validation frame: valid
```



We observe that the training error is higher than the validation error. This means that either the default model is not complex enough to overfit the training dataset or we need more training data. In our case, the former reason is more likely, because we used a simpler model with default hyperparameters, which is sufficient for a reference model.

▷ Example 10: Hyperparameter tuning using random forest

In this example, we explore different configurations of the hyperparameters to tune the random forest model. In general, a well-tuned model substantially improves the model performance and generalizes well to new data.

To demonstrate, we tune only two hyperparameters, the number of trees, `ntrees()`, and the minimum number of observations required for splitting a leaf node, `minobsleaf()`, and use a small grid space with a random grid search. In practice, hyperparameter tuning is an iterative process and often requires tuning many more hyperparameters; see table 3 in [H2OML] **Intro**. When the number of hyperparameters and the grid space are large, you can use the `parallel()` option to specify the number of models to build in parallel during the grid search. Beware that the H2O results for models built in parallel may not always be reproducible; see [H2OML] **H2O reproducibility**. By default, the models are built sequentially, which may take some time for complicated tuning models.

```
. h2oml rfregress logsaleprice $predictors, h2orseed(19) validframe(valid)
> ntrees(400(50)500) minobsleaf(3(2)7)
> tune(grid(random, h2orseed(19)) metric(mse))
Progress (%): 0 100

Random forest regression using H2O

Response: logsaleprice
Frame:                                      Number of observations:
  Training:   train                                    Training =  1,099
  Validation: valid                                  Validation =    361

Tuning information for hyperparameters

Method: Random
Metric: MSE

─────────────────────────────────────────────────────────────────────
                      │                 Grid values
      Hyperparameters │     Minimum        Maximum         Selected
──────────────────────┼──────────────────────────────────────────────
      Number of trees │         400            500              450
 Min. obs. leaf split │           3              7                3
─────────────────────────────────────────────────────────────────────

Model parameters

Number of trees      =  450
             actual =  450
Tree depth:                                 Pred. sampling value =     -1
           Input max =   20                 Sampling rate        =   .632
                 min =   12                 No. of bins cat.     =  1,024
                 avg = 15.1                 No. of bins root     =  1,024
                 max =   20                 No. of bins cont.    =     20
Min. obs. leaf split =    3                 Min. split thresh.   = .00001

Metric summary

───────────────────────────────────
           │
    Metric │   Training  Validation
───────────┼───────────────────────
  Deviance │   .0269402    .0208756
       MSE │   .0269402    .0208756
      RMSE │   .1641346     .144484
     RMSLE │   .0127415    .0112297
       MAE │   .1113531    .0995714
 R-squared │     .83306    .8639893
───────────────────────────────────
```

To ensure H2O reproducibility, we specified `h2orseed(19)` for both the random forest model and grid search. Despite tuning only a couple hyperparameters, we were able to reduce the validation MSE metric from 0.022 to 0.021. To explore tuning further, you may try to include more hyperparameters and consider a larger grid space.

To compare different configurations of hyperparameters with their respective metric values sorted from the most to least optimal, we can use the h2omlestat gridsummary command.

```
. h2omlestat gridsummary
Grid summary using H2O
```

ID	Number of trees	Min. obs. leaf split	MSE
1	450	3	.0208756
2	500	3	.0209012
3	400	3	.020924
4	400	5	.021525
5	450	5	.0215336
6	500	5	.0215765
7	500	7	.0221419
8	400	7	.022142
9	450	7	.0221425

Here the hyperparameter values are listed from the smallest to largest MSE. If you want to reduce execution time in favor of a slightly lower model performance, you may select the third model instead of the first (top) model. For this model, the number of trees is 400 compared with 450 for the top model, but the MSE value is only slightly higher. We can select the third model for further analysis by typing

```
. h2omlselect id = 3
```

Effect of categorical predictors

As we discussed in *Decision trees* of [H2OML] **Intro**, the ensemble decision tree methods are biased toward categorical predictors with many levels. In this example, we explore the effect of a categorical predictor with many levels on performance of tree-based methods. Even though we focus on a GBM here, similar results should also hold for a random forest.

▷ Example 11: Data setup

We use a subset of the Lending Club dataset available in Kaggle to explore this phenomenon. Kaggle is a platform for the machine learning community that provides datasets and other resources; see https://kaggle.com.

We start by initializing an H2O cluster and importing the dataset as an H2O frame by using the h2o init and _h2oframe put commands.

```
. h2o init
. use https://www.stata-press.com/data/r19/loan
(Lending club data)
. _h2oframe put, into(loan)
Progress (%): 0 100
```

Next we use the _h2oframe split command to split the dataset into training and validation frames with 80% of observations in the training frame.

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

▷ Example 12: Effect of categorical predictors on ensemble decision tree methods

Consider the categorical predictor addr_state with 50 levels that records the state where the loan applicant lives. To show the importance of carefully treating categorical variables when performing ensemble decision tree methods, we first run a GBM without paying special attention to categorical predictors.

Let's define a global macro, predictors, to store the names of the predictors.

```
. global predictors loan_amnt int_rate emp_length annual_inc dti delinq_2yrs
> revol_util total_acc credit_lngth term home_owner purpose addr_state
> verification
```

Next we use h2oml gbbinclass to perform gradient boosting binary classification. We perform validation using the valid frame and specify the h2orseed() option for H2O reproducibility. We use 200 trees, and, to avoid overfitting, we request an early stopping based on the AUC metric. We also specify scoreevery(1) to score the AUC metric after each tree is added to the model to ensure H2O reproducibility in the presence of early stopping.

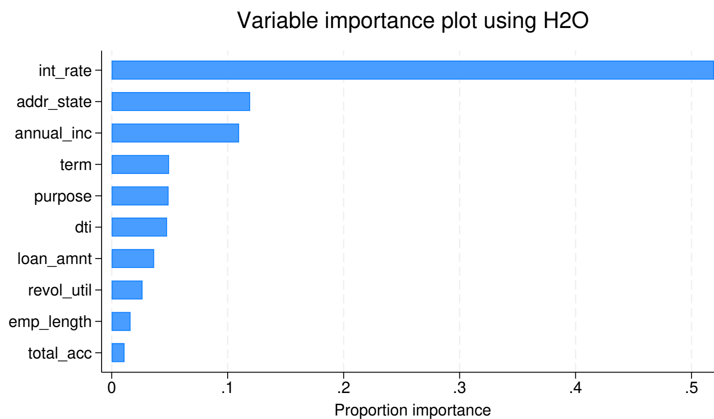
```
. h2oml gbbinclass bad_loan $predictors, h2orseed(19) validframe(valid)
> ntrees(200) stop(5, metric(auc)) scoreevery(1)
Progress (%): 0 1.4 5.0 10.9 18.0 100

Gradient boosting binary classification using H2O

Response: bad_loan
Loss:     Bernoulli
Frame:                                         Number of observations:
  Training:   train                                       Training = 131,294
  Validation: valid                                     Validation =  32,693

Model parameters

Number of trees      = 200               Learning rate        =       .1
              actual =  39               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

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

Metric summary
```

Metric	Training	Validation
Log loss	.4256225	.4381805
Mean class error	.3405512	.3471389
AUC	.7264524	.7081155
AUCPR	.3827862	.3495525
Gini coefficient	.4529049	.4162309
MSE	.1337261	.1384392
RMSE	.3656858	.3720742

Note: Metric is scored after every tree.

Let's plot the variable importance by using the `h2omlgraph varimp` command.

```
. h2omlgraph varimp
```



The variable `addr_state` is one of the important variables.

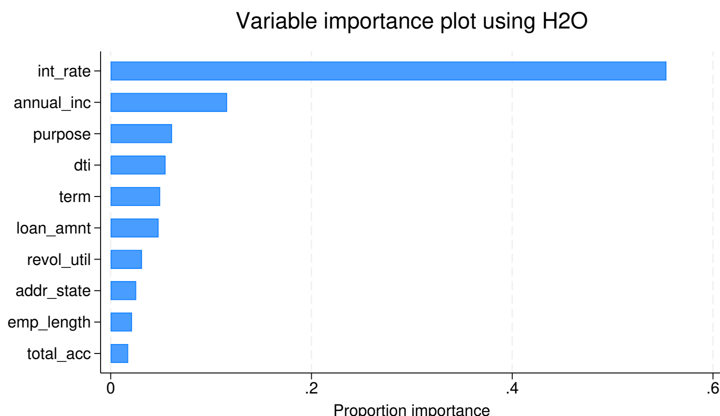
Now to account for the many categories in `addr_state`, we tune the hyperparameter `binscat()` on a grid of values [16, 50].

```
. h2oml gbbinclass bad_loan $predictors, h2orseed(19) validframe(valid)
> ntrees(200) binscat(16(5)50) stop(5, metric(auc)) scoreevery(1)
> tune(grid(cartesian) metric(auc))
Progress (%): 0 100

Gradient boosting binary classification using H2O

Response: bad_loan
Loss:     Bernoulli
Frame:                                   Number of observations:
  Training:   train                                Training = 131,294
  Validation: valid                              Validation =  32,693

Tuning information for hyperparameters

Method: Cartesian
Metric: AUC

--------------------------------------------------------------------
                  |                   Grid values
  Hyperparameters |         Minimum          Maximum        Selected
------------------+-------------------------------------------------
 No. of bins cat. |              16               46              46
--------------------------------------------------------------------

Model parameters

Number of trees      = 200             Learning rate       =       .1
             actual  =  46             Learning rate decay =        1
Tree depth:                            Pred. sampling rate =        1
           Input max =   5             Sampling rate       =        1
                 min =   5             No. of bins cat.    =       46
                 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

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

Metric summary

------------------------------------------
                  |
           Metric |  Training  Validation
------------------+-----------------------
         Log loss |  .4274797    .4368557
 Mean class error |  .3422759    .3435895
              AUC |  .7210886    .7100941
            AUCPR |  .3725785    .3557051
 Gini coefficient |  .4421772    .4201882
              MSE |  .1344013    .1379741
             RMSE |  .3666078    .3714487
------------------------------------------

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

Based on the tuning information, the value of 46 for `binscat()` provides the highest AUC value.

The variable importance graph for the selected best model, displayed below, shows that after accounting for the many levels of the categorical variable `addr_state`, its importance has decreased substantially.

```
. h2omlgraph varimp
```



Detecting nuisance predictors

▷ Example 13: Detecting nuisance predictors with ensemble decision tree methods

Let's use ensemble decision trees to detect important and nuisance predictors in the dataset. Here we use a random forest, but the results should be similar for a GBM as well. We use a simulated dataset, in which predictors `important1` through `important5` are important and `noise1` through `noise5` are nuisance (random noise). For the data-generation details, see Wright, Ziegler, and König (2016).

We start by initializing an H2O cluster and importing the dataset as an `h2oframe`.

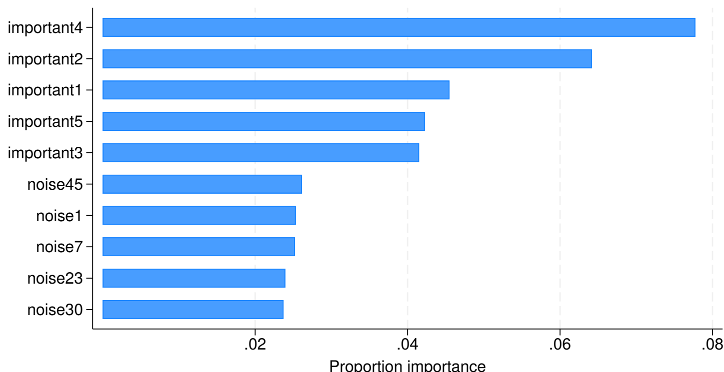
```
. use https://www.stata-press.com/data/r19/effect
(Simulated data with many nuisance predictors)
. h2o init
  (output omitted)
. _h2oframe put, into(sim)
Progress (%): 0 100
. _h2oframe change sim
```

Next we run a random forest binary classification with default hyperparameter values and plot the variable importance.

```
. h2oml rfbinclass y important1-important5 noise1-noise45, h2orseed(19)
Progress (%): 0 47.9 100
Random forest binary classification using H2O
Response: y
Frame:                                          Number of observations:
  Training: sim                                            Training =   1,000
Model parameters
Number of trees       =   50
             actual =   50
Tree depth:                                     Pred. sampling value =     -1
           Input max =   20                     Sampling rate        =   .632
                 min =   15                     No. of bins cat.     =  1,024
                 avg = 18.6                     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	.6693054
Mean class error	.3711672
AUC	.689691
AUCPR	.6739805
Gini coefficient	.3793821
MSE	.2227112
RMSE	.4719228

```
. h2omlgraph varimp
```



All important predictors are in the top five, but the separation between the important and nuisance predictors is not drastic. We can improve this by tuning the model.

We use a 3-fold modulo cross-validation and 500 trees. For illustration purposes, we train only hyperparameters that control the depth or complexity of the tree, `maxdepth()`, and the number of training samples used to build a tree, `samprate()`. We use the AUC metric for training.

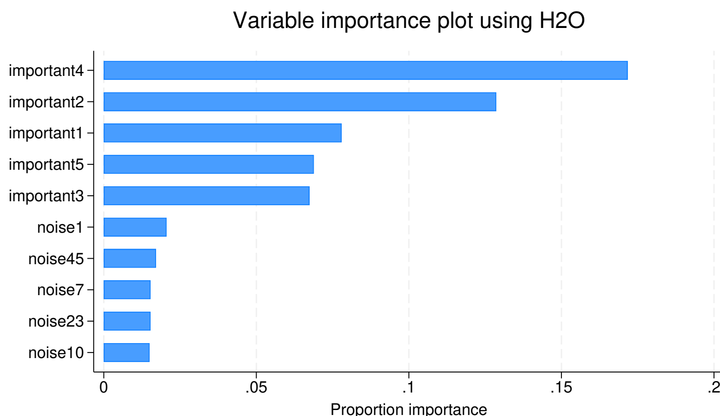
```
. h2oml rfbinclass y important1-important5 noise1-noise45, h2orseed(19)
> cv(3,modulo) ntrees(500) maxdepth(5(1)7) samprate(0.4(0.1)0.6)
> tune(metric(auc))

Progress (%): 0 100

Random forest binary classification using H2O

Response: y
Frame:                                 Number of observations:
  Training: sim                                   Training =  1,000
                                          Cross-validation =  1,000
Cross-validation: Modulo               Number of folds      =      3

Tuning information for hyperparameters

Method: Cartesian
Metric: AUC
```

Hyperparameters	Grid values Minimum	Maximum	Selected
Max. tree depth	5	7	6
Sampling rate	.4	.6	.5

```
Model parameters

Number of trees      = 500
             actual  = 500
Tree depth:                            Pred. sampling value =     -1
           Input max =   6             Sampling rate        =     .5
                 min =   6             No. of bins cat.     =  1,024
                 avg = 6.0             No. of bins root     =  1,024
                 max =   6             No. of bins cont.    =     20
Min. obs. leaf split =   1             Min. split thresh.   = .00001

Metric summary
```

Metric	Training	Cross-validation
Log loss	.6169953	.6233988
Mean class error	.3141157	.340729
AUC	.7528826	.7385296
AUCPR	.7392935	.7251183
Gini coefficient	.5057653	.4770591
MSE	.2130054	.2160959
RMSE	.4615251	.4648612

From the tuning output, the respective selected best values for `maxdepth()` and `samprate()` are 6 and 0.5. Let's plot the variable importance again.



Now there is a clearer separation between the important and nuisance predictors.

Gradient boosting Poisson regression

▷ Example 14: Explaining Poisson regression predictions

In example 7 of [H2OML] ***h2oml gbm***, we demonstrated how to perform a gradient boosting Poisson regression. In this example, we want to explain the Poisson regression predictions using that model. We repeat some of the steps from that example below and fit the final model.

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

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

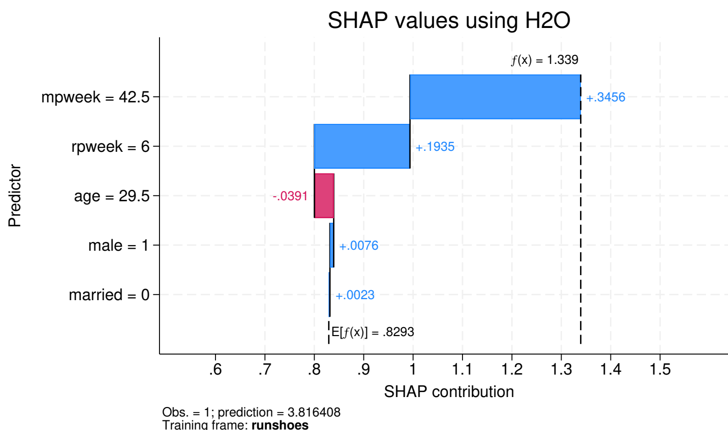
To perform a Poisson regression with `h2oml gbregress`, we specify the `loss(poisson)` option.

```
. 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

Next we explain the prediction for the first observation in the `runshoes` frame by using the `h2omlgraph shapvalues` command; see [H2OML] **h2omlgraph shapvalues**. You can follow the same steps to explain predictions for other observations.

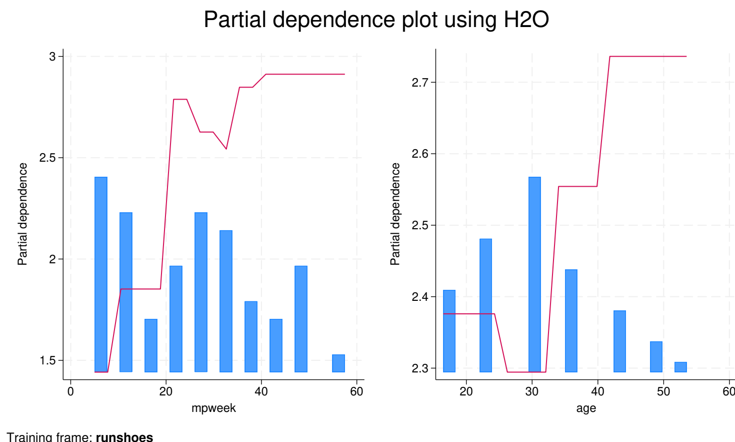
```
. h2omlgraph shapvalues, obs(1) xlabel(0.6(0.1)1.5)
```



The blue bars represent predictors that increase the probability of purchasing running shoes, whereas the red bars represent predictors that decrease it. For this observation, running 42.5 miles per week has a positive effect on the number of shoes purchased, whereas an age of 29.5 has a negative effect.

We continue our analysis and produce a PDP for the predictors mpweek and age by using the h2omlgraph pdp command.

```
. h2omlgraph pdp mpweek age, combineopts(cols(2))
```



The PDP (red line) supports the previous result. Specifically, in the graph for age on the right, we observe a noticeable decrease in PDP roughly between ages 25 and 30, which implies a negative effect of age on buying running shoes. But after age 30, the effect is positive.

◁

References

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.

De Cock, D. 2011. Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education* 19(3). https://doi.org/10.1080/10691898.2011.11889627.

Raschka, S. 2020. Model evaluation, model selection, and algorithm selection in machine learning. arXiv:1811.12808 [cs.LG], https://doi.org/10.48550/arXiv.1811.12808.

Valdenegro-Toro, M., and M. Sabatelli. 2023. "Machine learning students overfit to overfitting". In *Proceedings of the Third Teaching Machine Learning and Artificial Intelligence Workshop*, edited by K. M. Kinnaird, P. Steinbach, and O. Guhr, vol. 207: 46–51. Clearwater Beach, FL: Proceedings of Machine Learning Research.

Wright, M. N., A. Ziegler, and I. R. König. 2016. Do little interactions get lost in dark random forests? *BMC Bioinformatics* 17: art. 145. https://doi.org/10.1186/s12859-016-0995-8.

Also see

[H2OML] **Intro** — Introduction to machine learning and ensemble decision trees

[H2OML] **Glossary**

H2O setup — Prepare data for H2O analysis in Stata

Description   Remarks and examples   Also see

Description

In this entry, we provide an introduction to the H2O integration with Stata. We introduce commands for initiating H2O and working with data frames in H2O, both of which are necessary before you can use `h2oml` commands described in [H2OML] **h2oml** and throughout this manual.

Remarks and examples

Remarks are presented under the following headings:

What is H2O?
How does H2O work from Stata?
  Start a local H2O cluster
  Connect to an existing H2O cluster
Interact with the H2O cluster
Close and disconnect the H2O cluster

What is H2O?

H2O is a scalable and distributed machine learning and predictive platform. It is an open-source platform, and its core code is written in Java. Stata uses H2O's REST API to connect to H2O. You can perform in-memory data analysis and machine learning using this framework. More information about the H2O framework can be found on the H2O website at https://docs.h2o.ai/. We also refer you to H2O's *User Guide*.

We separate H2O related commands in Stata into two categories:

1. Commands to establish connection with H2O and work with H2O frames. For details, see [P] **H2O intro** and https://www.stata.com/h2o/.

2. Commands for machine learning (`h2oml`). For the Stata examples, see [H2OML] **h2oml**.

How does H2O work from Stata?

You can either start a new H2O cluster or connect to an existing H2O cluster from within Stata. Then you use the suite of Stata commands (`h2o`, `_h2oframe`, and `h2oml`) to interact with the H2O cluster.

Start a local H2O cluster

You can start a local H2O cluster by typing in Stata

```
. h2o init
```

`h2o init` will look for the existence of an `h2o.jar` file, a Java Archive (JAR) file that is used to start H2O. This file is distributed by H2O. Stata does not distribute `h2o.jar` with its installation.

Downloading and placing an h2o.jar

To download the h2o.jar file and place it in the local directory so that Stata can locate it, you can follow the steps below. Note that these steps need to be completed only once.

You can obtain the h2o.jar file from H2O's download page.

1. Go to https://h2o.ai/resources/download/.
2. Click on the tab **H2O Open Source Platform**.
3. Go to **Latest Stable Release** or **Prior Releases**. Stata's H2OML documentation is written using **Version 3.46.0.6**.
4. Click on **Download H2O**.
5. After downloading the file (for example, h2o-3.46.0.6.zip), unzip it and look for the h2o.jar file. This is the only file from within the zip file that you will need.

After downloading the h2o.jar file, place the file in a directory included in Stata's system directories (ado-path). To view directories on the ado-path, you can use the adopath command. For details, see [P] **sysdir**. For example, the following is a typical Stata output on a Windows computer:

```
. adopath
  [1]  (BASE)      "C:\Program Files\Stata19\ado\base"
  [2]  (SITE)      "C:\Program Files\Stata19\ado\site"
  [3]              "."
  [4]  (PERSONAL)  "C:\ado\personal"
  [5]  (PLUS)      "C:\ado\plus"
  [6]  (OLDPLACE)  "C:\ado"
```

We recommend using the SITE, PERSONAL, or PLUS directory. When h2o.jar is placed along the ado-path, h2o init will use it directly to start a new local H2O cluster. If multiple copies of h2o.jar exist along the ado-path, Stata will prioritize based on the order that the adopath command presents and will use the first h2o.jar it locates. Because we are looking for a .jar file, h2o init can locate h2o.jar if it is placed in a jar/ subdirectory. Please create the jar/ subdirectory if it does not exist in any of the defined ado-path locations. If h2o.jar cannot be located, h2o init will produce an error.

After h2o.jar is located, h2o init will determine whether a cluster is already running on your local machine.

When the cluster has been successfully initialized, Stata will automatically connect to this cluster, and a summary of the H2O cluster status similar to the following will be displayed:

```
. h2o init
Connecting to the H2O cluster running at http://127.0.0.1:54321.....not found.
Starting a new cluster running at http://127.0.0.1:54321.
Connecting to the H2O cluster running at http://127.0.0.1:54321... Successful.
--------------------------------------------------------------------------------
H2O cluster uptime:         1 sec
H2O cluster timezone:       America/Chicago
H2O data parsing timezone: UTC
H2O cluster version:        3.46.0.6
H2O cluster version age:    4 months and 29 days
H2O cluster total nodes:    1
H2O cluster free memory:    15.67 Gb
H2O cluster total cores:    32
H2O cluster allowed cores: 32
H2O cluster status:         accepting new members, healthy
H2O connection url:         http://127.0.0.1:54321
--------------------------------------------------------------------------------
```

`h2o init` allows some options for customizing the initialization of the H2O cluster. For example, we can specify the `nthreads()` option to set the maximum number of parallel threads to use when launching the H2O cluster. For details, see https://www.stata.com/h2o/h2o18/h2o.html.

❑ Technical note

`h2o init` uses the address of **localhost:54321**, where the IP of localhost is **127.0.0.1** and the port is **54321**. If a cluster is not already running, `h2o init` will attempt to create one at this location, and by default, the new cluster will allow connections only from the local machine.

❑

Connect to an existing H2O cluster

Another way to interact with H2O is to connect to an existing H2O cluster by using the `h2o connect` command. For example, an existing H2O cluster can be a cluster previously started by `h2o init`. For details, see https://www.stata.com/h2o/h2o18/h2o.html.

To connect to an existing H2O cluster, we can type `h2o connect` in Stata. If the connection is built successfully, Stata will report a summary of the cluster status similar to the following:

```
. h2o connect
Connecting to the H2O cluster running at http://localhost:54321. Successful.
--------------------------------------------------------------------------------
H2O cluster uptime:         1 sec
H2O cluster timezone:       America/Chicago
H2O data parsing timezone: UTC
H2O cluster version:        3.46.0.6
H2O cluster version age:    4 months and 29 days
H2O cluster total nodes:    1
H2O cluster free memory:    15.67 Gb
H2O cluster total cores:    32
H2O cluster allowed cores: 32
H2O cluster status:         locked, healthy
H2O connection url:         http://localhost:54321
--------------------------------------------------------------------------------
```

You can also connect to an H2O cluster running on a remote machine by specifying its IP and port in the `ip()` and `port()` options in the `h2o connect` command. For details, see *Options for h2o connect*.

❑ Technical note

By default, `h2o connect` will attempt to connect to a cluster running at **localhost:54321** on your local machine; if you started a local cluster with `h2o init`, then credentials will automatically be used.

❑

When you connect to an existing H2O cluster, a new Stata H2O session is created between Stata (the client) and the H2O cluster. Multiple clients can be connecting to the H2O cluster at the same time, and they will all share its resources, such as the data and models within the cluster.

Interact with the H2O cluster

Once a connection with an H2O cluster has been established, you can interact with it directly from within Stata.

For example, you can import data from the local drive to the cluster as an H2O frame or put data currently in Stata into an H2O frame. The following code will load the `iris` dataset to the cluster into an H2O frame `h2oiris`. For details, see https://www.stata.com/h2o/h2o18/.

```
. use https://www.stata-press.com/data/r19/iris
(Iris data)
. _h2oframe put, into(h2oiris)
```

To load a subset of the data, you can specify *varlist* and the `if` and `in` qualifiers. For more details, see https://www.stata.com/h2o/h2o18/h2oframe_put.html.

You can type `_h2oframe dir` to list all H2O frames in the cluster, along with the dimensions of the data and the amount of memory the data consume in the cluster.

```
. _h2oframe dir
```

Name	Rows	Cols	Size
h2oiris	150	5	1.773 Kb

```
Total: 1
```

For more information about H2O frames, see https://www.stata.com/h2o/h2o18/h2oframe.html.

You can set or change to the `h2oiris` frame as the current working H2O frame by using the `_h2oframe change` command. Then to perform, for instance, gradient boosting multiclass classification using the dataset on this frame, type

```
. _h2oframe change h2oiris
. h2oml gbmulticlass iris seplen sepwid petlen petwid
  (output omitted)
```

Instead of separate `_h2oframe put` and `_h2oframe change` commands, it is often convenient to put data into an H2O frame and make that frame current in a single step by typing, for instance,

```
_h2oframe put, into(h2oiris) current
```

Close and disconnect the H2O cluster

Once you have finished the analysis on the H2O cluster, you can type

```
. h2o disconnect
```

to close the connection from the H2O session between Stata and the cluster or

```
. h2o shutdown
```

to shut down the cluster.

The `h2o disconnect` command will close the H2O connection between Stata and the cluster, leaving the H2O cluster running. Later in the same Stata session, you can type `h2o connect` to rebuild the connection to it and reaccess the resources it contains.

The `h2o shutdown` command will destroy the cluster you are currently connected to along with all its resources. By default, `h2o shutdown` will exit with an error and give a warning about its destructive nature. To override this warning and actually shut down the cluster, use the `force` option. This will force the cluster to shut down, and everything in the cluster will be destroyed regardless of whether the cluster was created from Stata or outside of Stata.

Note that if the cluster was created by Stata using the `h2o init` command, then by exiting a Stata session, it will be automatically shut down. We recommend to ensure that all the necessary resources within the cluster are saved before exiting. To prevent a cluster that Stata created from automatically getting shut down, use `h2o disconnect` before closing Stata. If the cluster was created outside of Stata and a connection was made using `h2o connect`, then exiting Stata will close only the connection, leaving all resources within the cluster intact.

The table below summarizes the alternatives to close or disconnect an H2O frame.

Option	Cluster created by Stata	Cluster created outside of Stata
`h2o disconnect`	close H2O session without loss of information	close H2O session without loss of information
`h2o shutdown, force`	close H2O session and discard information in the cluster	close H2O session and discard information in the cluster
Exit Stata session	same as `h2o shutdown, force`	same as `h2o disconnect`

In practice, if you are certain that all necessary results have been saved, it is preferable to use `h2o shutdown` to shut down the H2O cluster. Putting all H2O-related commands between `h2o init` and `h2o shutdown, force` is the recommended practice.

Also see

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

[P] **H2O intro** — Introduction to integration with H2O

h2oml gbm — Gradient boosting machine for regression and classification

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
```

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

Same 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)
```

Same 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))
```

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 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))
```

Same 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))
```

Same 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.

Remarks and examples

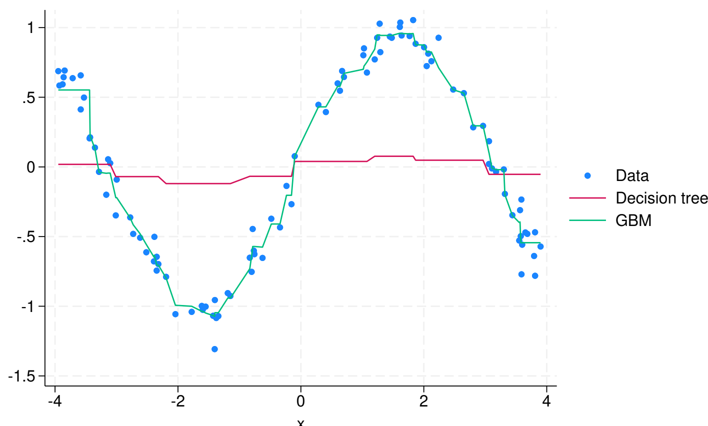
We assume you have read the introduction to decision trees and ensemble methods in [H2OML] **Intro**.

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/r19/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 95.6 100

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/r19/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/r19/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 23.8 68.1 100

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 60.3 98.6 100
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/r19/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/r19/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
```

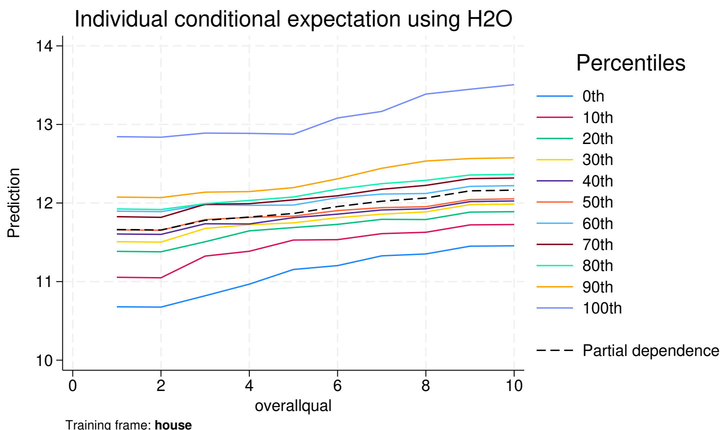
To perform GBM quantile regression with `h2oml gbmregress`, we specify the `loss(quantile)` option with the `alpha(0.25)` suboption for the desired quantile. We also prespecify values for some hyperparameters.

```
. h2oml gbregress logsaleprice $predictors, loss(quantile, alpha(0.25))
> h2orseed(19) ntrees(500) minobsleaf(1) binscat(115) samprate(0.8)
Progress (%): 0 6.5 28.0 57.2 82.9 100
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/r19/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 14.1 100
. _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 8.3 15.6 21.9 28.2 35.6 43.0 50.4 58.3 67.5 77.7
> 87.9 96.7 100

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 2.1 3.7 5.4 8.3 11.9 15.8 19.5 22.8 26.8 30.7 34.5 38.4 42.3
>  46.3 50.0 53.7 57.8 61.7 65.3 69.1 73.1 77.3 80.8 84.7 88.8 92.7 96.2 100
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

h2oml gbbinclass — Gradient boosting binary classification

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

Description

h2oml gbbinclass implements gradient boosting classification for binary responses. You can validate your model by using validation data or cross-validation, and you can tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the gradient boosting machine (GBM) method, see [H2OML] **Intro**.

Quick start

Before running the h2oml gbbinclass command, 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 binary classification of binary response y1 on predictors x1 through x100

```
h2oml gbbinclass y1 x1-x100
```

Same as above, but also report measures of fit for the validation frame named valid, and set an H2O random-number seed for reproducibility

```
h2oml gbbinclass y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation

```
h2oml gbbinclass y1 x1-x100, cv(3) h2orseed(123)
```

Same 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 gbbinclass y1 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) lrate(0.01) predsamprate(0.6)
```

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 gbbinclass y1 x1-x100, cv(3) h2orseed(123) lrate(0.01)     ///
    predsamprate(0.6) 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 gbbinclass y1 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))
```

Same 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 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(logloss) grid(random, h2orseed(456)) maxtime(200))
```

Menu

Statistics > H2O machine learning

Syntax

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

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

options	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 the minority class
h2orseed(#)	set H2O random-number seed for GBM
encode(*encode_type*)	specify H2O encoding type for categorical predictors; default is encode(enum)
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
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 evaluation 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 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

validframe(), cv[()], balanceclasses, h2orseed(), encode(), stop[()], maxtime(), scoreevery(), and monotone(); see [H2OML] ***h2oml gbm***.

Hyperparameter

ntrees(), lrate(), lratedecay(), maxdepth(), minobsleaf(), predsamprate(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml gbm***.

Tuning

tune(); see [H2OML] ***h2oml gbm***.

Remarks and examples

For examples, see Remarks and examples in [H2OML] ***h2oml gbm***.

Stored results

h2oml gbbinclass 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_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(maxtime)	maximum run time
e(balanceclass)	1 if classes are balanced; 0 otherwise
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 gbbinclass
e(cmdline)	command as typed
e(subcmd)	gbbinclass
e(method)	gbm
e(method_type)	classification
e(class_type)	binary
e(method_full_name)	Gradient boosting binary classification
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 (with option validframe())
e(valid_frame)	name of the validation frame (with option cv())
e(cv_method)	fold assignment method (with option cv())
e(cv_varname)	name of variable identifying cross-validation folds
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

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 gbm*** — Gradient boosting machine for regression and classification

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

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

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

[U] **20 Estimation and postestimation commands**

h2oml gbmulticlass — Gradient boosting multiclass classification

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

Description

h2oml gbmulticlass implements gradient boosting multiclass classification for categorical responses. You can validate your model by using validation data or cross-validation, and you can tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the gradient boosting machine (GBM) method, see [H2OML] **Intro**.

Quick start

Before running the h2oml gbmulticlass command, 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 multiclass classification of categorical response y1 on predictors x1 through x100

```
h2oml gbmulticlass y1 x1-x100
```

Same as above, but also report measures of fit for the validation frame named valid, and set an H2O random-number seed for reproducibility

```
h2oml gbmulticlass y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation

```
h2oml gbmulticlass y1 x1-x100, cv(3) h2orseed(123)
```

Same 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 y1 x1-x100, cv(3) h2orseed(123) ntrees(30)     ///
    maxdepth(10) lrate(0.01) predsamprate(0.6)
```

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 gbmulticlass y1 x1-x100, cv(3) h2orseed(123) lrate(0.01)     ///
    predsamprate(0.6) 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 gbmulticlass y1 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))
```

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

```
h2oml gbmulticlass 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(logloss) grid(random, h2orseed(456)) maxtime(200))
```

Menu

Statistics > H2O machine learning

Syntax

h2oml gbmulticlass *response_mult predictors* [, *options*]

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

options	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
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 and area under the precision–recall curve metrics
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
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 evaluation 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 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

validframe(), cv[()], balanceclasses, h2orseed(), encode(), auc, stop[()], maxtime(), and scoreevery(); see [H2OML] ***h2oml gbm***.

Hyperparameter

ntrees(), lrate(), lratedecay(), maxdepth(), minobsleaf(), predsamprate(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml gbm***.

Tuning

tune(); see [H2OML] ***h2oml gbm***.

Remarks and examples

For examples, see *Remarks and examples* in [H2OML] ***h2oml gbm***.

Stored results

h2oml gbmulticlass 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
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(auc)	1 if auc; 0 otherwise
e(maxtime)	maximum run time
e(balanceclass)	1 if classes are balanced; 0 otherwise
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 gbmulticlass
e(cmdline)	command as typed
e(subcmd)	gbmulticlass
e(method)	gbm
e(method_type)	classification
e(class_type)	multiclass
e(method_full_name)	Gradient boosting multiclass classification
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(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

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 gbm*** — Gradient boosting machine for regression and classification

[H2OML] **h2oml gbbinclass** — Gradient boosting binary classification

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

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

[U] **20 Estimation and postestimation commands**

Description	Quick start	Menu	Syntax
Options	Remarks and examples	Stored results	Also see

Description

h2oml gbregress implements gradient boosting regression for continuous and count responses. You can choose from six loss functions, validate your model by using validation data or cross-validation, and tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the gradient boosting machine (GBM) method, see [H2OML] **Intro**.

Quick start

Before running the h2oml gbregress command, 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
```

Same as above, but also report measures of fit for the validation frame named valid, and set an H2O random-number seed for reproducibility

```
h2oml gbregress y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation

```
h2oml gbregress y1 x1-x100, cv(3) h2orseed(123)
```

Same 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 gbregress y1 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) lrate(0.01) predsamprate(0.6)
```

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 mean squared error (MSE) metric

```
h2oml gbregress y1 x1-x100, cv(3) h2orseed(123) lrate(0.01)      ///
    predsamprate(0.6) ntrees(10(5)100) maxdepth(3(1)10)      ///
    tune(metric(mse))
```

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 gbregress y1 x1-x100, cv(3) h2orseed(123) lrate(0.01)      ///
    predsamprate(0.6) ntrees(10(5)100) maxdepth(3(1)10)      ///
    tune(metric(mse) grid(random, h2orseed(456)) maxtime(200))
```

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

```
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))
```

Run gradient boosting quantile regression by specifying the quantile loss function

```
h2oml gbregress y1 x1-x100, loss(quantile)
```

Menu

Statistics > H2O machine learning

Syntax

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

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

options	Description
Model	
`loss(`*losstype*`)`	specify the loss function; 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
`h2oseed(`#`)`	set H2O random-number seed for GBM
`encode(`*encode_type*`)`	specify H2O encoding type for categorical predictors; default is `encode(enum)`
`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
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 evaluation metrics are reported for the training dataset.

monotone() can be specified only with loss(gaussian), loss(tweedie), or loss(quantile).

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 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 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(), validframe(), cv[()], h2orseed(), encode(), stop[()], maxtime(), scoreevery(), and monotone(); see [H2OML] ***h2oml gbm***.

Hyperparameter

ntrees(), lrate(), lratedecay(), maxdepth(), minobsleaf(), predsamprate(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml gbm***.

Tuning

tune(); see [H2OML] ***h2oml gbm***.

Remarks and examples

For examples, see *Remarks and examples* in [H2OML] ***h2oml gbm***.

Stored results

h2oml gbregress 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_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(maxtime)	maximum run time
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
e(cmdline)	command as typed
e(subcmd)	gbregress
e(method)	gbm
e(method_type)	regression
e(method_full_name)	Gradient boosting regression
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

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 gbm*** — Gradient boosting machine for regression and classification

[H2OML] **h2oml gbbinclass** — Gradient boosting binary classification

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

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

[U] 20 Estimation and postestimation commands

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

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
Tuning hyperparameters
Examples of using random forest
Example 1: Random forest binary classification using default settings
Example 2: Using validation data and early stopping
Example 3: Using cross-validation
Example 4: User-specified hyperparameters
Example 5: Multiclass classification and model performance

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.

Example 1: Random forest binary classification using default settings
Example 2: Using validation data and early stopping
Example 3: Using cross-validation
Example 4: User-specified hyperparameters
Example 5: Multiclass classification and model performance

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

h2oml rfbinclass — Random forest binary classification

Description Quick start Menu Syntax
Options Remarks and examples Stored results Also see

Description

h2oml rfbinclass implements random forest classification for binary responses. You can validate your model by using validation data or cross-validation, and you can tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the random forest method, see [H2OML] **Intro**.

Quick start

Before running the h2oml rfbinclass command, 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 binary classification of binary response y1 on predictors x1 through x100

```
h2oml rfbinclass y1 x1-x100
```

Same as above, but also report measures of fit for the validation frame named valid, and set an H2O random-number seed for reproducibility

```
h2oml rfbinclass y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation

```
h2oml rfbinclass y1 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 15

```
h2oml rfbinclass y1 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) predsampvalue(15)
```

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

```
h2oml rfbinclass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
    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, and limit the maximum search time to 200 seconds

```
h2oml rfbinclass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
    ntrees(10(5)100) maxdepth(3(1)10)                           ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200))
```

Same as above, but use early stopping with the default stopping log-loss metric and 5 iterations of tuning

```
h2oml rfbinclass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
   ntrees(10(5)100) maxdepth(3(1)10)                  ///
   tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200) ///
   stop(5))
```

Same as above, but tune the number of bins for the categorical and continuous predictors

```
h2oml rfbinclass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
   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) stop(5))
```

Menu

Statistics > H2O machine learning

Syntax

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

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

options	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 the minority class
h2orseed(#)	set H2O random-number seed for random forest
encode(*encode_type*)	specify H2O encoding type for categorical predictors; default is encode(enum)
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 evaluation 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(), cv[()], balanceclasses, h2orseed(), encode(), stop[()], maxtime(), and scoreevery(); see [H2OML] ***h2oml rf***.

Hyperparameter

ntrees(), maxdepth(), minobsleaf(), predsampvalue(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml rf***.

Tuning

tune(); see [H2OML] *h2oml rf*.

Remarks and examples

For examples, see *Remarks and examples* in [H2OML] *h2oml rf*.

Stored results

h2oml rfbinclass 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_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(binsroot)	number of bins for root node
e(h2orseed)	H2O random-number seed
e(maxtime)	maximum run time
e(balanceclass)	1 if classes are balanced; 0 otherwise
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 rfbinclass
e(cmdline)	command as typed
e(subcmd)	rfbinclass
e(method)	randomforest
e(method_type)	classification
e(class_type)	binary
e(method_full_name)	Random forest binary classification
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

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 rf*** — Random forest for regression and classification

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

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

[H2OML] **h2oml gbbinclass** — Gradient boosting binary classification

[U] 20 Estimation and postestimation commands

h2oml rfmulticlass — Random forest multiclass classification

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

Description

h2oml rfmulticlass implements random forest multiclass classification for categorical responses. You can validate your model by using validation data or cross-validation, and you can tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the random forest method, see [H2OML] **Intro**.

Quick start

Before running the h2oml rfmulticlass command, 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 multiclass classification of categorical response y1 on predictors x1 through x100

```
h2oml rfmulticlass y1 x1-x100
```

Same as above, but also report measures of fit for the validation frame named valid, and set an H2O random-number seed for reproducibility

```
h2oml rfmulticlass y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation to report measures of fit

```
h2oml rfmulticlass y1 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 15

```
h2oml rfmulticlass y1 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) predsampvalue(15)
```

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 y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15) ///
    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, and limit the maximum search time to 200 seconds

```
h2oml rfmulticlass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15) ///
    ntrees(10(5)100) maxdepth(3(1)10)                     ///
    tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200))
```

Same as above, but use early stopping with the default stopping log-loss metric and 5 iterations of tuning

```
h2oml rfmulticlass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15) ///
   ntrees(10(5)100) maxdepth(3(1)10)                        ///
   tune(metric(logloss) grid(random, h2orseed(456)) maxtime(200)  ///
   stop(5))
```

Same as above, but tune the number of bins for the categorical and continuous predictors

```
h2oml rfmulticlass y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15) ///
   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) stop(5))
```

Menu

Statistics > H2O machine learning

Syntax

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

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

options	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
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 and area under the precision–recall curve metrics
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 evaluation 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(), cv[()], balanceclasses, h2orseed(), encode(), auc, stop[()], maxtime(), and scoreevery(); see [H2OML] ***h2oml rf***.

Hyperparameter

ntrees(), maxdepth(), minobsleaf(), predsampvalue(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml rf***.

Tuning

tune(); see [H2OML] *h2oml rf*.

Remarks and examples

For examples, see *Remarks and examples* in [H2OML] *h2oml rf*.

Stored results

h2oml rfmulticlass 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
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(maxtime)	maximum run time
e(balanceclass)	1 if classes are balanced; 0 otherwise
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 rfmulticlass
e(cmdline)	command as typed
e(subcmd)	rfmulticlass
e(method)	randomforest
e(method_type)	classification
e(class_type)	multiclass
e(method_full_name)	Random forest multiclass classification
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

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 rf*** — Random forest for regression and classification

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

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

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

[U] 20 Estimation and postestimation commands

Description Quick start Menu Syntax
Options Remarks and examples Stored results Also see

Description

`h2oml rfregress` implements random forest regression for continuous responses. You can validate your model by using validation data or cross-validation, and you can tune hyperparameters and stop early to improve model performance on new data. This command provides 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 the random forest method, see [H2OML] **Intro**.

Quick start

Before running the `h2oml rfregress` command, 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 also report measures of fit for the validation frame named `valid`, and set an H2O random-number seed for reproducibility

```
h2oml rfregress y1 x1-x100, validframe(valid) h2orseed(123)
```

Same as above, but instead of a validation frame, use 3-fold cross-validation

```
h2oml rfregress y1 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 15

```
h2oml rfregress y1 x1-x100, cv(3) h2orseed(123) ntrees(30)      ///
    maxdepth(10) predsampvalue(15)
```

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 mean squared error (MSE) metric

```
h2oml rfregress y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)   ///
    ntrees(10(5)100) maxdepth(3(1)10) tune(metric(mse))
```

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

```
h2oml rfregress y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)   ///
    ntrees(10(5)100) maxdepth(3(1)10)                    ///
    tune(metric(mse) grid(random, h2orseed(456)) maxtime(200))
```

Same as above, but use early stopping with the MSE metric and 5 iterations of tuning

```
h2oml rfregress y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
   ntrees(10(5)100) maxdepth(3(1)10)                    ///
   tune(metric(mse) grid(random, h2orseed(456)) maxtime(200)   ///
   stop(5, metric(mse)))
```

Same as above, but tune the number of bins for the categorical and continuous predictors

```
h2oml rfregress y1 x1-x100, cv(3) h2orseed(123) predsampvalue(15)  ///
   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(5, metric(mse)))
```

Menu

Statistics > H2O machine learning

Syntax

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

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

options	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
h2orseed(#)	set H2O random-number seed for random forest
encode(*encode_type*)	specify H2O encoding type for categorical predictors; default is encode(enum)
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 evaluation 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(), cv[()], h2orseed(), encode(), stop[()], maxtime(), and scoreevery(); see [H2OML] ***h2oml rf***.

Hyperparameter

ntrees(), maxdepth(), minobsleaf(), predsampvalue(), samprate(), minsplitthreshold(), binscat(), binsroot(), and binscont(); see [H2OML] ***h2oml rf***.

Tuning

tune(); see [H2OML] *h2oml rf*.

Remarks and examples

For examples, see *Remarks and examples* in [H2OML] *h2oml rf*.

Stored results

h2oml rfregress 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_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(maxtime)	maximum run time
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
e(cmdline)	command as typed
e(subcmd)	rfregress
e(method)	randomforest
e(method_type)	regression
e(method_full_name)	Random forest regression
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

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 rf*** — Random forest for regression and classification

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

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

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

[U] 20 Estimation and postestimation commands

h2oml postestimation — Postestimation tools for h2oml gbm and h2oml rf

Postestimation commands | h2omlpredict | Remarks and examples | References
Also see

Postestimation commands

The following postestimation commands are of special interest after h2oml *gbm* and h2oml *rf*:

Command	Description
Estimation results and postestimation frame	
h2omlest	store and restore estimation results
h2omlpostestframe	specify frame for postestimation analysis
Tuning and estimation summaries	
h2omlestat metrics	display performance metrics
h2omlgraph scorehistory	produce score history plot
h2omlestat cvsummary	display cross-validation summary
h2omlestat gridsummary	display grid-search summary
h2omlexplore	explore models after grid search
h2omlselect	select model after grid search
h2omlgof	compare goodness of fit for machine learning models
Model performance after binary classification	
h2omlestat threshmetric	display threshold-based metrics
h2omlgraph prcurve	produce precision–recall curve plot
h2omlgraph roc	produce ROC curve plot
Model performance after multiclass classification	
h2omlestat aucmulticlass	display AUC and AUCPR metrics
h2omlestat hitratio	display hit-ratio table
Model performance after binary and multiclass classification	
h2omlestat confmatrix	display confusion matrix
Prediction	
h2omlpredict	predict continuous responses, probabilities, and classes
Model explainability	
h2omlgraph varimp	produce variable importance plot
h2omlgraph pdp	produce partial dependence plot
h2omlgraph ice	produce individual conditional expectation plot
h2omltree	save decision tree DOT file and display rule set
Explainability after regression and binary classification	
h2omlgraph shapvalues	produce SHAP values plot for individual observations
h2omlgraph shapsummary	produce SHAP beeswarm plot

h2omlpredict

Description for h2omlpredict

`h2omlpredict` generates new variables (H2O columns) containing predictions, probabilities, and class predictions. The latter two are provided for the binary and multiclass classification problems.

Menu for h2omlpredict

Statistics > H2O machine learning

Syntax for h2omlpredict

After `h2oml gbregress` and `h2oml rfregress`

`h2omlpredict` *newvar* [, `frame(`*framename*`)`]

After `h2oml gbbinclass` and `h2oml rfbinclass`

`h2omlpredict` *stub*`*` | *newvar* | *newvarlist* [, *binopts* `frame(`*framename*`)`]

After `h2oml gbmulticlass` and `h2oml rfmulticlass`

`h2omlpredict` *stub*`*` | *newvar* | *newvarlist* [, *multopts* `frame(`*framename*`)`]

binopts	Description
Main	
`class`	predicted classes
`pr`	predicted probability of each class
`threshold(#)`	specify threshold for predicting classes

multopts	Description
Main	
`class`	predicted classes
`pr`	predicted probability of each class
`outcome(`*outcome*`)`	specify outcome level (class) for which probabilities are computed

You specify one or k new variables with `pr`, where k is the number of outcomes. If you specify one new variable and you do not specify `outcome()`, then `outcome(#1)` is assumed.

Options for h2omlpredict

Main

`frame(`*framename*`)` specifies the H2O frame in which predictions are stored.

class computes class predictions for each observation and is the default. For h2oml gbbinclass and h2oml rfbinclass, the predicted class for each observation is determined based on a threshold value. By default, the threshold is set to maximize the F1 score. Alternatively, a custom threshold can be specified using the threshold() option. For h2oml gbmulticlass and h2oml rfmulticlass, the predicted class for each observation is based on the highest predicted probability. Only one of class or pr is allowed.

pr computes the predicted probabilities for all outcome levels (classes) or for a specific outcome level (class) after classification. To compute probabilities for all outcome levels, you specify k new variables (H2O columns), where k is the number of classes of the response. Alternatively, you can specify *stub**, in which case pr will store predicted probabilities in variables (H2O columns) *stub1*, *stub2*, ..., *stubk*. To compute the probability for a specific outcome level, you specify one new variable (H2O column) and, optionally, the outcome value in option outcome(); if you omit outcome(), then the first outcome value, outcome(#1), is assumed. Say that you fit a model by typing h2oml *estimation_cmd* y x1 x2, and y has four classes. Then you could type h2omlpredict p1 p2 p3 p4, pr to obtain all four predicted probabilities; alternatively, you could type h2omlpredict p*, pr to generate the four predicted probabilities. To compute specific probabilities one at a time, you can type h2omlpredict p1, pr outcome(#1) (or simply h2omlpredict p1, pr); h2omlpredict p2, pr outcome(#2); and so on. See the outcome() option for other ways to refer to the outcome value. Only one of pr or class is allowed.

threshold(#) specifies the threshold for predicted classes for binary classification. The specified number should be between $[0, 1]$. By default, the threshold value that maximizes the F1 metric is used.

outcome(*outcome*) specifies for which outcome level (class) the predicted probabilities are to be calculated after multiclass classification. outcome() should contain either one class of the response or one of #1, #2, ..., with #1 meaning the first class of the response, #2 meaning the second class, etc. outcome() is not allowed with class.

Remarks and examples

Remarks and examples are presented under the following headings:

Binary classification prediction
Multiclass classification prediction
Testing frame prediction
Regression prediction

Binary classification prediction

▷ Example 1

In this example, we show how to use the h2omlpredict command to predict probabilities and classes for binary classification.

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 see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)
Progress (%): 0 100

. _h2oframe change auto
```

We use h2oml rfbinclass to perform random forest binary classification to predict classes of the car origin.

```
. global predictors price mpg length weight

. h2oml rfbinclass foreign $predictors, ntrees(100) h2orseed(19)
Progress (%): 0 100

Random forest binary classification using H2O

Response: foreign
Frame:                                 Number of observations:
  Training: auto                                   Training =     74

Model parameters

Number of trees      = 100
             actual = 100
Tree depth:                            Pred. sampling value =     -1
          Input max =  20              Sampling rate        =   .632
                min =   3              No. of bins cat.     =  1,024
                avg = 5.5              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
Log loss	.3053323
Mean class error	.1284965
AUC	.9309441
AUCPR	.8455917
Gini coefficient	.8618881
MSE	.1046538
RMSE	.3235024

Next we use h2omlpredict to create a new variable (a column in the current H2O frame) containing the predicted classes.

```
. h2omlpredict foreignhat, class
Progress (%): 0 100
```

The threshold value is a cutpoint that determines the predicted classes from the predicted probabilities. In binary classification, the threshold is the value that maximizes the F1 score. We can determine this threshold value by using `h2omlestat threshmetric`.

```
. h2omlestat threshmetric

Maximum or minimum metrics using H2O
Training frame: auto
```

Metric	Max/Min	Threshold
F1	.7778	.125
F2	.8871	.0732
F0.5	.7979	.6286
Accuracy	.8649	.6286
Precision	1	1
Recall	1	.0732
Specificity	1	1
Min. class accuracy	.8269	.2258
Mean class accuracy	.8715	.125
True negatives	52	1
False negatives	0	.0732 +
True positives	22	.0732
False positives	0	1 +
True-negative rate	1	1
False-negative rate	0	.0732 +
True-positive rate	1	.0732
False-positive rate	0	1 +
MCC	.6855	.125

```
+ identifies minimum metrics.
```

The threshold that maximizes the F1 score is 0.125. Thus, the observations with predicted probabilities greater than 0.125 are assigned to the positive class (`Foreign` in our example), and the remaining observations are assigned to the negative class (`Domestic` in our example). We can specify a different threshold with the `threshold()` option. For example, we can select the threshold that maximizes the true-positive rate, which is 0.0732.

```
. h2omlpredict foreignhat_tpr, class threshold(0.0732)
```

If we want to obtain predicted probabilities, we can use the `pr` option.

```
. h2omlpredict foreignpr1 foreignpr2, pr
Progress (%): 0 100
```

We can get the predictions and the rest of the data in the H2O frame back into Stata by using the `_h2oframe get` command.

```
. clear
. _h2oframe get auto
```

◁

Multiclass classification prediction

▷ Example 2

In this example, we show how to use the `h2omlpredict` command to predict probabilities and classes for multiclass classification.

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. We then use the `_h2oframe split` command to randomly split the `iris` frame into a training frame (80% of observations) and a testing frame (20% of observations), which we name `train` and `test`, respectively. We also change the current frame to `train`.

```
. use https://www.stata-press.com/data/r19/iris
(Iris data)
. h2o init
  (output omitted)
. _h2oframe put, into(iris)
Progress (%): 0 100
. _h2oframe split iris, into(train test) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

Next, we use `h2oml rfmulticlass` to perform random forest multiclass classification.

```
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, ntrees(100) h2orseed(19)
Progress (%): 0 100
Random forest multiclass classification using H2O
Response: iris                          Number of classes    =      3
Frame:                                  Number of observations:
  Training: train                                   Training =    125
Model parameters
Number of trees       = 100
               actual = 100
Tree depth:                             Pred. sampling value =     -1
            Input max =  20             Sampling rate        =   .632
                  min =   1             No. of bins cat.     =  1,024
                  avg = 3.5             No. of bins root     =  1,024
                  max =   8             No. of bins cont.    =     20
Min. obs. leaf split =   1              Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Log loss	.1282741
Mean class error	.0650407
MSE	.0389344
RMSE	.197318

Now, we use `h2omlpredict` to obtain the predicted classes of the iris plant.

```
. h2omlpredict irishat, class
Progress (%): 0 100
```

For multiclass classification, the class is assigned based on the class with the largest predicted probability. We can use the pr option to see the predicted probabilities. The number of specified new variable names should correspond to the number of classes (or we can specify *stub**, such as irispr*).

```
. h2omlpredict irispr1 irispr2 irispr3, pr
Progress (%): 0 100
```

By default, the variables (H2O columns) corresponding to the predicted probabilities and classes are created in the current frame, which in our case is train.

Testing frame prediction

▷ Example 3

We continue the previous example and show how to obtain predictions on the testing data. In general, there are two approaches to achieve this goal.

In the first approach, which we recommend, we use the h2omlpostestframe command.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
. h2omlpredict irishat, class
Progress (%): 0 100
```

The above commands generate variable irishat in the frame test.

In the second approach, we use the frame() option.

```
. h2omlpredict irishat1, class frame(test)
```

Note that neither approach physically changes the working frame to the specified frame, test.

If we are interested in listing the generated variable, then we can type the following.

```
. _h2oframe change test
. _h2oframe list in 1/5
     iris  seplen  sepwid  petlen  petwid  irishat  irishat1
1 Setosa     4.7     3.2     1.3      .2   Setosa    Setosa
2 Setosa     5.1     3.8     1.5      .3   Setosa    Setosa
3 Setosa     5.1     3.7     1.5      .4   Setosa    Setosa
4 Setosa     5.5     4.2     1.4      .2   Setosa    Setosa
5 Setosa     4.9     3.6     1.4      .1   Setosa    Setosa
[5 rows x 7 columns]
```

Regression prediction

▷ Example 4

In this example, we show how to obtain predictions for regression.

We again use auto.dta.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)
Progress (%): 0 100

. _h2oframe change auto
```

We perform gradient boosting regression to predict prices.

```
. h2oml gbregress price mpg weight length, ntrees(100) 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      = 100                   Learning rate       =     .1
              actual = 100                   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 = 4.1                   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    1612524
       MSE    1612524
      RMSE   1269.852
     RMSLE   .1750365
       MAE   853.3532
 R-squared   .8121031
```

Then we use h2omlpredict to obtain predictions.

```
. h2omlpredict pricehat
Progress (%): 0 100
```

The new variable (H2O column) pricehat now contains the predicted prices based on our model.

◁

References

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

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.

Also see

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

h2omlest — Store and restore H2OML estimation results

Description | Quick start | Menu | Syntax
Option | Remarks and examples | Stored results | Also see

Description

`h2omlest` allows you to store, restore, list, and drop estimation results after `h2oml` *gbm* or `h2oml` *rf*.

`h2omlest store` *name* stores the current (active) estimation results as *name*.

`h2omlest restore` *name* loads the specified results into the current (active) estimation results.

`h2omlest dir` displays a list of the stored estimates.

`h2omlest drop` *namelist* drops the specified stored estimation results.

`h2omlest clear` drops all stored estimation results.

`h2omlest clear`, `h2omlest drop _all`, and `h2omlest drop *` do the same thing. `h2omlest drop` and `h2omlest clear` do not eliminate the current (active) estimation results.

Quick start

Store estimation results as `m1` for use later in the same session

    h2omlest store m1

Restore estimation results from `m2`

    h2omlest restore m2

Drop stored estimation results `m3`

    h2omlest drop m3

Drop all stored results

    h2omlest clear

Display table of information about all stored results

    h2omlest dir

Menu

Statistics > H2O machine learning

Syntax

h2omlest store name [, nocopy]

h2omlest restore name

h2omlest dir

h2omlest drop *namelist*

h2omlest clear

where *namelist* is a name, a list of names, _all, or *. _all and * mean the same thing.

Option

nocopy, used with h2omlest store, specifies that the current (active) estimation results be moved into *name* rather than copied. Typing

```
. h2omlest store hold, nocopy
```

is the same as typing

```
. h2omlest store hold
. ereturn clear
```

except that the former is faster. The nocopy option is sometimes used by programmers.

Remarks and examples

h2omlest store stores estimation results in memory after h2oml *rf* and h2oml *gbm* so that you can access them later.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
. _h2oframe change auto
. h2oml gbregress price weight displ
  (output omitted)
. h2omlest store myreg

. ... you do other things, including fitting other models ...

. h2omlest restore myreg
. h2oml gbregress
  (same output shown again)
```

After h2omlest restore myreg, things are once again as they were, estimationwise, just after you typed h2oml gbregress price weight displ.

h2omlest store stores results in memory. When you exit Stata, those stored results vanish.

You make copies in memory so that you can quickly switch between them and so that you can compare estimation results. Concerning the latter, see [H2OML] **h2omlgof**.

Stored results

h2omlest dir stores the following in r():

Macros
  r(names) names of stored results

Also see

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

h2omlestat aucmulticlass — Display AUC and AUCPR after multiclass classification

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

Description

h2omlestat aucmulticlass reports area under the curve (AUC) and area under the precision–recall curve (AUCPR) metrics after multiclass classification performed by h2oml gbmulticlass or h2oml rfmulticlass. These metrics measure how well the model can classify observations. Unlike after binary classification, multiple variations of AUC and AUCPR metrics can be defined with multiclass classification. The variations include one-versus-one metrics, one-versus-rest metrics, and averages of these metrics.

AUC and AUCPR metrics can be computationally intensive. To obtain these metrics, the auc option must be specified in the h2oml gbmulticlass or h2oml rfmulticlass command before the metrics can be reported by h2omlestat aucmulticlass.

Quick start

Report AUC and AUCPR metrics

```
h2omlestat aucmulticlass
```

Same as above, but report testing results based on data in frame test

```
h2omlestat aucmulticlass, test(test)
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlestat aucmulticlass` [, *options*]

options	Description
`title(`*string*`)`	specify title to be displayed above the table
`train`	specify that metrics be reported using training results
`valid`	specify that metrics be reported using validation results
`cv`	specify that metrics be reported using cross-validation results
`test`	specify that metrics be computed using the testing frame
`test(`*framename*`)`	specify that metrics be computed using data in testing frame *framename*
`frame(`*framename*`)`	specify that metrics be computed using data in H2O frame *framename*
`framelabel(`*string*`)`	label frame as *string* in the output

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

`train`, `valid`, `cv`, `test`, `test()`, `frame()`, and `framelabel()` do not appear in the dialog box.

Options

`title(`*string*`)` specifies the title to be displayed above the table.

The following options are available with `h2omlestat aucmulticlass` but are not shown in the dialog box:

`train`, `valid`, `cv`, `test`, `test()`, and `frame()` specify the H2O frame for which AUC and AUCPR metrics are reported. Only one of `train`, `valid`, `cv`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that AUC and AUCPR metrics be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that AUC and AUCPR metrics be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`cv` specifies that AUC and AUCPR metrics be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `cv` may be specified only when the `cv` or `cv()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that AUC and AUCPR metrics be computed on the testing frame specified with `h2oml-postestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set with `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that AUC and AUCPR metrics be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, `h2omlpostestframe` provides a more convenient and computationally efficient process for doing this.

`frame(`*framename*`)` specifies that AUC and AUCPR metrics be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output. This option is not allowed with the `cv` option.

Remarks and examples

`h2omlestat aucmulticlass` computes AUC and AUCPR metrics after multiclass classification. These metrics measure how well the model can classify observations. Unlike with binary classification, observations are not classified into simply one positive and one negative class. Instead, with multiclass classification, variations of these metrics are defined. The one-versus-one metrics compute the AUC and AUCPR for all pairwise combinations of the classes. The one-versus-rest metrics compute the AUC and AUCPR for each class versus all the other classes combined. `h2omlestat aucmulticlass` reports all one-versus-one and one-versus-rest AUC and AUCPR metrics. It also reports the macro (unweighted) average and the prevalence weighted average of each metric. For definitions of these metrics, see [H2OML] ***metric_option***.

Because calculation of the AUC and AUCPR metrics is computationally expensive for multiclass classification, these metrics are not calculated by default by `h2oml gbmulticlass` and `h2oml rfmulticlass`. To enable the calculation, we must specify the `auc` option during estimation. Additionally, AUC and AUCPR metrics may not be requested when the number of response classes is greater than 50.

▷ Example 1: AUC and AUCPR metrics

We 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. 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 see [H2OML] **H2O setup**.

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

We define the global macro `predictors` to store the names of the predictors, and we use the `h2oml rfmulticlass` command to perform random forest multiclass classification. We use default settings for all hyperparameters, and we specify an H2O random-number seed for reproducibility. We also specify the `auc` option to request that the AUC and AUCPR metrics be computed.

```
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, h2orseed(19) auc
Progress (%): 0 100
Random forest multiclass classification using H2O
Response: iris                        Number of classes    =      3
Frame:                                Number of observations:
  Training: iris                                  Training =    150
Model parameters
Number of trees      =  50
              actual =  50
Tree depth:                           Pred. sampling value =     -1
           Input max =  20            Sampling rate        =   .632
                 min =   1            No. of bins cat.     =  1,024
                 avg = 3.7            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
------------------+-----------
         Log loss |   .3438683
 Mean class error |   .0533333
              AUC |   .9906667
            AUCPR |   .9816699
              MSE |   .0384685
             RMSE |    .196134
------------------------------
Note: AUC and AUCPR computed
      using macro average OVR.
```

The output reports an AUC of 0.991 and an AUCPR of 0.982. The note at the bottom of the table tells us that these values are the macro average OVR (one-versus-rest) metrics.

To report all computed AUC and AUCPR metrics, we type

```
. h2omlestat aucmulticlass

AUC and AUCPR summary using H2O
Training frame: iris
```

	AUC	AUCPR
One vs. rest (OVR)		
Setosa vs. rest	1	1
Versicolor vs. rest	.983	.978
Virginica vs. rest	.989	.967
Macro OVR	.991	.982
Weighted OVR	.991	.982
One vs. one (OVO)		
Setosa vs. Versicolor	.995	.997
Setosa vs. Virginica	1	1
Versicolor vs. Virginica	.977	.974
Macro OVO	.991	.99
Weighted OVO	.991	.99

As with standard AUC, a value closer to 1 for each of these metrics indicates better classification. In the first table, we see the one-versus-rest AUC values followed by the one-versus-one AUC values. The `Setosa vs. Rest` AUC value is 1. This means that if we run a binary classification where `Setosa` is considered the positive class and the remaining classes are considered the negative class, then the model will perfectly classify all observations.

Similarly, the `Versicolor vs. Rest` AUC is the AUC for a binary classification where `Versicolor` is treated as the positive class and the other classes jointly comprise the negative class. `Macro OVR` is an unweighted average of the above one-versus-rest AUCs that gives all classes the same weight. `Weighted OVR` is a prevalence weighted average of the one-versus-rest AUCs, where weights are assigned to classes based on the number of positives in each class.

In the next portion of the first table, the AUCs are computed by treating one class as the positive class and one class as the negative class while ignoring all other classes.

The second table can be interpreted similarly to the first table, but it reports AUCPR metrics rather than AUC metrics. The AUCPR is preferred when the classes of the response variable are highly imbalanced.

In this example, all the reported AUC and AUCPR metrics are close to 1, indicating that the model can accurately distinguish between each class and the other classes. However, as we illustrate in the next example, this does not mean that the model is highly accurate at performing multiclass classification in terms of assigning the correct class to every observation.

◁

▷ Example 2: AUC and AUCPR for validation and testing frames

Above, we performed classification and evaluated metrics using a single training frame. To demonstrate how to obtain the AUC and AUCPR metrics for other frames, such as validation and testing frames, we first use the `_h2oframe _split` command to split the dataset, specifying 60% of observations in the training frame, 20% in the validation frame, and 20% in the testing frame. We then change to the training frame.

```
. use https://www.stata-press.com/data/r19/iris, clear
(Iris data)
. h2o init
. _h2oframe put, into(iris)
. _h2oframe split iris, into(training validation testing) split(0.6 0.2 0.2)
> rseed(19)
. _h2oframe change training
```

Next we perform random forest multiclass classification, setting the number of trees to 500 and leaving the other hyperparameters at their default values. We also specify the name of our validation frame in the `validframe()` option.

```
. h2oml rfmulticlass iris $predictors, h2orseed(19) auc ntrees(500)
> validframe(validation)
Progress (%): 0 42.1 68.8 90.2 100
Random forest multiclass classification using H2O
Response: iris                         Number of classes    =      3
Frame:                                 Number of observations:
  Training:   training                            Training =     95
  Validation: validation                        Validation =     30
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 = 3.0             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	.1027022	.1406913
Mean class error	.0423591	.0666667
AUC	.995535	1
AUCPR	.9915411	1
MSE	.0300273	.0473201
RMSE	.1732838	.2175318

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

Now we can run h2omlestat aucmulticlass to see how well our model classifies the data in the validation frame. Because we specified the validation frame during estimation, h2omlestat aucmulticlass defaults to reporting metrics for the validation frame.

```
. h2omlestat aucmulticlass

AUC and AUCPR summary using H2O
Validation frame: validation

─────────────────────────────────────────────────────────────
                                  │        AUC     AUCPR
──────────────────────────────────┼──────────────────────────
One vs. rest (OVR)                │
                 Setosa vs. rest  │          1         1
             Versicolor vs. rest  │          1         1
              Virginica vs. rest  │          1         1
                       Macro OVR  │          1         1
                    Weighted OVR  │          1         1
                                  │
One vs. one (OVO)                 │
           Setosa vs. Versicolor  │          1         1
            Setosa vs. Virginica  │          1         1
        Versicolor vs. Virginica  │          1         1
                       Macro OVO  │          1         1
                    Weighted OVO  │          1         1
─────────────────────────────────────────────────────────────
```

We get a score of 1 for each of the one-versus-rest AUC metrics, meaning that if we performed three binary classifications, one for each class being positive while the rest of the classes are negative, those models will correctly classify all observations. Similarly, all the one-versus-one AUC metrics are 1, corresponding to perfect prediction for all pairwise binary classifications where one class is considered positive and another is considered negative.

However, it is important to remember that computation of one-versus-one AUC and one-versus-rest AUC metrics ignores the fact that the initial problem is multiclass. The results can differ compared with other performance metrics that take into account the true multiclass nature of the problem. For example, let's look at the confusion matrix by using the h2omlestat confmatrix command.

```
. h2omlestat confmatrix

Confusion matrix using H2O
Validation frame: validation

             │              Predicted
        iris │     Setosa  Versico~r  Virginica │  Total  Error    Rate
─────────────┼──────────────────────────────────┼──────────────────────
      Setosa │         12          0          0 │     12      0       0
  Versicolor │          0          8          0 │      8      0       0
   Virginica │          0          2          8 │     10      2      .2
─────────────┼──────────────────────────────────┼──────────────────────
       Total │         12         10          8 │     30      2    .067
```

We see that Setosa and Versicolor were perfectly classified, but the model did misclassify some Virginica flowers as Versicolor.

In addition to the default metrics that are reported for the validation frame in this case, we can obtain metrics for other frames. Here we are interested in results from the testing frame, and we have two ways to request these. One approach is to use the test(testing) option to specify the testing frame. The second approach, our preferred method, is to use h2omlpostestframe to set the testing frame to be used as the default for all affected postestimation commands. For details, see [H2OML] **h2omlpostestframe**.

```
. h2omlpostestframe testing
(testing frame testing is now active for h2oml postestimation)
. h2omlestat aucmulticlass
AUC and AUCPR summary using H2O
Testing frame: testing

                                        AUC   AUCPR

One vs. rest (OVR)
               Setosa vs. rest            1       1
           Versicolor vs. rest            1       1
            Virginica vs. rest            1       1
                     Macro OVR            1       1
                  Weighted OVR            1       1

One vs. one (OVO)
         Setosa vs. Versicolor            1       1
          Setosa vs. Virginica            1       1
      Versicolor vs. Virginica            1       1
                     Macro OVO            1       1
                  Weighted OVO            1       1
```

As with the validation frame, we obtain values of 1 for all AUC and AUCPR metrics calculated on the testing frame.

◁

Stored results

`h2omlestat aucmulticlass` stores the following in `r()`:

Matrices
`r(aucmulticlass)` one-versus-rest and one-versus-one AUC and AUCPR scores

References

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

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.

Also see

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

[H2OML] **h2omlestat confmatrix** — Display confusion matrix

h2omlestat confmatrix — Display confusion matrix

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

Description

`h2omlestat confmatrix` displays a confusion matrix after binary or multiclass classification performed by `h2oml gbbinclass`, `h2oml rfbinclass`, `h2oml gbmulticlass`, or `h2oml rfmulticlass`. A confusion matrix is a summary table for the prediction performance of a machine learning classification model. It displays how different observations are classified based on correct and incorrect predictions. It provides a more informative breakdown of a model's performance than a single metric.

Quick start

Display the confusion matrix after classification

```
h2omlestat confmatrix
```

Same as above, but report confusion matrix based on a validation set

```
h2omlestat confmatrix, valid
```

Same as above, but use a threshold value of 0.5 to determine negative versus positive predicted classes

```
h2omlestat confmatrix, valid threshold(0.5)
```

Menu

Statistics > H2O machine learning

Syntax

h2omlestat confmatrix [, *options*]

options	Description
Main	
metric(*metric*)	specify the metric to be used to select the optimal threshold after binary classification
threshold(#)	specify the threshold value for the predicted probabilities after binary classification
Reporting	
title(*string*)	specify the title to be displayed above the table
labels(*lnames*)	specify label names for rows and columns
nototals	suppress row and column totals
norowtotals	suppress row totals
nocoltotals	suppress column totals
noerrors	suppress the error column
norate	suppress the rate column
train	specify that the confusion matrix be reported using training results
valid	specify that the confusion matrix be reported using validation results
cv	specify that the confusion matrix be reported using cross-validation results
test	specify that the confusion matrix be computed using the testing frame
test(*framename*)	specify that the confusion matrix be computed using data in testing frame *framename*
frame(*framename*)	specify that the confusion matrix be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

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

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

metric(*metric*) specifies the classification metric to be used for selecting a threshold value. This option is valid only after binary classification. *metric* can be one of f1 (the default), f2, fhalf, accuracy, precision, recall, specificity, minclassaccuracy, meanclassaccuracy, tn, fn, tp, fp, tnr, fnr, tpr, fpr, or mcc. For definitions, see [H2OML] ***metric_option***. Only one of metric() or threshold() is allowed.

threshold(#) specifies the cutpoint for the predicted probabilities after binary classification. The specified # must be a value between 0 and 1. Observations with a predicted probability greater than the specified threshold() will be classified as “positive”, and the remaining observations will be classified as “negative”. By default, the selected threshold value maximizes the F1 score. The list of threshold values for which threshold-based metrics are computed corresponds to the predicted probabilities of the positive class (the positive class is the largest numeric value, such as 1 in a 0/1 coded

variable, or the second label in lexicographical order). If the specified `threshold(#)` is not in the list of predicted probabilities, a result based on the closest threshold value is reported. Only one of `threshold()` or `metric()` is allowed.

Reporting

`title(`*string*`)` specifies the title to be displayed above the table.

`labels(`*lnames*`)` specifies the label names for rows and columns. By default, label names show the class names of the categorical response variable. The specified number of labels must be equal to the number of classes of the categorical response variable. The specified labels should be separated by spaces. If the label itself contains spaces, it must be enclosed with double quotes.

`nototals` suppresses the totals for rows and columns. `nototals` is not allowed with `norowtotals` or `nocoltotals`.

`norowtotals` suppresses the totals for rows. `norowtotals` is not allowed with `nototals`.

`nocoltotals` suppresses the totals for columns. `nocoltotals` is not allowed with `nototals`.

`noerrors` suppresses the error column.

`norate` suppresses the rate column.

The following options are available with `h2omlestat confmatrix` but are not shown in the dialog box:

`train`, `valid`, `cv`, `test`, `test()`, and `frame()` specify the H2O frame for which the confusion matrix is reported. Only one of `train`, `valid`, `cv`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that the confusion matrix be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that the confusion matrix be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`cv` specifies that the confusion matrix be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `cv` may be specified only when the `cv` or `cv()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that the confusion matrix be computed on the testing frame specified with `h2oml-postestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set with `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that the confusion matrix be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, `h2omlpostestframe` provides a more convenient and computationally efficient process for doing this.

`frame(`*framename*`)` specifies that the confusion matrix be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output. This option is not allowed with the `cv` option.

Remarks and examples

A confusion matrix is a popular tool for assessing model performance for classification. It consists of a simple grid that contains information about the model's performance in terms of correct and incorrect predictions. A confusion matrix summarizes the types of errors the model makes and allows you to determine areas in which the model predictions can be improved.

Below is an example of a confusion matrix where we predict the origin of a car to be either `Domestic` or `Foreign`. Rows of the confusion matrix correspond to the actual classes, and columns correspond to predicted classes. In H2O, a "positive" class corresponds to a class that contains 1, `True`, or the second label in lexicographical order. In our case, the positive class corresponds to the car origin being `Foreign`.

```
. h2omlestat confmatrix

Confusion matrix using H2O
Training frame: train

             |      Predicted
     foreign |  Domestic    Foreign |  Total   Error    Rate
-------------+----------------------+------------------------
    Domestic |        37          8 |     45       8    .178
     Foreign |         0         18 |     18       0       0
-------------+----------------------+------------------------
       Total |        37         26 |     63       8    .127
Note: Probability threshold .2083 that maximizes F1
      metric used for classification.
```

In this example, the 37 in the upper left cell indicates that there are 37 observations for which the actual class is `Domestic` and the model correctly predicts this class. Because `Domestic` is treated as a "negative" class in this example, the result in this cell is also known as the number of true negatives. On the other hand, 8 is the number of observations belonging to the `Domestic` class that were misclassified by the model as `Foreign`, that is, 8 is the number of false positives. Similarly, 0 and 18 are the numbers of false negatives and true positives, respectively. The predicted class for each observation is determined based on a threshold value of 0.208, which is reported above the table. A predicted probability greater than 0.208 will classify the car as `Foreign`, while a probability below this threshold will classify the car as `Domestic`. By default, `h2omlestat confmatrix` uses the threshold that maximizes the F1 score. However, you can select a threshold value or specify that a threshold be selected that maximizes another metric.

The `Error` column in the output reports the number of misclassified observations for each class, and the `Rate` column reports the misclassification error rate.

When there are more than two classes, the number of rows and columns in the confusion matrix corresponds to the number of classes. The examples below demonstrate `h2omlestat confmatrix` after binary classification. For an example with more than two classes, see example 1 in [H2OML] **h2omlestat aucmulticlass**.

▷ Example 1: Model comparison

In this example, we use the confusion matrix obtained from 3-fold cross-validation to compare two machine learning methods, random forest and gradient boosting machine (GBM), at their default values.

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 see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init

. _h2oframe put, into(auto)

. _h2oframe change auto
```

We run random forest binary classification with 3-fold cross-validation. We store the estimation results by using the `h2omlest store` command so that we can use the results in example 2.

```
. h2oml rfbinclass foreign price mpg trunk weight length, cv(3, modulo)
> h2orseed(19)

Progress (%): 0 100

Random forest binary classification using H2O

Response: foreign
Frame:                                   Number of observations:
  Training: auto                                    Training =     74
                                            Cross-validation =     74
Cross-validation: Modulo                 Number of folds      =      3

Model parameters

Number of trees       =  50
               actual =  50
Tree depth:                              Pred. sampling value =     -1
           Input max =  20               Sampling rate        =   .632
                 min =   4               No. of bins cat.     =  1,024
                 avg = 5.8               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	Cross-validation
Log loss	.7514549	.4192503
Mean class error	.1127622	.1809441
AUC	.9200175	.8706294
AUCPR	.7622589	.624291
Gini coefficient	.840035	.7412587
MSE	.1081766	.1406502
RMSE	.3289021	.3750336

```
. h2omlest store myrf
```

We report the confusion matrix by using the `h2omlestat confmatrix` command.

```
. h2omlestat confmatrix

Cross-validation confusion matrix using H2O

             |      Predicted      |
     foreign |  Domestic   Foreign |  Total  Error    Rate
-------------+---------------------+-----------------------
    Domestic |        45         7 |     52      7    .135
     Foreign |         5        17 |     22      5    .227
-------------+---------------------+-----------------------
       Total |        50        24 |     74     12    .162
Note: Probability threshold .38 that maximizes F1 metric
      used for classification.
```

Because cross-validation was implemented during estimation, by default, `h2omlestat confmatrix` reports results that correspond to cross-validation.

Next we implement GBM and report the confusion matrix.

```
. h2oml gbbinclass foreign price mpg trunk weight length, cv(3, modulo)
> h2orseed(19)
Progress (%): 0 100

Gradient boosting binary classification using H2O

Response: foreign
Loss:     Bernoulli
Frame:                                       Number of observations:
  Training: auto                                        Training =     74
                                                Cross-validation =     74
Cross-validation: Modulo                     Number of folds      =      3

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 = 3.9                   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 |   .0796245    .3856675
 Mean class error |          0    .1284965
              AUC |          1    .9125874
            AUCPR |          1    .8214532
 Gini coefficient |          1    .8251748
              MSE |    .017155    .1286581
             RMSE |   .1309771    .3586894
-----------------------------------------
```

```
. h2omlestat confmatrix
Cross-validation confusion matrix using H2O
```

foreign	Predicted Domestic	Foreign	Total	Error	Rate
Domestic	41	11	52	11	.212
Foreign	1	21	22	1	.045
Total	42	32	74	12	.162

```
Note: Probability threshold .1228 that maximizes F1
      metric used for classification.
```

We can see that random forest is better in predicting `Domestic` cars (45 true negatives versus 41). However, it is not straightforward to quantify how much better because random forest also has more false negatives than does GBM (5 false negatives versus 1). In such cases, we recommend comparing the recall and precision metrics of the two models, which can be obtained from the `h2omlestat threshmetric` command.

In general, when you are interested in quantifying how well a method predicts positives, then the recall metric is recommended.

◁

▷ Example 2: Threshold and metric selection

In example 1, the entries of the confusion matrix were computed using the threshold value that maximizes the F1 score. However, we can instead select a different threshold by using the `threshold()` option or request that `h2omlestat confmatrix` select a threshold value based on optimizing a different metric. Recall that the threshold is a cutoff above which observations are predicted to belong to the positive class and below which observations are predicted to belong to the negative class. Thus, if we change the threshold, the entries of the confusion matrix will also change. Below, we show two confusion matrices with threshold values equal to 0.5 and 0.25 for the random forest.

When we specify the threshold value, `h2omlestat confmatrix` may not report the confusion matrix for the exact value specified. In H2O, the list of possible threshold values for which threshold-based metrics have been computed is limited to the predicted probabilities of the positive class. Therefore, `h2omlestat confmatrix` reports a confusion matrix using the closest available predicted probability of a positive class as the threshold value.

We first restore the random forest estimation results from example 1 with the `h2omlest restore` command and then specify the threshold value in `h2omlestat confmatrix` by using the `threshold(0.25)` option.

```
. h2omlest restore myrf
(results myrf are active now)
. h2omlestat confmatrix, threshold(0.25)
Cross-validation confusion matrix using H2O
```

foreign	Predicted Domestic	Foreign	Total	Error	Rate
Domestic	38	14	52	14	.269
Foreign	3	19	22	3	.136
Total	41	33	74	17	.23

```
Note: Probability threshold .244 that is closest to the
      specified .25 used for classification.
```

Next we obtain the confusion matrix for a threshold value of 0.5.

```
. h2omlestat confmatrix, threshold(0.5)
Cross-validation confusion matrix using H2O

             |      Predicted      |
     foreign |  Domestic   Foreign |  Total   Error    Rate
-------------+---------------------+-----------------------
    Domestic |        46         6 |     52       6    .115
     Foreign |         9        13 |     22       9    .409
-------------+---------------------+-----------------------
       Total |        55        19 |     74      15    .203
Note: Probability threshold .5 used for classification.
```

We can see that different threshold values substantially change the reported results. The selection of the threshold value depends on the problem that the data scientist is trying to answer. For example, if it is important to classify all `Foreign` cars correctly, then we could choose the threshold that maximizes the true-positive rate by specifying the `metric(tpr)` option.

```
. h2omlestat confmatrix, metric(tpr)
Cross-validation confusion matrix using H2O

             |      Predicted      |
     foreign |  Domestic   Foreign |  Total   Error    Rate
-------------+---------------------+-----------------------
    Domestic |        32        20 |     52      20    .385
     Foreign |         0        22 |     22       0       0
-------------+---------------------+-----------------------
       Total |        32        42 |     74      20     .27
Note: Probability threshold .0885 that maximizes
      true-positive rate metric used for classification.
```

◁

Stored results

`h2omlestat confmatrix` stores the following in `r()`:

Scalars
- `r(threshold)` — specified threshold (with option `threshold()`)
- `r(threshold_a)` — actual threshold

Macros
- `r(metric)` — metric for threshold selection

Matrices
- `r(confmatrix)` — confusion matrix

Also see

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

[H2OML] **h2omlestat aucmulticlass** — Display AUC and AUCPR after multiclass classification

[H2OML] **h2omlestat threshmetric** — Display threshold-based metrics for binary classification

h2omlestat cvsummary — Display cross-validation summary

Description Quick start Menu Syntax
Option Remarks and examples Stored results Reference
Also see

Description

`h2omlestat cvsummary` displays the cross-validation summary for each fold after performing cross-validation with `h2oml` *gbm* or `h2oml` *rf*. `h2omlestat cvsummary` reports performance metrics for each fold as well as the mean and standard deviation of each metric. The individual metrics and summary statistics are useful for evaluating the stability of the machine learning method and whether results will generalize well to new data.

Quick start

Display the 5-fold cross-validation summary after `h2oml rfregress`

```
h2oml rfregress y1 x1-x100, cv(5) h2orseed(19)
h2omlestat cvsummary
```

Specify a title for the table

```
h2omlestat cvsummary, title(5-fold CV summary)
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlestat cvsummary` [, `title(`*string*`)`]

Option

`title(`*string*`)` specifies the title to be displayed above the table.

Remarks and examples

We assume you have read *Model selection in machine learning* in [H2OML] **Intro**.

k-fold cross-validation is one of the most common model evaluation and selection techniques. Similar to the two-way holdout method, we start by splitting data into training and testing sets. However, k-fold cross-validation additionally splits the training set into k folds. In each iteration, it uses one fold for validation and the remaining $k-1$ folds as a training subset for model fitting. One way to compute a cross-validation metric is to take the average of the k validation metrics of the cross-validated models. `h2omlestat cvsummary` reports this average along with the standard deviation and the estimated metrics for each fold.

Looking at the standard deviation of cross-validated metrics over the folds can provide useful insights into the stability and reliability of a machine learning model. For example, if the standard deviation across the folds is large, it may indicate that the performance of the model is not consistent across different subsets of data and that the model will not generalize well to new data. A large standard deviation could also indicate data issues; for example, data may be insufficient for reliable training or may suffer from imbalanced classes.

Another common reason for a large standard deviation is the bias–variance tradeoff of the machine learning model. A large standard deviation can indicate overfitting, where the model is too complex and closely learns patterns in the training data. In such cases, a less complex model that provides slightly lower performance metrics but also low variance might be preferable.

Several authors have tried to find the best value of k that minimizes the bias–variance tradeoff. Based on numerous empirical analyses, Kohavi (1995) suggests $k = 10$ folds. However, cross-validation with this many folds can be computationally intensive when the dataset is large. In general, as the number of folds increases, the performance bias decreases but the variance of the performance metric and computational cost increases.

The steps for hyperparameter tuning with k-fold cross-validation are as follows:

1. Split the dataset into two sets—a training set for model fitting and selection and a testing set for the final model evaluation.
2. Perform hyperparameter tuning. For each hyperparameter configuration, apply the k-fold cross-validation method on the training set.
3. Select the best hyperparameter settings from the k-fold cross-validation, and apply them to the entire training set.
4. Use the independent testing set and the hyperparameter setting from the previous step to estimate the generalization performance.

To perform cross-validation with the `h2oml` *gbm* and `h2oml` *rf* commands, we specify the `cv()` option. After estimation, we can use `h2omlestat cvsummary` to summarize performance metrics and examine their results for each fold.

▷ Example 1: Cross-validation summary for bias–variance tradeoff

In this example, we use gradient boosting binary classification on the `auto` dataset to examine the standard deviation of a cross-validated metric as an indicator for overfitting.

We start by opening `auto.dta` in Stata and then putting it in 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. (Because we are focused on evaluating cross-validation, we do not split the data into training and testing sets as we typically would in practice.) For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

We perform gradient boosting binary classification with 3-fold cross-validation and use 5,000 trees.

```
. h2oml gbbinclass foreign price mpg weight length, cv(3, modulo) h2orseed(19)
> ntrees(5000)
Progress (%): 0 0.7 2.5 4.0 5.1 17.0 31.2 32.9 33.7 34.1 34.7 41.1 53.8 100
Gradient boosting binary classification using H2O
Response: foreign
Loss:     Bernoulli
Frame:                                       Number of observations:
  Training: auto                                        Training =     74
                                                Cross-validation =     74
Cross-validation: Modulo                     Number of folds     =      3
Model parameters
Number of trees       = 5,000                Learning rate       =     .1
               actual = 5,000                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 =    2.7                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	Cross-validation
Log loss	1.80e-17	2.487799
Mean class error	0	.1197552
AUC	1	.8902972
AUCPR	1	.7719202
Gini coefficient	1	.7805944
MSE	4.00e-33	.1135748
RMSE	6.32e-17	.3370087

Next we report the cross-validated metrics for each fold, together with the mean and standard deviation.

```
. h2omlestat cvsummary
```

Cross-validation summary using H2O

Metric	Mean	Std. dev.	Fold 1	Fold 2	Fold 3
Log loss	2.467125	2.757786	.8134241	5.650739	.9372107
F1	.8586183	.0740218	.9230769	.7777778	.875
F2	.8872107	.0564633	.882353	.8333333	.9459459
F0.5	.8369541	.1209393	.9677419	.7291667	.8139535
Accuracy	.9055555	.0607667	.96	.84	.9166667
Precision	.825926	.1556878	1	.7	.7777778
Recall	.9107143	.0778375	.8571429	.875	1
Specificity	.9019608	.0898544	1	.8235294	.882353
Misclassification	.0944444	.0607667	.04	.16	.0833333
Mean class error	.0936625	.0498267	.0714286	.1507353	.0588235
Max. class error	.1456583	.0295116	.1428571	.1764706	.1176471
Mean class accuracy	.9063376	.0498267	.9285714	.8492647	.9411765
Misclassification count	2.333333	1.527525	1	4	2
AUC	.919779	.0744504	.984127	.8382353	.9369748
AUCPR	.7621639	.180335	.9663477	.624682	.6954619
MSE	.1134442	.0786849	.0400411	.196517	.1037744
RMSE	.3218485	.1216001	.2001026	.4433024	.3221404

For illustration purposes, we focus on the log-loss metric; for details, see [H2OML] ***metric_option***. In the first row of the output, the mean is 2.47 and the standard deviation is 2.76. Further analysis reveals that fold 2 has a large log-loss metric. One possible explanation is that, given the simplicity of this dataset, fitting a model with a large number of trees might lead to overfitting, which is why the model does not generalize well for data in fold 2. To investigate, we fit a less complex model with the default 50 trees and report the cross-validation results.

```
. h2oml rfbinclass foreign price mpg weight length, cv(3, modulo) h2orseed(19)
Progress (%): 0 100
Random forest binary classification using H2O
Response: foreign
Frame:                                 Number of observations:
  Training: auto                                   Training =     74
                                           Cross-validation =     74
Cross-validation: Modulo               Number of folds      =      3
Model parameters
Number of trees        =  50
              actual =  50
Tree depth:                            Pred. sampling value =     -1
           Input max =  20             Sampling rate        =   .632
                 min =   3             No. of bins cat.     =  1,024
                 avg = 5.6             No. of bins root     =  1,024
                 max =   8             No. of bins cont.    =     20
Min. obs. leaf split =   1             Min. split thresh.   = .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.3097282	.8764794
Mean class error	.1284965	.2036713
AUC	.9278846	.8435315
AUCPR	.8502403	.6751862
Gini coefficient	.8557692	.6870629
MSE	.1088474	.1504919
RMSE	.3299203	.3879328

```
. h2omlestat cvsummary
Cross-validation summary using H2O
```

Metric	Mean	Std. dev.	Fold 1	Fold 2	Fold 3
Log loss	.8879563	.7421946	.3638948	.5627286	1.737245
F1	.7857143	.0795395	.8571429	.7	.8
F2	.8286436	.0311104	.8571429	.7954546	.8333333
F0.5	.7504579	.1172045	.8571429	.625	.7692308
Accuracy	.8516667	.0825126	.92	.76	.875
Precision	.7301587	.1379789	.8571429	.5833333	.75
Recall	.8630952	.0103098	.8571429	.875	.8571429
Specificity	.8442266	.1237666	.9444444	.7058824	.882353
Misclassification	.1483333	.0825126	.08	.24	.125
Mean class error	.1463391	.0569079	.0992063	.2095588	.1302521
Max. class error	.1932773	.0873303	.1428571	.2941177	.1428571
Mean class accuracy	.8536609	.0569079	.9007937	.7904412	.8697479
Misclassification count	3.666667	2.081666	2	6	3
AUC	.843643	.067583	.9206349	.8161765	.7941176
AUCPR	.663395	.0049219	.6678722	.6581247	.6641881
MSE	.150353	.0437331	.112672	.1983087	.1400785
RMSE	.3850852	.0556203	.3356665	.4453186	.3742706

We can see that the mean and standard deviation of the log loss are now much smaller.

◁

Stored results

`h2omlestat cvsummary` stores the following in `r()`:

Matrices

`r(cvsummary)`	summary of cross-validation metrics and metrics for each fold

Reference

Kohavi, R. 1995. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, August 20–25*, vol. 2: 1137–1143. San Francisco: Morgan Kaufman.

Also see

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

h2omlestat gridsummary — Display grid-search summary

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

Description

h2omlestat gridsummary displays the grid summary for configurations of hyperparameters after h2oml *gbm* and h2oml *rf* perform tuning using a grid search.

When tuning is performed, the h2oml *gbm* and h2oml *rf* commands report performance metrics for the best model based on the tuning metric. h2omlestat gridsummary reports the tuning metric or another specified metric for additional models that were evaluated as part of the grid search. It also assigns an ID number to each model. You can then specify these ID numbers in h2omlexplore to compare a variety of performance metrics for the chosen models. You can also use h2omlselect to select a model based on the ID number so that subsequent postestimation commands will be based on this model instead of the one selected by tuning h2oml *gbm* or h2oml *rf*.

Quick start

Display the grid summary of log-loss metrics after h2oml gbbinclass

```
h2oml gbbinclass y x2-x5, ntrees(50(5)80) tune(grid(cartesian))
h2omlestat gridsummary
```

Same as above, but report the grid summary for the area under the curve (AUC) metric

```
h2omlestat gridsummary, metric(auc)
```

Menu

Statistics > H2O machine learning

Syntax

h2omlestat gridsummary [, *options*]

options	Description
metric(*metric*)	specify the metric to be reported
top(#)	report the top # models; top(_all) reports all models; default is top(10)
title(*string*)	specify title to be displayed above the table

Options

metric(*metric*) specifies the metric for which the grid summary will be reported. Allowed metrics are provided in [H2OML] ***metric_option***. If the metric() suboption is specified in the tune() option of the h2oml *gbm* or h2oml *rf* command, then h2omlestat gridsummary will use the same metric. Otherwise, the default metric is deviance for regression and log loss for classification.

top(#) specifies that the top # models be included in the summary table. top(_all) specifies that all models be reported. The default is top(10).

title(*string*) specifies the title to be displayed above the table.

Remarks and examples

To build a machine learning model that generalizes well to new data involves choosing an appropriate method and selecting a model by tuning hyperparameters; see *Hyperparameter tuning* in [H2OML] **Intro** for more information on tuning. For example, suppose we want to perform gradient boosting binary classification and use an exhaustive grid search to select the optimal number of trees. We could type

```
h2oml gbbinclass y x1-x100, ntrees(10(5)100)
```

We can use h2omlestat gridsummary to report the models ranked based on the default log-loss tuning metric.

```
h2omlestat gridsummary
```

Alternatively, we can request a grid summary for another metric, such as the AUC.

```
h2omlestat gridsummary, metric(auc)
```

After reporting the grid-search summary, we can compare models with different hyperparameters based on other performance metrics by using the h2omlexplore command; we select the desired model by using the h2omlselect command. See [H2OML] **h2omlexplore** and [H2OML] **h2omlselect** for examples demonstrating how to use h2omlestat gridsummary in combination with these commands.

▷ Example 1: Sequential hyperparameter tuning

When the dataset is large and there are many hyperparameters, tuning these hyperparameters simultaneously can be computationally intensive. We can reduce the computational burden by tuning hyperparameters sequentially. That is, in the first iteration of tuning, a small set of hyperparameters are tuned to narrow the search space. Then in the second iteration, the best results from the previous iteration can be used with additional hyperparameters. However, note that this procedure might lead us to select suboptimal values for the hyperparameters, and it is only recommended for large datasets. As an alternative, which also may result in a suboptimal solution, one could use a random grid search and restrict the search space by specifying the maxmodels() or maxtime() suboption in the tune() option of the h2oml *gbm* or h2oml *rf* command.

In this example, we use gradient boosting to illustrate the sequential procedure.

We begin by opening the auto.dta dataset in Stata and then putting it 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 see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)
Progress (%): 0 100

. _h2oframe change auto
```

In the first step of our tuning procedure, we tune the maximum depth of the trees hyperparameter using 3-fold cross-validation and an exhaustive grid search. We set the learning rate to 0.05, a little higher than the recommended 0.01, because the learning rate decay is 0.9. For details on gradient boosting machine hyperparameters, see [H2OML] ***h2oml gbm***.

```
. h2oml gbbinclass foreign price mpg weight length, cv(3, modulo) h2orseed(19)
> lratedecay(0.9) lrate(0.05) maxdepth(1(1)10) tune(grid(cartesian))
Progress (%): 0 100

Gradient boosting binary classification using H2O

Response: foreign
Loss:     Bernoulli
Frame:                                      Number of observations:
  Training: auto                                       Training =     74
                                               Cross-validation =     74
Cross-validation: Modulo                    Number of folds     =      3

Tuning information for hyperparameters

Method: Cartesian
Metric: Log loss
```

Hyperparameters	Grid values Minimum	Maximum	Selected
Max. tree depth	1	10	10

```
Model parameters

Number of trees      =  50              Learning rate        =    .05
           actual =  50              Learning rate decay  =     .9
Tree depth:                             Pred. sampling rate  =      1
        Input max =  10              Sampling rate        =      1
              min =   2              No. of bins cat.     =  1,024
              avg = 3.0              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
Log loss	.3679234	.4914566
Mean class error	.0576923	.1958042
AUC	.9820804	.8535839
AUCPR	.9584095	.6989351
Gini coefficient	.9641608	.7071678
MSE	.1063068	.159142
RMSE	.3260472	.398926

Next we use `h2omlestat gridsummary` to report the configurations that achieve the best performance based on the log-loss metric.

```
. h2omlestat gridsummary
Grid summary using H2O

        Max. tree
  ID        depth    Log loss

   1           10    .4914566
   2            3    .4914566
   3            4    .4914566
   4            5    .4914566
   5            6    .4914566
   6            7    .4914566
   7            8    .4914566
   8            9    .4914566
   9            2    .4919681
  10            1    .5266221
```

We see that the performance of the model in terms of the log-loss metric does not change for maximum tree depths between 3 and 10. Therefore, to have a parsimonious model, we select a maximum tree depth of 3. In the second step of our tuning procedure, we specify the `maxdepth(3)` option and tune the learning rate and sampling rate hyperparameters.

```
. h2oml gbbinclass foreign price mpg weight length, cv(3, modulo) h2orseed(19)
> lratedecay(0.9) maxdepth(3) samprate(0.4(0.1)1) lrate(0.2(0.02)0.3)
> tune(grid(cartesian))
Progress (%): 0 100
Gradient boosting binary classification using H2O
Response: foreign
Loss:     Bernoulli
Frame:                                      Number of observations:
  Training: auto                                       Training =     74
                                               Cross-validation =     74
Cross-validation: Modulo                    Number of folds     =      3
Tuning information for hyperparameters
Method: Cartesian
Metric: Log loss

                                           Grid values
 Hyperparameters          Minimum            Maximum         Selected

   Learning rate               .2                 .3              .28
   Sampling rate               .4                  1                1
```

```
Model parameters
Number of trees       =   50            Learning rate        =    .28
                actual =   50            Learning rate decay =     .9
Tree depth:                              Pred. sampling rate =      1
           Input max =    3              Sampling rate       =      1
                 min =    2              No. of bins cat.    =  1,024
                 avg =  3.0              No. of bins root    =  1,024
                 max =    3              No. of bins cont.   =     20
Min. obs. leaf split =   10              Min. split thresh.  = .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.1357221	.2983633
Mean class error	.0227273	.090035
AUC	.9982517	.9370629
AUCPR	.9961309	.8555774
Gini coefficient	.9965035	.8741259
MSE	.0326208	.097178
RMSE	.1806123	.3117338

Once again, we use `h2omlestat gridsummary` to report the configurations that achieve the best performance based on the log-loss metric.

```
. h2omlestat gridsummary
Grid summary using H2O
```

ID	Learning rate	Sampling rate	Log loss
1	.28	1	.2983633
2	.3	1	.2998373
3	.24	1	.3038322
4	.26	1	.3042715
5	.28	.9	.3087905
6	.3	.9	.3102182
7	.22	1	.3137784
8	.26	.9	.3159972
9	.24	.9	.3176375
10	.28	.7	.3319306

We see that the top model achieved a log-loss of 0.298, and the corresponding hyperparameters are a learning rate of 0.28 and a sampling rate of 1.

Stored results

`h2omlestat gridsummary` stores the following in `r()`:

Matrices
`r(gridsummary)` grid-search summary of hyperparameters and metrics

Also see

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

[H2OML] **h2omlexplore** — Explore models after grid search

[H2OML] **h2omlselect** — Select model after grid search

h2omlestat hitratio — Display hit-ratio table

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

Description

h2omlestat hitratio reports hit ratios after multiclass classification performed by h2oml gbmulticlass or h2oml rfmulticlass. A hit ratio measures how often the correct class is within the top-k predicted classes. The top-k hit ratio is the proportion of observations for which the correct class has one of the k highest predicted probabilities.

Quick start

Display the top-k hit ratios

```
h2omlestat hitratio
```

Same as above, but report results for the validation frame

```
h2omlestat hitratio, valid
```

Menu

Statistics > H2O machine learning

Syntax

h2omlestat hitratio [, *options*]

options	Description
title(*string*)	specify title to be displayed above the table
train	specify that hit ratios be reported using training results
valid	specify that hit ratios be reported using validation results
cv	specify that hit ratios be reported using cross-validation results
test	specify that hit ratios be computed using the testing frame
test(*framename*)	specify that hit ratios be computed using data in testing frame *framename*
frame(*framename*)	specify that hit ratios be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

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

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

title(*string*) specifies the title to be displayed above the table.

The following options are available with h2omlestat hitratio but are not shown in the dialog box:

train, valid, cv, test, test(), and frame() specify the H2O frame for which hit ratios are reported. Only one of train, valid, cv, test, test(), or frame() is allowed.

train specifies that hit ratios be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe.

valid specifies that hit ratios be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. valid may be specified only when the validframe() option is specified with h2oml *gbm* or h2oml *rf*.

cv specifies that hit ratios be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. cv may be specified only when the cv or cv() option is specified with h2oml *gbm* or h2oml *rf*.

test specifies that hit ratios be computed on the testing frame specified with h2omlpostestframe. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set with h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that hit ratios be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, h2omlpostestframe provides a more convenient and computationally efficient process for doing this.

frame(*framename*) specifies that hit ratios be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output. This option is not allowed with the cv option.

Remarks and examples

For multiclass classification, the hit ratio measures how often the correct class is in one of the top-k predicted classes, where the top-k predicted classes are ranked by predicted probabilities. For example, when computing the top-2 hit ratio, if the true class for an observation has one of the two highest predicted probabilities, then it is considered a "hit"; it is considered a "miss" otherwise. The top-2 hit ratio is the proportion of observations having such a hit. h2omlestat hitratio provides a table of top-k hit ratios. If there are more than 10 classes, H2O limits the computation to a maximum of top-10 hit ratios.

In practice, the hit ratio is useful in situations where multiple predictions are made and the true class does not need to have the highest predicted probability but does need to be within the top few. For example, in recommendation systems or search engines, the output is presented as a ranked list of results. The correct result needs to be somewhere near the top of that list, but it does not necessarily need to be the first one.

▷ Example 1: Hit ratios

We 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. 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. We also use the _h2oframe split command to split the dataset, specifying 70% of observations in the training frame and 30% in the validation frame. For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/iris
(Iris data)
. h2o init
  (output omitted)
. _h2oframe put, into(iris)
Progress (%): 0 100
. _h2oframe split iris, into(train valid) split(0.7 0.3) rseed(19)
. _h2oframe change train
```

We define the global macro `predictors` to store the names of the predictors, and we use the `h2oml rfmulticlass` command to perform random forest multiclass classification. We use default settings for all hyperparameters, and we specify an H2O random-number seed for reproducibility. We also specify the name of our validation frame in the `validframe()` option.

```
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, validframe(valid) h2orseed(19)
Progress (%): 0 100
Random forest multiclass classification using H2O
Response: iris                          Number of classes    =      3
Frame:                                  Number of observations:
  Training:   train                                Training =    113
  Validation: valid                              Validation =     37
Model parameters
Number of trees       =  50
              actual =  50
Tree depth:                             Pred. sampling value =     -1
           Input max =  20              Sampling rate        =   .632
                 min =   1              No. of bins cat.     =  1,024
                 avg = 3.2              No. of bins root     =  1,024
                 max =   6              No. of bins cont.    =     20
Min. obs. leaf split =   1              Min. split thresh.   = .00001
Metric summary
```

Metric	Training	Validation
Log loss	.0821639	.1523995
Mean class error	.0456654	.0747475
MSE	.0269054	.0555373
RMSE	.1640287	.2356636

The top-1 hit ratio is closely related to the misclassification error, which we will report first by using the `h2omlestat confmatrix` command.

```
. h2omlestat confmatrix
Confusion matrix using H2O
Validation frame: valid
```

iris	Predicted Setosa	Versico~r	Virginica	Total	Error	Rate
Setosa	11	0	0	11	0	0
Versicolor	0	10	1	11	1	.091
Virginica	0	2	13	15	2	.133
Total	11	12	14	37	3	.081

This confusion matrix based on validation results shows that the highest predicted probabilities from the model misclassified three observations, resulting in a misclassification error of 0.08. This means that the top-1 hit ratio is 0.92 ($1 - 0.08$). In other words, the true class has the highest predicted probability for 92% of observations.

To determine the top-2 hit ratio, we need to know whether the true class for each of the three misclassified observations has the second highest predicted probability. To check, we predict the class and corresponding probabilities using the validation frame. By default, `h2omlpredict` generates predictions in the current working frame. (We can use `_h2oframe pwf` to check which is the current

frame.) To make predictions in the validation frame, we set it as our postestimation frame by using the `h2omlpostestframe` command. We use `h2omlpredict` to obtain the predicted class, the default prediction. We then specify the `pr` option to obtain the predicted probabilities of each class.

```
. h2omlpostestframe _valid
(validation frame valid is now active for h2oml postestimation)
. h2omlpredict pr_class
(option class assumed; predicted class)
Progress (%): 0 100
. h2omlpredict pr_setosa pr_versicolor pr_virginica, pr
Progress (%): 0 100
```

Because the `h2omlpostestframe` command does not physically change the current frame, we use the `_h2oframe change` command to change the working frame before listing the misclassified observations.

```
. _h2oframe change valid
. _h2oframe list iris pr_class pr_setosa pr_versicolor pr_virginica
> if pr_class != iris, abbreviate(14)
        iris    pr_class  pr_setosa  pr_versicolor  pr_virginica
1 Versicolor   Virginica          0       .2038981      .7961019
2  Virginica  Versicolor          0       .8080754      .1919246
3  Virginica  Versicolor          0       .8631397      .1368603
[3 rows x 5 columns]
```

In the first row, we see that the model misclassified true class `Versicolor` as `Virginica` with the probability 0.8. For this observation, the probability of predicting `Versicolor`, the true class, is the second highest probability of 0.2. Similarly, for the next two observations, the second highest predicted probability corresponds to the true class. Consequently, for all misclassified observations, the top-2 predicted classes contain the true class; thus, the top-2 hit ratio is 1.

The `h2omlestat hitratio` command provides an easy way to obtain the hit ratios we computed manually.

```
. h2omlestat hitratio
Hit-ratio table using H2O
Validation frame: valid
```

Top	Hit ratio
1	.9189189
2	1
3	1

From this table, we confirm that the true class has the highest predicted probability for 92% of observations in the validation data. The true class has one of the two highest predicted probabilities for 100% of the observations.

In this example, we see top-1, top-2, and top-3 hit ratios. For classification problems in which the response has many classes, `h2omlestat hitratio` will report all top-k hit ratios up to the top-10 hit ratio.

◁

Stored results

h2omlestat hitratio stores the following in r():

Matrices

r(hitratio)	hit ratios

References

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

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.

Also see

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

[H2OML] **h2omlestat aucmulticlass** — Display AUC and AUCPR after multiclass classification

[H2OML] **h2omlestat confmatrix** — Display confusion matrix

h2omlestat metrics — Display performance metrics

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

Description

h2omlestat metrics reports the performance metrics after h2oml *gbm* and h2oml *rf*.

Quick start

Report the performance metrics

    h2omlestat metrics

Same as above, but report performance metrics for the validation frame

    h2omlestat metrics, valid

Report performance metrics for frame myframe

    h2omlestat metrics, frame(myframe)

Menu

Statistics > H2O machine learning

Syntax

h2omlestat metrics [, *options*]

options	Description
train	specify that performance metrics be reported using training results
valid	specify that performance metrics be reported using validation results
cv	specify that performance metrics be reported using cross-validation results
test	specify that performance metrics be computed using the testing frame
test(*framename*)	specify that performance metrics be computed using data in testing frame *framename*
frame(*framename*)	specify that performance metrics be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

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

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

The following options are available with `h2omlestat metrics` but are not shown in the dialog box:

`train`, `valid`, `cv`, `test`, `test()`, and `frame()` specify the H2O frame for which performance metrics are reported. Only one of `train`, `valid`, `cv`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that performance metrics be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that performance metrics be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`cv` specifies that performance metrics be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `cv` may be specified only when the `cv` or `cv()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that performance metrics be computed on the testing frame specified with `h2omlpostestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set with `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that performance metrics be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, `h2omlpostestframe` provides a more convenient and computationally efficient process for doing this.

`frame(`*framename*`)` specifies that performance metrics be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output. This option is not allowed with the `cv` option.

Remarks and examples

`h2omlestat metrics` reports the performance metrics of a machine learning model after `h2oml` *gbm* or `h2oml` *rf*.

The default frame for which metrics are reported depends on options specified in the estimation command and on whether a postestimation frame has been set by using `h2omlpostestframe`.

If no postestimation frame has been set and if neither the `cv()` nor `validframe()` option was specified during estimation, performance metrics are reported for the training frame. If the `validframe()` option is specified during estimation, performance metrics are reported by the validation frame. If the `cv()` option is specified during estimation, performance metrics are reported for cross-validation. If a postestimation frame has been set by `h2omlpostestframe`, the performance metrics are reported for the specified postestimation frame by default; see [H2OML] **h2omlpostestframe**. You can also specify one of the `train`, `valid`, `cv`, `test`, `test()`, or `frame()` options with `h2omlestat metrics` to indicate the frame for which metrics are reported.

▷ Example 1: Performance metrics on different frames

In this example, we demonstrate how to obtain performance metrics based on multiple frames after estimation.

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. We then use the `_h2oframe split` command to randomly split the `auto` frame into a training frame (80% of observations) and a testing frame (20% of observations), which we name `train` and `test`, respectively. We also change the current frame to `train`. 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/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)

. _h2oframe split auto, into(train test) split(0.8 0.2) rseed(19)

. _h2oframe change train
```

We perform random forest binary classification with default hyperparameters and use 3-fold cross-validation.

```
. h2oml rfbinclass foreign price mpg length, cv(3, modulo) h2orseed(19)
  (output omitted)
```

By default, because cross-validation was used during estimation, `h2omlestat metrics` reports estimation metrics based on cross-validation.

```
. h2omlestat metrics

Performance metrics using H2O
Random forest binary classification

Response: foreign

Number of observations = 63
```

Metric	Cross-validation
Log loss	.4275175
Mean class error	.1777778
AUC	.8666667
AUCPR	.6008256
Gini coefficient	.7333333
MSE	.1446453
RMSE	.3803227

If we wish to compute and report results based on a testing frame, we can set the testing frame with the `h2omlpostestframe` command.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
. h2omlestat metrics
Performance metrics using H2O
Random forest binary classification
Response:      foreign
Testing frame: test
Number of observations = 11
```

Metric	Testing
Log loss	.3117297
Mean class error	.0714286
AUC	.9285714
AUCPR	.8722936
Gini coefficient	.8571429
MSE	.1053455
RMSE	.3245696

◁

Stored results

`h2omlestat metrics` stores the following in `r()`:

Scalars

`r(N)`	number of observations

Macros

`r(method)`	`gbm` or `randomforest`
`r(method_type)`	`regression` or `classification`
`r(class_type)`	`binary` or `multiclass` (with classification)
`r(method_full_name)`	full method name
`r(response)`	name of response
`r(title)`	title in output
`r(loss)`	name of the loss function (only after `h2oml` *gbm*)

Matrices

`r(metric)`	performance metrics

Also see

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

h2omlestat threshmetric — Display threshold-based metrics for binary classification

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

Description

`h2omlestat threshmetric` reports threshold-based performance metrics after binary classification performed by `h2oml gbbinclass` or `h2oml rfbinclass`. Threshold-based metrics are functions of predicted classes, which are determined by comparing predicted probabilities with a threshold value. Observations with predicted probabilities greater than the threshold are predicted to be in the "positive" class, and observations with predicted probabilities below the threshold are predicted to be in the "negative" class. The elements of the confusion matrix—the numbers of true positives, false positives, true negatives, and false negatives—are threshold-based metrics and are components of a variety of additional threshold-based metrics that are reported by `h2omlestat threshmetric`. Each of these metrics has a different threshold value.

`h2omlestat threshmetric` reports the optimized (minimum or maximum) value of each metric and the corresponding threshold that produces that optimized metric. Alternatively, the metrics can be reported for one or more selected threshold values.

Quick start

Display threshold-based metrics

```
h2omlestat threshmetric
```

Same as above, but report metrics based on a validation set

```
h2omlestat threshmetric, valid
```

Same as above, but report metrics corresponding to threshold values of 0.4, 0.5, 0.6, 0.7, and 0.8

```
h2omlestat threshmetric, valid thresholds(0.4(0.1)0.8)
```

Menu

Statistics > H2O machine learning

Syntax

h2omlestat threshmetric [, *options*]

options	Description
Main	
thresholds(*numlist*)	specify the thresholds for which to compute the metrics; by default, the threshold that optimizes each metric is reported
Table options	
all	report metrics for all stored threshold values
index	display threshold index
title(*string*)	specify the title to be displayed above the table
train	specify that performance metrics be reported using training results
valid	specify that performance metrics be reported using validation results
cv	specify that performance metrics be reported using cross-validation results
test	specify that performance metrics be computed using the testing frame
test(*framename*)	specify that performance metrics be computed using data in testing frame *framename*
frame(*framename*)	specify that performance metrics be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

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

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

thresholds(*numlist*) specifies the list of threshold values in *numlist*. All values in *numlist* must be between 0 and 1. Observations with predicted probabilities greater than the specified threshold are classified as “positive”, and the remaining observations are classified as “negative”. The threshold-based metrics are calculated based on these classifications. By default, the threshold values that optimize (maximize or minimize) each metric are reported.

The list of threshold values for which threshold-based metrics are computed corresponds to the predicted probabilities of the positive class (the predicted class is the largest numeric value, such as 1 in a 0/1 coded variable, or the second label in lexicographical order). If a value specified in *numlist* is not in the list of predicted probabilities, the metric based on the closest threshold value is reported. thresholds() is not allowed with all.

Table options

all returns all stored threshold values and metrics. The default is to report the optimized (maximum or minimum) values for each metric. all is not allowed with thresholds().

index displays the index number of the threshold. By default, the index column is suppressed.

title(*string*) specifies the title to be displayed above the table.

The following options are available with `h2omlestat threshmetric` but are not shown in the dialog box:

`train`, `valid`, `cv`, `test`, `test()`, and `frame()` specify the H2O frame for which performance metrics are reported. Only one of `train`, `valid`, `cv`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that performance metrics be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that performance metrics be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`cv` specifies that performance metrics be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `cv` may be specified only when the `cv` or `cv()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that performance metrics be computed on the testing frame specified with `h2omlpostestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set with `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that performance metrics be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, `h2omlpostestframe` provides a more convenient and computationally efficient process for doing this.

`frame(`*framename*`)` specifies that performance metrics be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output. This option is not allowed with the `cv` option.

Remarks and examples

Binary classification divides observations into two classes, typically labeled as "positive" and "negative". In H2O, the positive class corresponds to the class that contains 1, `True`, or the second label in lexicographical order. A binary classifier classifies all observations as either positive or negative by comparing the predicted probability for each observation with a threshold value. Observations greater than the threshold are classified as positive, and the remaining observations are classified as negative. This results in two types of correct or true classification, true positive and true negative, and two types of incorrect or false classification, false positive and false negative. These four metrics are reported in the confusion matrix produced by the `h2omlestat confmatrix` command. The `h2omlestat threshmetric` command reports these metrics as well as other performance metrics that are derived from the elements of a confusion matrix.

By default, `h2omlestat threshmetric` reports the optimized (minimum or maximum) value of each metric and the corresponding threshold value that produces the optimized metric. You can also evaluate how different threshold values affect each metric by specifying one or more threshold values in the

thresholds() option. When you specify the thresholds() option, metrics may not be reported for the exact threshold values you have selected. In H2O, the available thresholds are limited to the list of predicted probabilities of the positive class. Threshold-based metrics are reported for the threshold corresponding to the closest available predicted probability.

The table below provides definitions of the available threshold-based metrics. See *Metrics for classification* in [H2OML] ***metric_option*** for additional information.

Metric	Formula
true positive (tp)	number of correct predictions of the positive class
true negative (tn)	number of correct predictions of the negative class
false positive (fp)	number of incorrect predictions of the positive class
false negative (fn)	number of incorrect predictions of the negative class
true-positive rate (tpr), recall	$\frac{\text{tp}}{\text{tp}+\text{fn}}$
true-negative rate (tnr)	$\frac{\text{tn}}{\text{tn}+\text{fp}}$
false-positive rate (fpr)	$\frac{\text{fp}}{\text{tn}+\text{fp}}$
false-negative rate (fnr)	$\frac{\text{fn}}{\text{tp}+\text{fn}}$
accuracy	$\frac{\text{tp}+\text{tn}}{\text{tp}+\text{tn}+\text{fp}+\text{fn}}$
mean per class accuracy	$\frac{\text{tpr}+\text{tnr}}{2}$
min. per class accuracy	minimum of $\{\text{tpr}, \text{tnr}\}$
specificity	$\frac{\text{tn}}{\text{tn}+\text{fp}}$
precision	$\frac{\text{tp}}{\text{tp}+\text{fp}}$
F_β score, for $\beta = \{1, 0.5, 2\}$	$(1+\beta^2)\frac{\text{precision}\times\text{recall}}{\beta^2(\text{precision}+\text{recall})}$
Matthews correlation coefficient	$\frac{\text{tp}\times\text{tn}-\text{fp}\times\text{fn}}{\sqrt{(\text{tp}+\text{fp})(\text{tp}+\text{fn})(\text{tn}+\text{fp})(\text{tn}+\text{fn})}}$

▷ Example 1: Report threshold-based metrics

Below, we illustrate the use of h2omlestat threshmetric after h2oml gbbinclass.

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 see [H2OML] **H2O setup**.

We use the _h2oframe split command to randomly split the auto frame into a training frame (70% of observations) and a testing frame (30% of observations), which we name train and test, respectively. We also change the current frame to train.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)

Progress (%): 0 100

. _h2oframe split auto, into(train test) split(0.7 0.3) rseed(19)

. _h2oframe change train
```

Next we perform gradient boosting binary classification with default values.

```
. h2oml gbbinclass foreign price mpg weight length, h2orseed(19)

Progress (%): 0 100

Gradient boosting binary classification using H2O

Response: foreign
Loss:     Bernoulli
Frame:                                      Number of observations:
  Training: train                                      Training =     57

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
------------------------+-----------
               Log loss |  .1057473
       Mean class error |     .0125
                    AUC |  .9948529
                  AUCPR |  .9870295
       Gini coefficient |  .9897059
                    MSE |  .0255994
                   RMSE |  .1599981
------------------------------------

. h2omlest store mygbm
```

To report threshold-based metrics, we use the h2omlestat threshmetric command.

```
. h2omlestat threshmetric

Maximum or minimum metrics using H2O
Training frame: train
```

Metric	Max/Min	Threshold
F1	.9714	.6608
F2	.9884	.6608
F0.5	.9551	.6608
Accuracy	.9825	.6608
Precision	1	.9694
Recall	1	.6608
Specificity	1	.9694
Min. class accuracy	.975	.6608
Mean class accuracy	.9875	.6608
True negatives	40	.9694
False negatives	0	.6608 +
True positives	17	.6608
False positives	0	.9694 +
True-negative rate	1	.9694
False-negative rate	0	.6608 +
True-positive rate	1	.6608
False-positive rate	0	.9694 +
MCC	.9596	.6608

```
+ identifies minimum metrics.
```

By default, because we did not use validation or cross-validation, h2omlestat threshmetric reports training results. The reported table has three columns. The first column provides the names of the classification metrics. The second and third columns report the optimal value of each metric (maximum or minimum) and the threshold value that achieves the optimum. The reported optimal value of the metric is the minimum for the false-negative rate, false-positive rate, false negatives, and false positives metrics and is the maximum for all other metrics.

We can use the `thresholds()` option to obtain the reported metrics for a different threshold value or values. For example, to report metrics for a threshold of 0.5, we type

```
. h2omlestat threshmetric, thresholds(0.5)

Metrics for specific threshold using H2O
Training frame: train
```

Threshold	
Input	.5
Computed	.4477
Metric	
F1	.9444
F2	.977
F0.5	.914
Accuracy	.9649
Precision	.8947
Recall	1
Specificity	.95
Min. class accuracy	.95
Mean class accuracy	.975
True negatives	38
False negatives	0
True positives	17
False positives	2
True-negative rate	.95
False-negative rate	0
True-positive rate	1
False-positive rate	.05
MCC	.922

We see that, even though we specified `thresholds(0.5)`, H2O returned results for a threshold of 0.4477, which is the closest available threshold (those found among the stored predicted probabilities).

◁

▷ Example 2: Threshold-based metrics using testing frame

Above, we reported metrics for the training frame. If we wish to report those metrics on the new testing data frame, then we can take one of two approaches.

In the first approach, we specify the `test()` option with the name of our testing frame.

```
. h2omlest restore mygbm
(results mygbm are active now)

. h2omlestat threshmetric, test(test)

Maximum or minimum metrics using H2O
Testing frame: test

--------------------------------------------------
                    Metric |  Max/Min   Threshold
---------------------------+----------------------
                        F1 |    .8333       .4477
                        F2 |    .9259       .4477
                      F0.5 |    .8824       .8916
                  Accuracy |    .8824       .8916
                 Precision |        1       .9694
                    Recall |        1       .4477
               Specificity |        1       .9694
       Min. class accuracy |    .8333       .4477
       Mean class accuracy |    .9167       .4477
            True negatives |       12       .9694
           False negatives |        0       .4477 +
            True positives |        5       .4477
           False positives |        0       .9694 +
        True-negative rate |        1       .9694
       False-negative rate |        0       .4477 +
        True-positive rate |        1       .4477
       False-positive rate |        0       .9694 +
                       MCC |    .7715       .4477
--------------------------------------------------
+ identifies minimum metrics.
```

In the second approach, which we recommend, we use the h2omlpostestframe command to specify test as the default testing frame to be used by this and other postestimation commands.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
. h2omlestat threshmetric
Maximum or minimum metrics using H2O
Testing frame: test
```

Metric	Max/Min	Threshold
F1	.8333	.4477
F2	.9259	.4477
F0.5	.8824	.8916
Accuracy	.8824	.8916
Precision	1	.9694
Recall	1	.4477
Specificity	1	.9694
Min. class accuracy	.8333	.4477
Mean class accuracy	.9167	.4477
True negatives	12	.9694
False negatives	0	.4477 +
True positives	5	.4477
False positives	0	.9694 +
True-negative rate	1	.9694
False-negative rate	0	.4477 +
True-positive rate	1	.4477
False-positive rate	0	.9694 +
MCC	.7715	.4477

+ identifies minimum metrics.

◁

Stored results

h2omlestat threshmetric stores the following in r():

Macros
- r(thresholds) — specified thresholds
- r(thresholds_a) — actual thresholds

Matrices
- r(threshmetric) — classification performance metrics

Also see

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

h2omlexplore — Explore models after grid search

Description | Quick start | Menu | Syntax
Remarks and examples | Stored results | Reference | Also see

Description

`h2omlexplore` allows you to compare models with different hyperparameter configurations after `h2omlestat gridsummary`. In the process of tuning hyperparameters with `h2oml` *gbm* and `h2oml` *rf*, you can use `h2omlestat gridsummary` to report the specified metric for different hyperparameter configurations. `h2omlexplore` allows you to further explore a few selected models by reporting several performance metrics.

Quick start

After performing multiclass classification and obtaining the grid-search summary, view the performance metrics of the models with IDs 2, 4, and 8

```
h2oml rfmulticlass y1 x1-x20, ntrees(10(5)100) maxdepth(3(1)10)
h2omlestat gridsummary
h2omlexplore id = 2 4 8
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlexplore id =` # | *numlist*

where # is a grid ID from `h2omlestat gridsummary` corresponding to a model with the desired hyperparameter configuration, and *numlist* is a list of grid IDs.

Remarks and examples

Building a machine learning model that generalizes well to new data involves choosing an appropriate method and selecting a model by tuning hyperparameters. We can perform a grid search using gradient boosting and random forest methods and then use `h2omlestat gridsummary` to report the hyperparameter configurations that achieve the top performance based on the specified metric. In some cases, you may decide to choose the best-performing model reported in `h2omlestat gridsummary`; in other cases, you may want to explore other well-performing models further, which you can do using `h2omlexplore`. With `h2omlexplore`, you can report several performance metrics for models with different hyperparameter configurations.

▷ Example 1: Exploring different models

In example 1 of [H2OML] **h2omlselect**, we used the social pressure dataset (Gerber, Green, and Larimer 2008) to implement a hyperparameter tuning, and we used the `h2omlselect` command to select the second-best model, which was comparably less complex than the best model. In that example, our decision was based on the area under the precision–recall curve (AUCPR) metric. Suppose now we want to compare those two models based on different performance metrics to make sure that the same pattern holds.

We start by opening the social pressure dataset 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 in an H2O frame, and `_h2oframe change` makes the specified frame the current H2O frame. We use the `_h2oframe split` command to randomly split the `social` frame into a training frame (80% observations) and a validation frame (20% of observations), which we name `train` and `valid`, respectively. We also change the current frame to `train`. For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and see [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 _split social, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe _change train
```

We define a global macro, `predictors`, to store the names of our predictors. We perform random forest binary classification, and we specify the `maxdepth()` and `predsampvalue()` options to tune the maximum tree depth and predictor sampling rate hyperparameters. For illustration, we use the AUCPR metric for tuning.

```
. global predictors gender g2000 g2002 p2000 p2002 p2004 treatment age

. h2oml rfbinclass voted $predictors, validframe(valid) h2orseed(19)
> ntrees(200) maxdepth(3(3)12) predsampvalue(-1, 1(2)8) tune(metric(aucpr))

Progress (%): 0 100

Random forest binary classification using H2O

Response: voted
Frame:                                   Number of observations:
  Training:   train                                Training = 183,607
  Validation: valid                              Validation =  45,854

Tuning information for hyperparameters

Method: Cartesian
Metric: AUCPR

─────────────────────────────────────────────────────────────────────
                      │                  Grid values
      Hyperparameters │        Minimum         Maximum       Selected
──────────────────────┼──────────────────────────────────────────────
      Max. tree depth │              3              12              6
 Pred. sampling value │             -1               7              7
─────────────────────────────────────────────────────────────────────

Model parameters

Number of trees      = 200
              actual = 200
Tree depth:                              Pred. sampling value =       7
           Input max =   6               Sampling rate        =    .632
                 min =   6               No. of bins cat.     =   1,024
                 avg = 6.0               No. of bins root     =   1,024
                 max =   6               No. of bins cont.    =      20
Min. obs. leaf split =   1               Min. split thresh.   =  .00001

Metric summary

──────────────────────────────────────────
               Metric │ Training Validation
──────────────────────┼───────────────────
             Log loss │ .5724664   .5705699
     Mean class error │ .3935492   .3943867
                  AUC │ .6705554   .6734867
                AUCPR │ .4658395   .4725543
     Gini coefficient │ .3411109   .3469735
                  MSE │ .1946923   .1935647
                 RMSE │ .4412395   .4399599
──────────────────────────────────────────
```

Next we obtain the grid-search summary by using the `h2omlestat gridsummary` command. This command lists the configuration of the hyperparameters we are tuning ranked by AUCPR.

```
. h2omlestat gridsummary
Grid summary using H2O
```

ID	Max. tree depth	Pred. sampling value	AUCPR
1	6	7	.4725543
2	6	5	.4723736
3	6	3	.4714554
4	9	3	.4712076
5	6	-1	.4708614
6	12	-1	.4706606
7	9	-1	.4705794
8	9	5	.4689799
9	9	7	.4682457
10	9	1	.4674565

To compare the first two models based on other metrics, we use the `h2omlexplore` command.

```
. h2omlexplore id = 1 2
Performance metric summary using H2O
Training frame  : train
Validation frame: valid
```

	Model index 1	2
Training		
No. of observations	183,607	183,607
Log loss	.5724664	.57237
Mean class error	.3935492	.3979593
AUC	.6705554	.671146
AUCPR	.4658395	.4670326
Gini coefficient	.3411109	.342292
MSE	.1946923	.1946602
RMSE	.4412395	.4412031
Validation		
No. of observations	45,854	45,854
Log loss	.5705699	.5704978
Mean class error	.3943867	.3945857
AUC	.6734867	.6737527
AUCPR	.4725543	.4723736
Gini coefficient	.3469735	.3475054
MSE	.1935647	.1935627
RMSE	.4399599	.4399576

The first section of the output corresponds to the training metrics, while the second presents the validated metrics of the specified models. For each of the metrics, we see that the difference between the best and second-best models is not substantial. Therefore, the decision to switch to the less complex model may be justified.

◁

Stored results

`h2omlexplore` stores the following in `r()`:

Macros

`r(id)`	model IDs

Matrices

`r(table)`	performance metrics for selected models

Reference

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** — Introduction to commands for Stata integration with H2O machine learning

h2omlgof — Compare goodness of fit for machine learning models

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

Description

`h2omlgof` reports goodness of fit after the `h2oml` *rf* and `h2oml` *gbm* commands. This command creates a table with side-by-side performance metrics from selected machine learning methods or models for easy comparison.

Quick start

Goodness of fit for comparing stored estimation results `myrf` and `mygbm`

```
h2omlgof myrf mygbm
```

Goodness-of-fit for comparing all stored estimation results using H2O frame `mynewframe`

```
h2omlgof *, frame(mynewframe)
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlgof` *namelist* [, *options*]

namelist is a name of a stored estimation result, a list of names, `_all`, or `*`. `_all` or `*` requests all stored results. See [H2OML] **h2omlest**.

options	Description
Main	
`title(`*string*`)`	specify the title to be displayed above the table
`train`	specify that performance metrics be reported using training results
`valid`	specify that performance metrics be reported using validation results
`cv`	specify that performance metrics be reported using cross-validation results
`test`	specify that performance metrics be computed using the testing frame
`test(`*framename*`)`	specify that performance metrics be computed using data in testing frame *framename*
`frame(`*framename*`)`	specify that performance metrics be computed using data in H2O frame *framename*
`framelabel(`*string*`)`	label frame as *string* in the output

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

`train`, `valid`, `cv`, `test`, `test()`, `frame()`, and `framelabel()` do not appear in the dialog box.

Options

Main

`title(`*string*`)` specifies the title to be displayed above the table.

The following options are available with `h2omlgof` but are not shown in the dialog box:

`train`, `valid`, `cv`, `test`, `test()`, and `frame()` specify the H2O frame for which performance metrics are reported. Only one of `train`, `valid`, `cv`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that performance metrics be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that performance metrics be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`cv` specifies that performance metrics be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `cv` may be specified only when the `cv` or `cv()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

test specifies that performance metrics be computed on the testing frame specified with h2oml-postestframe. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set with h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that performance metrics be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, h2omlpostestframe provides a more convenient and computationally efficient process for doing this.

frame(*framename*) specifies that performance metrics be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output. This option is not allowed with the cv option.

Remarks and examples

The h2omlgof command provides a concise table of performance metrics for comparing different machine learning methods or models.

After h2oml gbregress and h2oml rfregress, h2omlgof reports the deviance, mean squared error (MSE), root mean squared error (RMSE), root mean squared logarithmic error (RMSLE), mean absolute error (MAE), and R^2. After h2oml gbbinclass and h2oml rfbinclass, it reports log loss, mean of per-class error rates, area under the curve (AUC), area under the precision–recall curve (AUCPR), Gini coefficient, MSE, and RMSE. Finally, after h2oml gbmulticlass and h2oml rfmulticlass, it reports log loss, mean of per-class error rates, MSE, and RMSE. See [H2OML] ***metric_option*** for more information on the reported metrics.

▷ Example 1: Comparing performance in H2OML

In this example, we use h2omlgof to compare results of h2oml *rf* and h2oml *gbm*.

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. We then use the _h2oframe split command to randomly split the auto frame into a training frame (70% of observations), a validation frame (20% of observations), and a testing frame (10% of observations), which we name train, valid, and test, respectively. We also change the current frame to train. 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/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe _put, into(auto)
Progress (%): 0 100
. _h2oframe split auto, into(train valid test) split(0.7 0.2 0.1) rseed(19)
. _h2oframe change train
```

We perform random forest binary classification with default values, and we specify the validation frame in the `validframe()` option. We store the estimation results by using the `h2omlest store` command.

```
. h2oml rfbinclass foreign price length weight, validframe(valid)
> h2orseed(19)
Progress (%): 0 100
Random forest binary classification using H2O
Response: foreign
Frame:                                   Number of observations:
  Training:   train                                 Training =     57
  Validation: valid                               Validation =     10
Model parameters
Number of trees       =  50
              actual =  50
Tree depth:                              Pred. sampling value =     -1
           Input max =  20               Sampling rate        =   .632
                 min =   3               No. of bins cat.     =  1,024
                 avg = 5.7               No. of bins root     =  1,024
                 max =   8               No. of bins cont.    =     20
Min. obs. leaf split =   1               Min. split thresh.   = .00001
Metric summary

                 Metric |  Training  Validation
------------------------+----------------------
               Log loss |  .8466057    .3177202
       Mean class error |     .0625    .1666667
                    AUC |  .9235294    .9047619
                 AUCPR  |  .6822189    .8512376
       Gini coefficient |  .8470588    .8095238
                    MSE |  .0948292      .11421
                   RMSE |  .3079434    .3379497

. h2omlest store RF
```

Next we perform gradient boosting binary classification and store the estimation results.

```
. h2oml gbbinclass foreign price length weight, validframe(valid)
> h2orseed(19)
Progress (%): 0 100
Gradient boosting binary classification using H2O
Response: foreign
Loss:     Bernoulli
Frame:                                   Number of observations:
  Training:   train                                Training =     57
  Validation: valid                              Validation =     10
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   Validation
───────────────────┼─────────────────────────
          Log loss │   .1072901     .2774807
  Mean class error │      .0125     .0714286
               AUC │   .9955882      .952381
             AUCPR │   .9889171      .904106
  Gini coefficient │   .9911765     .9047619
               MSE │   .0261993     .1002502
              RMSE │    .161862     .3166232
─────────────────────────────────────────────

. h2omlest store GBM
```

To compare random forest (RF) and gradient boosting machine (GBM) models, we type

```
. h2omlgof RF GBM

Performance metrics for model comparison using H2O
Training frame:   train
Validation frame: valid

────────────────────────────────────────────────────
                      │           RF           GBM
──────────────────────┼─────────────────────────────
Training              │
  No. of observations │           57            57
             Log loss │     .8466057      .1072901
     Mean class error │        .0625         .0125
                  AUC │     .9235294      .9955882
                AUCPR │     .6822189      .9889171
     Gini coefficient │     .8470588      .9911765
                  MSE │     .0948292      .0261993
                 RMSE │     .3079434       .161862
──────────────────────┼─────────────────────────────
Validation            │
  No. of observations │           10            10
             Log loss │     .3177202      .2774807
     Mean class error │     .1666667      .0714286
                  AUC │     .9047619       .952381
                AUCPR │     .8512376       .904106
     Gini coefficient │     .8095238      .9047619
                  MSE │       .11421      .1002502
                 RMSE │     .3379497      .3166232
────────────────────────────────────────────────────
```

In the output, the first section reports training results, and the second section reports validation results. Looking at the validation results, we see that the GBM method outperforms the RF method. The log loss, mean of per-class error rates, MSE, and RMSE are all smaller for GBM, while AUC, AUCPR, and the Gini coefficient are larger for GBM, all of which indicate better performance.

◁

▷ Example 2: Comparing performance in H2OML on a new frame

In example 1, we compared the performance of two methods on the validation frame. If we instead wish to compare methods on a new data frame, we can take one of two approaches. In the first, we specify the frame in the `frame()` option or, if it is a testing frame, in the `test()` option.

```
. h2omlgof RF GBM, test(test)

Performance metrics for model comparison using H2O
Testing frame: test
```

	RF	GBM
Testing		
No. of observations	7	7
Log loss	.236301	.1155489
Mean class error	0	0
AUC	1	1
AUCPR	1	1
Gini coefficient	1	1
MSE	.0878302	.0364771
RMSE	.2963615	.1909897

In the second approach, which we recommend, we use the `h2omlpostestframe` command to specify the postestimation frame to be used by this and other postestimation commands. With this approach, the new frame must be set for each set of estimation results. Thus, we first need to restore each set of estimates by using the `h2omlest restore` command. For the GBM results, we type

```
. h2omlest restore GBM
(results GBM are active now)

. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

Similarly, for the RF results, we type

```
. h2omlest restore RF
(results RF are active now)

. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

Finally, we compare the testing results by using the `h2omlgof` command.

```
. h2omlgof RF GBM

Performance metrics for model comparison using H2O
Testing frame: test
```

	RF	GBM
Testing		
No. of observations	7	7
Log loss	.236301	.1155489
Mean class error	0	0
AUC	1	1
AUCPR	1	1
Gini coefficient	1	1
MSE	.0878302	.0364771
RMSE	.2963615	.1909897

Here GBM again outperforms RF for most of the performance metrics.

Stored results

h2omlgof stores the following in r():

Macros
- r(names) — names of estimation results displayed

Matrices
- r(table) — matrix containing the values displayed

Also see

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

[H2OML] **h2omlestat metrics** — Display performance metrics

h2omlgraph ice — Produce individual conditional expectation plot

Description | Quick start | Menu | Syntax
Options | Remarks and examples | References | Also see

Description

`h2omlgraph ice` plots the individual conditional expectation (ICE) curves after `h2oml` *gbm* and `h2oml` *rf*. For regression, the ICE values correspond to predictions for an individual observation as values of a predictor of interest vary. For classification, the ICE values correspond to the predicted probabilities for an individual observation as values of a predictor of interest vary. Rather than plotting the ICE curve for every observation, `h2omlgraph ice` plots ICE curves at the boundaries of the deciles of the predictor of interest. The graph produced by `h2omlgraph ice` is useful for evaluating the partial effect of a predictor on the response and how that effect differs across deciles of the predictor. It is also useful for determining whether interaction effects exist between the variable of interest and other predictors.

The ICE plots are similar to the partial density plot (PDP), but the PDP estimates the average predictions for the entire dataset and can be considered as the average of the ICE curves for all observations.

Quick start

Plot the ICE for predictor `x1`

```
h2omlgraph ice x1
```

Same as above, but do not show histogram in the plot

```
h2omlgraph ice x1, nohistogram
```

Plot the ICE after the multiclass classification for the class `no` and using H2O frame `myframe`

```
h2omlgraph ice x1, target(no) frame(myframe)
```

Menu

Statistics > H2O machine learning

Syntax

h2omlgraph ice *predictor* [, *options*]

options	Description
Main	
* target(*class*)	specify the target class of the response after multiclass classification
maxlevels(#)	specify the maximum number of levels for categorical predictors; default is maxlevels(30)
savedata(*filename*[, replace])	save plot data to *filename*
Plot options	
nohistogram	do not plot histogram of the predictor
histopts(*bar_opts*)	affect rendition of the histogram
line#opts(*line_options*)	affect rendition of the ICE curve for quantile #
nopdline	do not plot partial dependence curve
pdlineopts(*line_options*)	affect rendition of partial dependence curve
twoway_options	any options other than by() documented in [G-3] ***twoway_options***
train	specify that the ICE be reported using training results
valid	specify that the ICE be reported using validation results
test	specify that the ICE be computed using testing frame
test(*framename*)	specify that the ICE be computed using data in testing frame *framename*
frame(*framename*)	specify that the ICE be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

*target() is required after multiclass classification.

train, valid, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

target(*class*) specifies for which class of the response variable the ICE should be plotted. target() is required after multiclass classification with h2oml gbmulticlass or h2oml rfmulticlass.

maxlevels(#) specifies the maximum number of levels of the specified categorical predictor to be included in the ICE estimation. The default is maxlevels(30).

savedata(*filename*[, replace]) saves the plot data to a Stata data file (.dta file). replace specifies that *filename* be overwritten if it exists.

Plot options

nohistogram removes the histogram of the predictor. By default, the histogram is included.

histopts(*bar_opts*) affects rendition of the histogram; see [G-2] **graph twoway bar**.

line#opts(*line_options*) affects the rendition of the ICE curve for decile #. See [G-3] ***line_options***.

`nopdline` removes the line for the partial dependence curve. The partial dependence curve is included by default.

`pdlineopts(`*line_options*`)` affects rendition of the partial dependence curve; see [G-3] ***line_options***.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding `by()`. These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with `h2omlgraph ice` but are not shown in the dialog box:

`train`, `valid`, `test`, `test()`, and `frame()` specify the H2O frame for which ICE is reported. Only one of `train`, `valid`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that ICE be reported using training results. This is the default when validation is not performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that ICE be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that ICE be computed on the testing frame specified with `h2omlpostestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set by using `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that ICE be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, it is more computationally efficient and convenient to specify the testing frame by using `h2omlpostestframe` instead of specifying `test(`*framename*`)` with individual postestimation commands.

`frame(`*framename*`)` specifies that ICE be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output.

Remarks and examples

We assume you have read the *Interpretation and explanation* in [H2OML] **Intro**.

Remarks are presented under the following headings:

Introduction
Examples of ICE curves

Introduction

The PDP, introduced in [H2OML] **h2omlgraph pdp**, graphs the average predictions across the values of a predictor of interest and is useful for understanding the average or partial effect of the predictor on the response. However, when there is an interaction effect among predictors, the PDP cannot fully capture the effect. In fact, there may be no average effect shown by a flat curve in the PDP, while there

are substantial effects at various levels of the predictor, but the effects are in opposite directions and cancel each other out when averaged in the PDP. The ICE plots improve upon the PDPs by visualizing the relationship between the response and the predictor for individual observations (Goldstein et al. 2015).

Formally, let $f(\mathbf{X}_S, \mathbf{X}_C)$ be our machine learning model, $\mathbf{X}_S$ be the predictor whose effect we wish to study, and $\mathbf{X}_C$ be all other predictors in our model.

To obtain ICE values for all observations $i = 1, 2, \ldots, n$, the values of predictors $\mathbf{X}_C$ are fixed to their observed values of $\mathbf{x}_{Ci}$. Then the values of $\mathbf{X}_S$ are iteratively set to the observed value $\mathbf{x}_{Sj}$ for observations $j = 1, 2, \ldots, n$ to obtain predictions $\widehat{f}(\mathbf{x}_{Sj}, \mathbf{x}_{Ci})$. Thus, for each observation i in the dataset, we obtain n predicted values. These correspond to predictions where $\mathbf{X}_S$ is set to its observed value in observations $j = 1, \ldots, n$, while the remaining predictors $\mathbf{X}_C$ are held at their observed values for the same observation.

The ICE curve for observation i plots the resulting predicted values on the y axis and the predictor of interest $\mathbf{X}_S$ on the x axis. In practice, if the number of observations n is large, displaying a graph with curves for each observation becomes difficult to read. Therefore, it is recommended to consider using only deciles or quantiles of the data. `h2omlgraph ice` plots ICE curves for deciles of the predictor of interest. By default, it also plots the partial dependence curve for comparison with the ICE curves.

Examples of ICE curves

In this section, we demonstrate the advantage of `h2omlgraph ice` when an interaction effect is present among predictors. As with most explainable machine learning methods, caution is advised when using those results for decision making. For examples where explainable machine learning methods fail, see example 2 of [H2OML] **h2omlgraph varimp**, Krishna et al. (2022), Lakkaraju and Bastani (2020), and Slack et al. (2020).

The examples are presented under the following headings:

Example 1: Capturing an interaction effect through ICE
Example 2: Finding regions of interactions
Example 3: ICE plot for multinomial classification

▷ Example 1: Capturing an interaction effect through ICE

This example is borrowed from Goldstein et al. (2015). We consider the following data-generation process with an interaction: $Y = 0.2X_1 + 5X_2 + \varepsilon$ if $X_3 \geq 0$ and $Y = 0.2X_1 - 5X_2 + \varepsilon$ otherwise. Here $X_1, X_2, X_3 \sim U(-1, 1)$ and $\varepsilon \sim N(0, 1)$.

We start by opening the simulated `interaction.dta` dataset in Stata and then putting it 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/interaction
(Fictional interaction data)
. h2o init
  (output omitted)
. _h2oframe put, into(interaction)
Progress (%): 0 100
. _h2oframe change interaction
```

For illustration purposes, we use h2oml rfregress to perform random forest regression with default values for hyperparameters. We then store the estimation results by using the h2omlest store command.

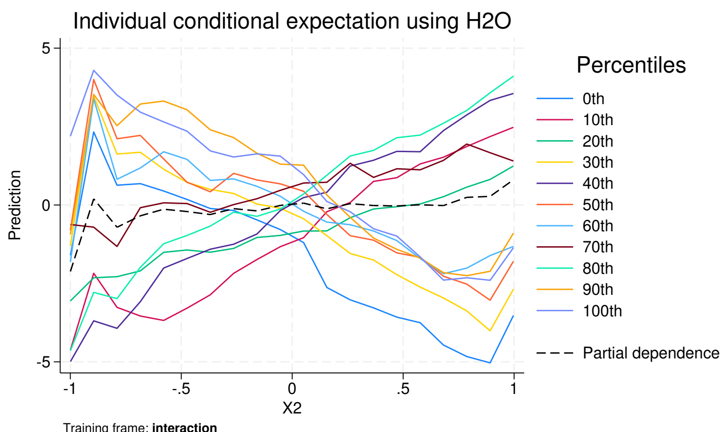
```
. h2oml rfregress Y X1 X2 X3, h2orseed(19)
Progress (%): 0 100
Random forest regression using H2O
Response: Y
Frame:                                     Number of observations:
  Training: interaction                               Training =    500
Model parameters
Number of trees        =   50
                actual =   50
Tree depth:                                Pred. sampling value =     -1
             Input max =   20              Sampling rate        =   .632
                   min =   16              No. of bins cat.     =  1,024
                   avg = 18.8              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
Deviance	2.876126
MSE	2.876126
RMSE	1.695915
RMSLE	.
MAE	1.29916
R-squared	.6973235

```
. h2omlest store rf_inter
```

Next we plot ICE curves for X2 by using the h2omlgraph ice command.

```
. h2omlgraph ice X2
```



Here the dashed black line represents the partial dependence, and the other 11 lines correspond to ICE computed at the boundaries of the deciles X2—the 0th, 10th, ..., 100th percentiles of the observed values of X2 in the dataset. The partial dependence suggests no partial effect of X2 on the response, because the

curve is mostly flat over the range of X2 values. This aggregate effect close to zero is actually the result of the individual effects canceling each other out. Some of them are positive (the ICE lines that increase with X2), and some of them negative (the ICE lines that decrease with X2).

In contrast to the PDP, the ICE curves provide a more comprehensive representation of the relationship between X2 and the response. Moreover, an interaction effect can be inferred from the ICE plots, because depending on the region of the X2 predictor space, ICE is either increasing or decreasing.

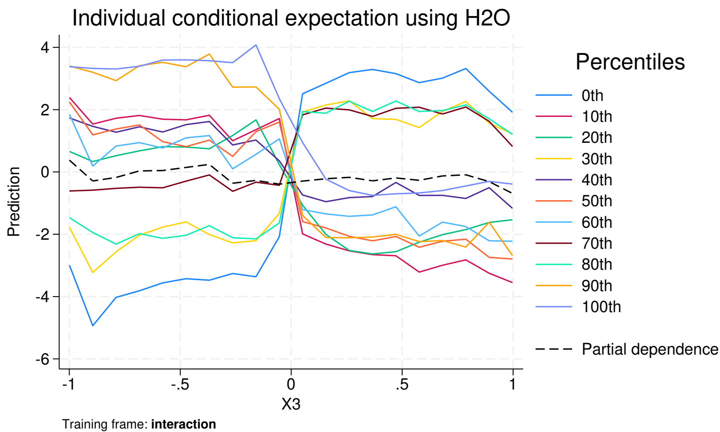
◁

▷ Example 2: Finding regions of interactions

In example 1, we showed that the ICE plots suggest some interaction effects among predictors. In this example, we are interested in detecting the regions where those interactions occur. For details, see Goldstein et al. (2015, sect. 4.2).

We now visualize ICE plots for the predictor X3.

```
. h2omlgraph ice X3
```



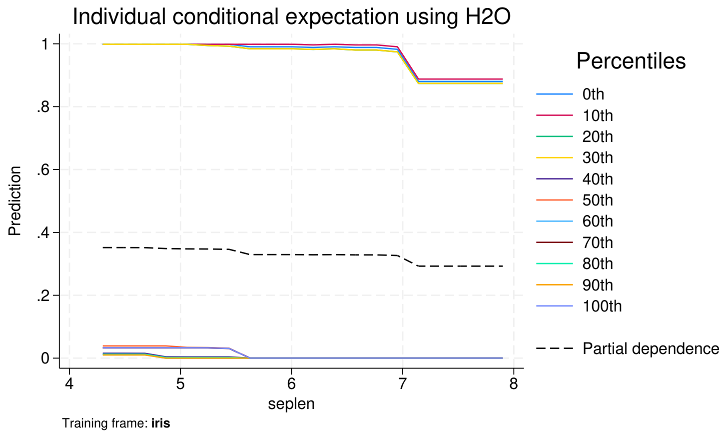
As in example 1, PDP suggests no effect of X3 on the response. However, the nonparallel ICE curves show the effect of X3 changes for each of the plotted percentiles near the neighborhood of $X3 = 0$. This indicates an interaction of X3 with another variable at this point, and we know this to be true based on the data-generating process for our simulated data.

◁

▷ Example 3: ICE plot for multinomial classification

In example 5 of [H2OML] **h2omlgraph pdp**, we showed how to implement and interpret PDP after multiclass classification. In this example, we continue from example 5 and plot ICE curves. Note that, compared with h2omlgraph pdp, the target() option of h2omlgraph ice supports only one class of the response variable. Here we plot ICE for the Setosa class in iris.

```
. h2omlgraph ice seplen, target(Setosa)
```



For observations below the 50th percentile of seplen, the probability of predicting Setosa is around 1 when seplen < 7 and goes down afterward. For observations in the higher percentiles of seplen, the probability of predicting Setosa is close to 0. PDP, the dashed black line, is an average of ICE curves for all observations.

◁

References

Goldstein, A., A. Kapelner, J. Bleich, and E. Pitkin. 2015. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics* 24: 44–65. https://doi.org/10.1080/10618600.2014.907095.

Krishna, S., T. Han, A. Gu, S. Wu, S. Jabbari, and H. Lakkaraju. 2022. The disagreement problem in explainable machine learning: A practitioner's perspective. arXiv:2202.01602 [cs.LG], https://doi.org/10.48550/arXiv.2202.01602.

Lakkaraju, H., and O. Bastani. 2020. ""How do I fool you?": Manipulating user trust via misleading black box explanations". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 79–85. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375833.

Slack, D., S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju. 2020. "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 180–186. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375830.

Also see

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

[H2OML] **h2omlgraph pdp** — Produce partial dependence plot

h2omlgraph pdp — Produce partial dependence plot

Description Quick start Menu Syntax
Options Remarks and examples References Also see

Description

h2ograph pdp produces the partial dependence plot (PDP) after h2oml *gbm* and h2oml *rf*. For regression, the PDP graphs the average prediction versus the values of a predictor of interest. For classification, PDP graphs average predicted probabilities versus values of a predictor of interest. Thus, PDP graphically depicts the average or partial effect of predictors on the response.

Quick start

Plot the PDP for the predictor x1

```
h2omlgraph pdp x1
```

Same as above, but plot for x1, x2, and x3, and combine the plots

```
h2omlgraph pdp x1 x2 x3, combine
```

Same as above, but show the standard deviations of the average response, and do not show the histogram

```
h2omlgraph pdp x1 x2 x3, combine sd nohistogram
```

Create a contour plot of the joint PDP for x1 and x2

```
h2omlgraph pdp x1 x2, pair
```

Menu

Statistics > H2O machine learning

Syntax

h2omlgraph pdp *predictors* [, *options*]

options	Description
Main	
* target(*classes*)	specify the target class(es) of the response variable for multiclass classification
obs(#)	specify the observation number for computing partial dependence
savedata(*filename* [, replace])	save plot data to *filename*
Plot options	
pair	create a contour plot of the joint marginal predictions
pairopts(*contour_options*)	affect rendition of PDP contour plot
lineopts(*line_options*)	affect rendition of PDP line
line#opts(*line_options*)	affect rendition of PDP line for target class #
sd	display standard deviation band with PDP
sdopts(*area_options*)	affect rendition of the standard deviation band
combine	combine multiple PDP graphs
combineopts(*comb_opts*)	affect rendition of the combined graphs
nohistogram	do not plot histogram of the predictor
histopts(*bar_opts*)	affect rendition of the histogram
Y axis, X axis, Titles, Legend, Overall	
name(*namespec* [, replace])	specify names of graphs
saving(*filespec* [, replace])	save graphs in files
twoway_options	any options other than by() documented in [G-3] ***twoway_options***
train	specify that the partial dependence be reported using training results
valid	specify that the partial dependence be reported using validation results
test	specify that the partial dependence be computed using testing frame
test(*framename*)	specify that the partial dependence be computed using data in testing frame *framename*
frame(*framename*)	specify that the partial dependence be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

*target() is required after multiclass classification.

train, valid, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

target(*classes*) specifies for which class or classes of the response variable the partial dependence should be plotted. target() is required after multiclass classification with h2oml gbmulticlass or h2oml rfmulticlass. target() is not allowed with pair.

obs(#) specifies the observation number for which partial dependence will be computed. The specified value should be a positive integer. If obs() is specified, the individual conditional expectation for obs(#) is computed; see [H2OML] **h2omlgraph ice**. obs() is not allowed with sd.

savedata(*filename*[, replace]) saves the plot data to a Stata data file (.dta file). replace specifies that *filename* be overwritten if it exists.

Plot options

pair specifies to create the contour plot of the joint marginal predictions of predictors. This option is valid only if two or more predictors are specified. pair is not allowed with any of sd, target(), lineopts(), histopts(), or line#opts().

pairopts(*contour_options*) affects the rendition of the contour plot. See [G-2] **graph twoway contour**.

lineopts(*line_options*) affects the rendition of the PDP line. See [G-3] ***line_options***. lineopts() is not allowed with pair.

line#opts(*line_options*) affects the rendition of the PDP line for the target class #. See [G-3] ***line_options***. line#opts() is valid only if target() is specified. line#opts() is not allowed with pair.

sd specifies to plot a standard deviation band. For each observed value of the specified predictor, PDP estimates the mean response, and the standard deviation is estimated using those responses. sd is not allowed with pair or obs().

sdopts(*area_options*) affects the rendition of the standard deviation band. See [G-3] ***area_options***.

combine specifies to combine the graphs of PDP for individual predictors when more than one predictor is specified.

combineopts(*comb_opts*) affects the rendition of the combined graphs. See [G-2] **graph combine**.

nohistogram removes the histogram of the predictor from the PDP. By default, the histogram is included.

histopts(*bar_opts*) affects the rendition of the histogram; see [G-2] **graph twoway bar**. histopts() is not allowed with pair.

Y axis, X axis, Titles, Legend, Overall

name(*namespec*[, replace]) specifies the name of the graph or multiple graphs. See [G-3] ***name_option*** for a single graph. If multiple graphs are produced, then the argument of name() is either a list of names or a *stub*, in which case graphs are named *stub*1, *stub*2, and so on. With multiple graphs, if name() is not specified and neither sleep() nor wait is specified, then name(Graph__#, replace) is assumed.

replace specifies to replace existing graphs with the specified name or names.

saving(*filespec*[, replace]) specifies the filename or filenames to use to save the graph or multiple graphs to disk. See [G-3] ***saving_option*** for a single graph. If multiple graphs are produced, then the argument of saving() is either a list of filenames or a *stub*, in which case graphs are saved with filenames *stub*1, *stub*2, and so on.

replace specifies to replace existing graphs with the specified name or names.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding by(). These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with h2omlgraph pdp but are not shown in the dialog box:

train, valid, test, test(), and frame() specify the H2O frame for which partial dependencies are reported. Only one of train, valid, test, test(), or frame() is allowed.

train specifies that partial dependencies be reported using training results. This is the default when validation is not performed during estimation and when a postestimation frame has not been set with h2omlpostestframe.

valid specifies that partial dependencies be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. valid may be specified only when the validframe() option is specified with h2oml *gbm* or h2oml *rf*.

test specifies that partial dependencies be computed on the testing frame specified with h2omlpostestframe. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set by using h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that partial dependencies be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, it is more computationally efficient and convenient to specify the testing frame by using h2omlpostestframe instead of specifying test(*framename*) with individual postestimation commands.

frame(*framename*) specifies that partial dependencies be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output.

Remarks and examples

We assume you have read the introduction to explainable machine learning in [H2OML] **Intro**.

Remarks are presented under the following headings:

Introduction
Examples of using PDP

Introduction

The partial dependence plot (PDP) is an intuitive tool to study the marginal effect of predictors on the response (Friedman 2001). The PDP allows you to easily visualize how the expected response changes across different values of a predictor. For regression, the PDP graphs the average prediction versus the values of a predictor of interest. For classification, the PDP graphs the average of the predicted probabilities versus the values of a predictor of interest.

In fact, to study the average predictions (or predictive margins) for a single predictor in regression or binary classification, the PDP is analogous to the plot of predictive margins we can obtain from marginsplot in Stata after fitting a model with regress or logit, respectively.

Formally, let $f(\mathbf{X}_S, \mathbf{X}_C)$ be our machine learning model, $\mathbf{X}_S$ be the predictors whose effect we wish to study, and $\mathbf{X}_C$ be all other predictors in our model. For $\mathbf{X}_S$ fixed at $\mathbf{x}_S$, the partial dependence is defined as

$$f_S(\mathbf{x}_S) = E_{\mathbf{X}_C}\{f(\mathbf{x}_S, \mathbf{X}_C)\} = \int f(\mathbf{x}_S, \mathbf{x}_C) dP(\mathbf{x}_C)$$

In words, partial dependence is an average (over the marginal distribution of $\mathbf{X}_C$) of the predictions our model makes when we fix $\mathbf{X}_S$ at some value $\mathbf{x}_S$. In the `h2omlgraph pdp` syntax, $\mathbf{X}_S$ corresponds to the input *predictors*. In a finite sample, for the jth observation, partial dependence is computed by averaging predictions computed at the observed values of predictors $\mathbf{x}_{C_i}$ for $i = 1, \dots, n$.

$$\widehat{f}_S(\mathbf{x}_{Sj}) = \frac{1}{n}\sum_{i=1}^{n} \widehat{f}(\mathbf{x}_{Sj}, \mathbf{x}_{C_i})$$

The PDP is a plot of such average predictions over the support of $\mathbf{X}_S$, which allows us to investigate how average predicted values of the response (in regression) or average predicted probabilities (in classification) vary over the support of the predictors of interest.

In practice, PDP works well when the dependence between $\mathbf{X}_S$ and $\mathbf{X}_C$ is not strong. When the dependence is strong or the true model includes interactions, PDP is not reliable and the individual conditional expectation curve is recommended for postestimation analysis of partial effects.

Examples of using PDP

In this section, we demonstrate some uses of the `h2omlgraph pdp` command. The examples are presented under the following headings.

Example 1: PDP interpretation for regression
Example 2: Caution on PDP causal interpretation
Example 3: PDP with a monotonicity constraint
Example 4: Joint marginal predictions through PDP
Example 5: PDP interpretation for multiclass classification

▷ Example 1: PDP interpretation for regression

In this example, we plot and interpret the PDP for a random forest regression model.

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 see [H2OML] **H2O setup**.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

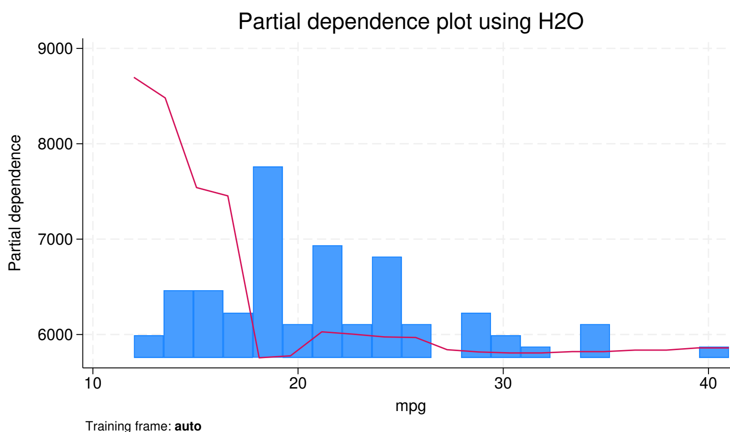
For simplicity, we save the predictor names in the global macro `predictors` in Stata. We then perform random forest regression with 100 trees and a maximum depth of 5.

```
. global predictors mpg trunk weight length
. h2oml rfregress price $predictors, h2orseed(19) ntrees(100) maxdepth(5)
Progress (%): 0 100
Random forest regression using H2O
Response: price
Frame:                                        Number of observations:
  Training: auto                                         Training =     74
Model parameters
Number of trees      = 100
              actual = 100
Tree depth:                                   Pred. sampling value =     -1
           Input max =   5                    Sampling rate        =   .632
                 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 =   1                    Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Deviance	3760463
MSE	3760463
RMSE	1939.191
RMSLE	.2626369
MAE	1361.947
R-squared	.5618179

Finally, we use the `h2omlgraph pdp` command to show how the average predicted `price` changes across levels of the predictor `mpg`.

```
. h2omlgraph pdp mpg
```



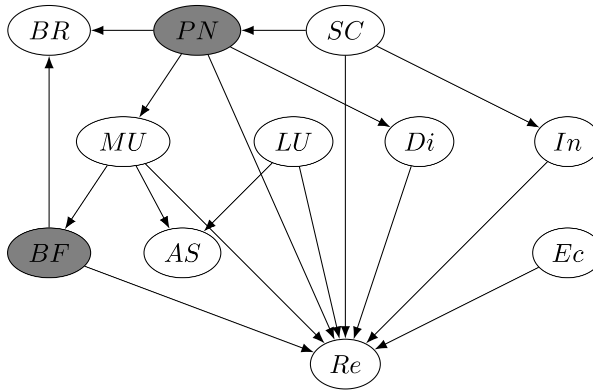
From the plot, we can see that the predicted price tends to decrease as the value of mpg increases. We also see a histogram of mpg, showing that only a few observations have mpg values over 30.

◁

▷ Example 2: Caution on PDP causal interpretation

In this example, we explore why it is important to exercise caution when using and interpreting machine learning explanation methods such as PDPs. See also example 2 of [H2OML] **h2omlgraph varimp** and examples in Krishna et al. (2022), Lakkaraju and Bastani (2020), and Slack et al. (2020).

The data-generating process and the discussion closely follow Lundberg (2021). Our goal is to understand how various predictors affect a subscriber's decision to renew their contract with a company, which is a causal question. We assume that our data are generated from the following causal directed acyclic graph (DAG).



See [CAUSAL] **Intro** for an introduction to DAGs. Here the abbreviations in the nodes correspond to the following predictors: MU is customer monthly usage, BF is the number of bugs faced, PN is product need, SC is the number of sales calls, Di is the customer discount, Ec is other macroeconomic activities, AS is the ad spending amount, LU is the last upgrade, Re is whether the customer renewed the contract, In is the number of interactions with a customer, and BR is bugs reported by a customer. The response is Re, whether the customer renewed the contract. The gray nodes represent unobserved confounders.

An important assumption to causally interpret PDP is that the model needs to satisfy the backdoor or unconfoundedness assumption (Zhao and Hastie 2021). In short, to identify the causal effect of one of these predictors on the response renewal, all other paths between the predictor and renewal must be blocked. Blocking the alternative paths involves "controlling for" or "conditioning on" a specific set of predictors. For definitions, see Pearl (2009) and Imbens and Rubin (2015).

We start by opening the retention.dta dataset in Stata and then putting it into an H2O frame.

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

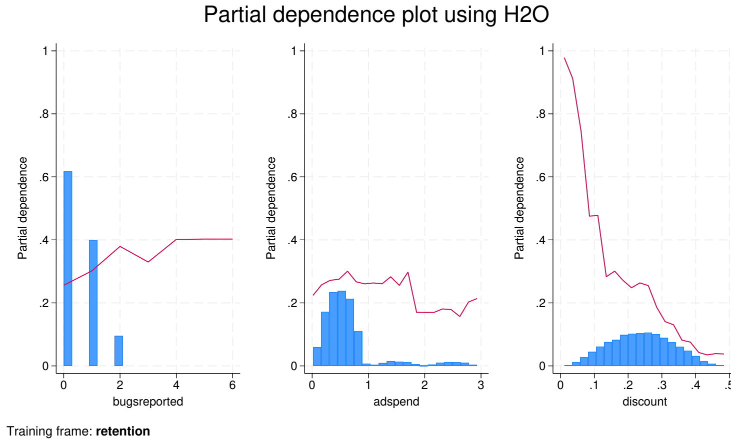
For convenience, we create a global macro predictors in Stata to store the names of the observed predictors. We then perform gradient boosting binary classification using these observed predictors.

```
. global predictors_obs salescalls interactions economy lastupgrade
> discount monthlyusage adspend bugsreported

. h2oml gbbinclass renew $predictors_obs, h2orseed(19) lrate(0.1)
> maxdepth(15) ntrees(300)

Progress (%): 0 9.6 23.0 36.3 47.9 71.3 95.3 100

Gradient boosting binary classification using H2O

Response: renew
Loss:     Bernoulli
Frame:                                      Number of observations:
  Training: retention                                  Training = 10,000

Model parameters

Number of trees      =  300             Learning rate       =     .1
              actual =  300             Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =   15             Sampling rate       =      1
                 min =   15             No. of bins cat.    =  1,024
                 avg = 15.0             No. of bins root    =  1,024
                 max =   15             No. of bins cont.   =     20
Min. obs. leaf split =   10             Min. split thresh.  = .00001

Metric summary
```

Metric	Training
Log loss	.007453
Mean class error	0
AUC	1
AUCPR	1
Gini coefficient	1
MSE	.0000988
RMSE	.0099407

Next we use h2omlgraph pdp to plot the partial dependence for the predictors bugsreported, adspend, and discount. To combine the plots, we specify the combine option. We also specify the combineopts() option with the cols(3) suboption to request three columns, and we give the y axis a common scale by specifying the ycommon suboption.

```
. h2omlgraph pdp bugsreported adspend discount, combine
> combineopts(cols(3) ycommon)
```



The figure suggests counterintuitive results. Specifically, as the number of bugs reported increases, the probability of retention also increases, and as the discount increases, the probability of retention decreases.

A closer look at a causal DAG sheds more light on the source of these counterintuitive results. The bugsreported (BR) predictor is a collider (for definitions, see *Causal diagrams* in [CAUSAL] **Intro**), and by conditioning on a collider, we open a path between its parents, BF and PN, which are unobserved. This leads to an incorrect positive effect for BR, when there is no true effect. Similarly, conditioning on the predictor adspend (AS), we introduce a collider bias. Finally, the effect of discount (Di) suffers from the unobserved confounders. In causal DAG language, because PN and BF are unobserved, there are open backdoor paths between Di and Re.

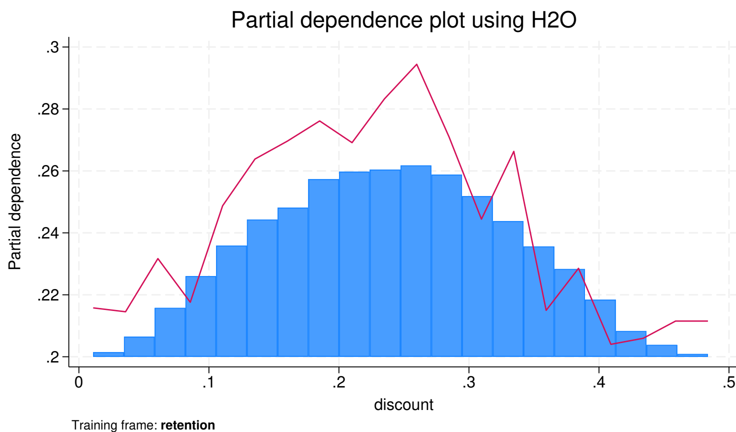
These results highlight the fundamental difference between prediction and causal inference. The same predictors can be good for predicting an outcome but may not be useful for causal inference. For details and more discussion, see Cinelli, Forney, and Pearl (2024).

Because the dataset is artificial, we can demonstrate the effect of controlling unobserved confounders on the average predicted probabilities. We now control for the number of bugs faced and product needed, and we omit BR and AS from our model. The new set of predictors is saved in the global macro `predictors` in Stata.

```
. global predictors salescalls interactions economy lastupgrade
> discount monthlyusage bugsfaced productneed

. h2oml gbbinclass renew $predictors, h2orseed(19) lrate(0.1)
> maxdepth(15) ntrees(300)

Progress (%): 0 9.0 19.3 31.3 42.6 67.0 92.0 100

Gradient boosting binary classification using H2O

Response: renew
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training: retention                              Training = 10,000

Model parameters

Number of trees       =   300           Learning rate       =      .1
              actual =   300            Learning rate decay =       1
Tree depth:                             Pred. sampling rate =       1
           Input max =    15            Sampling rate       =       1
                 min =    15            No. of bins cat.    =   1,024
                 avg =  15.0            No. of bins root    =   1,024
                 max =    15            No. of bins cont.   =      20
Min. obs. leaf split =    10            Min. split thresh.  = .00001

Metric summary
```

Metric	Training
Log loss	.0022039
Mean class error	0
AUC	1
AUCPR	1
Gini coefficient	1
MSE	9.28e-06
RMSE	.0030459

```
. h2omlgraph pdp discount

Progress (%): 0 100
```



We can see that the interpretation of Di changed substantially. The partial dependence first grows with the discount, but then clearly decreases for discounts greater than 0.25.

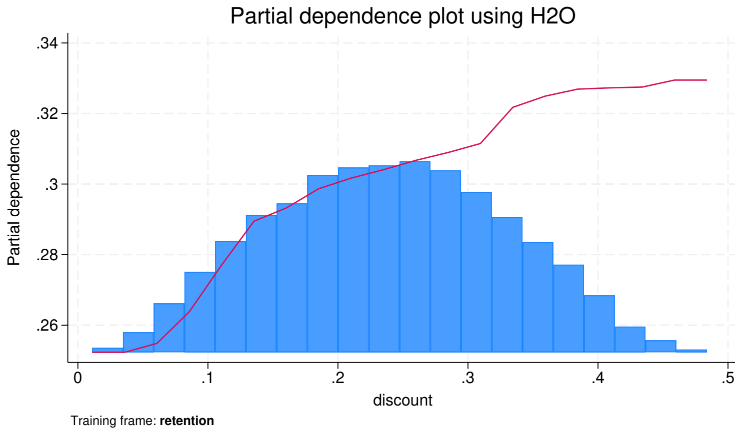
◁

▷ Example 3: PDP with a monotonicity constraint

In some applications, it is reasonable to assume that the response is a monotone function of the predictor. For details, see [H2OML] **Intro**. In this example, we continue with example 2 and show a PDP after enforcing monotonicity constraints. Suppose we strongly believe that the effect of the predictor `discount` should be monotonic increasing. This information can be directly imposed on the gradient boosting machine model by using the `monotone()` option.

```
. h2oml gbbinclass renew $predictors, h2orseed(19) lrate(0.1)
> maxdepth(15) ntrees(300) monotone(discount)
Progress (%): 0 5.9 15.9 26.3 36.6 52.3 70.3 88.3 100

Gradient boosting binary classification using H2O

Response: renew
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training: retention                              Training = 10,000

Model parameters

Number of trees       =  300            Learning rate       =     .1
              actual  =  300            Learning rate decay =      1
Tree depth:                             Pred. sampling rate =      1
           Input max =    15            Sampling rate       =      1
                 min =    15            No. of bins cat.    =  1,024
                 avg =  15.0            No. of bins root    =  1,024
                 max =    15            No. of bins cont.   =     20
Min. obs. leaf split =    10            Min. split thresh.  = .00001

Metric summary

             Metric   Training

           Log loss   .0050499
   Mean class error          0
                AUC          1
              AUCPR          1
   Gini coefficient          1
                MSE   .0000516
               RMSE   .0071842

Monotone increasing: discount
```

```
. h2omlgraph pdp discount
Progress (%): 0 100
```



Compared with the PDP in example 2, the partial dependence of the predictor `discount` is monotonically increasing as the size of the discount increases.

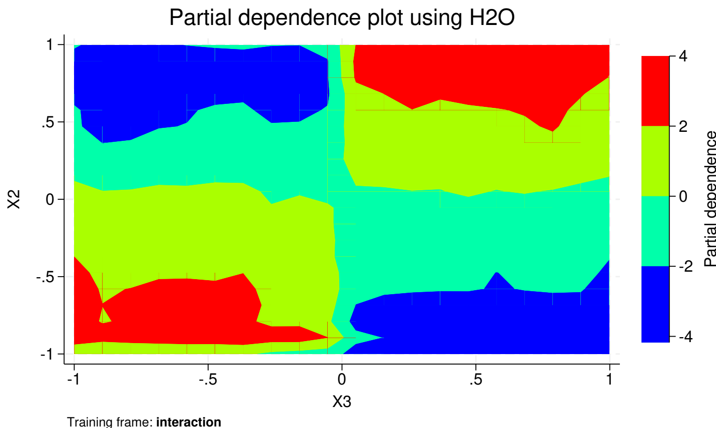
◁

▷ Example 4: Joint marginal predictions through PDP

In example 2 of [H2OML] **h2omlgraph ice**, we show that partial dependence curves are not useful for capturing an interaction effect and instead suggest to use ICE curves. In this example, we show how we might mitigate this issue by plotting the joint partial effect.

We start by restoring the `rf_inter` model by using the `h2omlest restore` command. The model was stored in example 1 of [H2OML] **h2omlgraph ice**.

```
. h2omlest restore rf_inter
(results rf_inter are active now)
. h2omlgraph pdp X2 X3, pair
```



We can see that the contour plot of the joint effect clearly captures the interaction, with the largest predictions in the regions $X_3 < 0$, $X_2 < -0.5$ and $X_3 > 0$, $X_2 > 0.5$.

◁

▷ Example 5: PDP interpretation for multiclass classification

In this example, we consider the 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 will demonstrate how to interpret the PDP for multiclass classification. For illustration purposes, we use random forest multiclass classification with 500 trees.

```
. use https://www.stata-press.com/data/r19/iris
(Iris data)
. h2o init
  (output omitted)
. _h2oframe put, into(iris)
Progress (%): 0 100
. _h2oframe change iris
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, h2orseed(19) ntrees(500)
Progress (%): 0 31.9 82.9 100
Random forest multiclass classification using H2O
Response: iris                         Number of classes     =      3
Frame:                                 Number of observations:
  Training: iris                                    Training =    150
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 = 3.7            No. of bins root      =  1,024
                  max =   9            No. of bins cont.     =     20
Min. obs. leaf split =   1            Min. split thresh.    = .00001
Metric summary
```

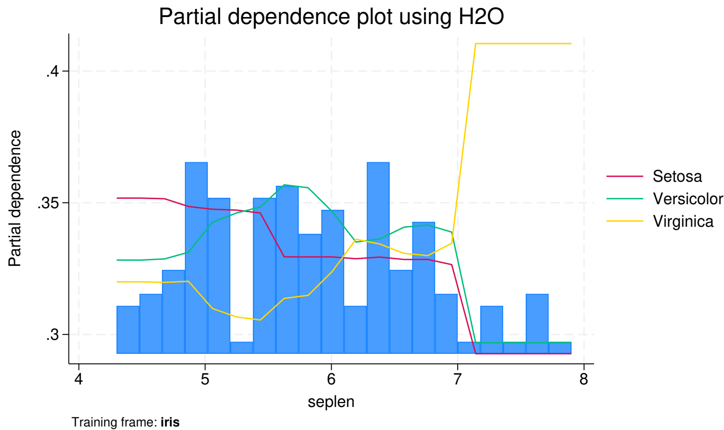
Metric	Training
Log loss	.118939
Mean class error	.0533333
MSE	.037385
RMSE	.1933519

To plot the partial dependence after multiclass classification, we need to specify the `target()` option in `h2omlgraph pdp`. In the `target()` option, we specify the names of the classes of the response `iris` for which we want to produce a PDP. We can list the classes of the response by typing

```
. _h2oframe levelsof iris
`"Setosa"' `"Versicolor"' `"Virginica"'
```

Next we plot the partial dependence of the predictor `seplen` on all three classes.

```
. h2omlgraph pdp seplen, target(Setosa Versicolor Virginica)
Progress (%): 0 100
```



On the plot, the red line corresponds to the PDP for the `Setosa` class. The plot shows how the average probability of predicting `Setosa` differs with the different values of the predictor `seplen`.

References

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

Cinelli, C., A. Forney, and J. Pearl. 2024. A crash course in good and bad controls. *Sociological Methods and Research* 53: 1071–1104. https://doi.org/10.1177/00491241221099552.

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.

Imbens, G. W., and D. B. Rubin. 2015. *Causal Inference for Statistics, Social, and Biomedical Sciences: An Introduction*. New York: Cambridge University Press. https://doi.org/10.1017/CBO9781139025751.

Krishna, S., T. Han, A. Gu, S. Wu, S. Jabbari, and H. Lakkaraju. 2022. The disagreement problem in explainable machine learning: A practitioner's perspective. arXiv:2202.01602 [cs.LG], https://doi.org/10.48550/arXiv.2202.01602.

Lakkaraju, H., and O. Bastani. 2020. ""How do I fool you?": Manipulating user trust via misleading black box explanations". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 79–85. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375833.

Lundberg, S. M. 2021. Be careful when interpreting predictive models in search of causal insights. *Medium: Thoughts and Theory*. https://medium.com/towards-data-science/be-careful-when-interpreting-predictive-models-in-search-of-causal-insights-e68626e664b6.

Pearl, J. 2009. *Causality: Models, Reasoning, and Inference*. 2nd ed. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511803161.

Slack, D., S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju. 2020. "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods". In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 180–186. New York: Association for Computing Machinery. https://doi.org/10.1145/3375627.3375830.

Zhao, Q., and T. J. Hastie. 2021. Causal interpretations of black-box models. *Journal of Business and Economic Statistics* 39: 272–281. https://doi.org/10.1080/07350015.2019.1624293.

Also see

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

[H2OML] **h2omlgraph ice** — Produce individual conditional expectation plot

h2omlgraph prcurve — Produce precision–recall curve plot

Description | Quick start | Menu | Syntax
Options | Remarks and examples | References | Also see

Description

h2omlgraph prcurve plots the precision–recall curve after binary classification performed by h2oml gbbinclass and h2oml rfbinclass. With binary classification, the predicted probability for each observation is compared with a threshold value to determine whether the observation is predicted to be in the positive class or the negative class. Thus, for different threshold values, different numbers of observations are classified as positive and negative. Metrics based on the predicted classes, including precision (the proportion of correct predictions out of all observations predicted to be in the positive class) and recall (the true-positive rate), also depend on the selected threshold. Plotting the precision versus the recall for a variety of threshold values produces the precision–recall curve, which allows us to evaluate the tradeoff between precision and recall for a model.

The precision–recall curve is useful for evaluating model performance, especially for models fit to imbalanced response variables. A large area under the precision–recall curve (AUCPR) indicates good fit with both precision and recall being high.

Quick start

Plot the precision–recall curve

```
h2omlgraph prcurve
```

Same as above, but plot the curve based on the validation data

```
h2omlgraph prcurve, valid
```

Same as above, but remove the reference line

```
h2omlgraph prcurve, valid norefline
```

Menu

Statistics > H2O machine learning

Syntax

h2omlgraph prcurve [, *options*]

options	Description
Main	
models(*namelist*)	specify the name or a list of names of the stored estimation results
savedata(*filename*[, replace])	save plot data to *filename*
Plot options	
rlopts(*line_options*)	affect rendition of reference line
norefline	suppress plotting reference line
lineopts(*line_options*)	affect rendition of all precision–recall curves
line#opts(*line_options*)	affect rendition of the precision–recall curve for model #
twoway_options	any options other than by() documented in [G-3] ***twoway_options***
train	specify that precision and recall be reported using training results
valid	specify that precision and recall be reported using validation results
cv	specify that precision and recall be reported using cross-validation results
test	specify that precision and recall be computed using the testing frame
test(*framename*)	specify that precision and recall be computed using data in testing frame *framename*
frame(*framename*)	specify that precision and recall be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

models(*namelist*) specifies the name or a list of names of the stored estimation results for which the precision–recall curve is being plotted. For each model, the displayed curve corresponds to the default frame of that model when the h2omlpostestframe command has not been used to set a postestimation frame.

savedata(*filename*[, replace]) saves the plot data to a Stata data file (.dta file). replace specifies to overwrite the existing file.

Plot options

rlopts(*line_options*) affects the rendition of the reference line. See [G-3] ***line_options***.

norefline suppresses plotting the reference line. The reference line of the precision–recall curve is determined by the proportion of the response variable in the positive class, that is, the ratio of the number of positives to the total number of observations.

lineopts(*line_options*) affects the rendition of all precision–recall curves. See [G-3] ***line_options***.

line#opts(*line_options*) affects the rendition of the precision–recall curve for model #. See [G-3] ***line_options***.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding by(). These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with h2omlgraph prcurve but are not shown in the dialog box:

train, valid, cv, test, test(), and frame() specify the H2O frame for which precision and recall are reported. Only one of train, valid, cv, test, test(), or frame() is allowed.

train specifies that precision and recall be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe.

valid specifies that precision and recall be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. valid may be specified only when the validframe() option is specified with h2oml *gbm* or h2oml *rf*.

cv specifies that precision and recall be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. cv may be specified only when the cv or cv() option is specified with h2oml *gbm* or h2oml *rf*.

test specifies that precision and recall be computed on the testing frame specified with h2oml-postestframe. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set with h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that precision and recall be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, h2omlpostestframe provides a more convenient and computationally efficient process for doing this.

frame(*framename*) specifies that precision and recall be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output. This option is not allowed with the cv option.

Remarks and examples

After performing binary classification, the receiver operating characteristic (ROC) curve, introduced in [H2OML] **h2omlgraph roc**, is a common tool for evaluating model performance. However, the ROC curve is not reliable when the data are imbalanced (when the data contain very few positive classes). For imbalanced data, a small false-positive rate and a large true-positive rate are expected. Consequently, the ROC curve will be close to the upper-left corner and will indicate good fit rather than reflecting the true performance of the model. The precision–recall curve is designed to mitigate this problem by plotting the precision (the proportion of correct predictions out of all observations predicted to be in the positive class) versus the recall (the proportion of correct predictions out of all observations actually in

the positive class; also known as the true-positive rate) (Davis and Goadrich 2006). The precision–recall curve is more reliable for imbalanced data compared with the ROC curve because the false-positive rate in the ROC curve is replaced with precision, which does not rely on the number of true negatives. (The number of true negatives will be large for imbalanced data and will strongly influence the false-positive rate.)

The computation of the precision and recall metrics relies on a threshold value. After binary classification, the predicted probability for each observation is compared with a threshold value to determine whether the observation is predicted to be in the positive class or the negative class. Observations with probabilities greater than the threshold are classified as positive, and the remaining observations are classified as negative. Different threshold values lead to different predicted classes. Therefore, as the threshold changes, the precision and recall also change.

The precision–recall curve plots the precision on the y axis and the recall on the x axis, where each metric is computed across a range of threshold values. When evaluating model performance, the closer the curve is to the upper-right corner, the better the performance. Similarly, the larger the AUCPR, the better the performance.

▷ Example 1: The precision–recall curve vs. the ROC

In this example, we compare ROC and precision–recall graphs for imbalanced data.

We use a popular credit card dataset available in Kaggle (Pozzolo et al. [2015], Pozzolo et al. [2018]) to predict whether a given credit card transaction is fraudulent.

The dataset contains 28 predictors, denoted `V1,...,V28`, which are obtained after a principal component analysis transformation. Due to confidentiality issues, the original predictors are not available. The response `fraud` is a binary variable that takes value 1 in the case of fraud and value 0 otherwise.

We start by opening the dataset in Stata and using the `tabulate` command to look at the distribution of the classes of `fraud`.

```
. use https://www.stata-press.com/data/r19/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; only 0.17% of the response belongs to the class `yes`.

Next we put 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. We use the `_h2oframe split` command to randomly split the `credit` frame into a training frame (70% of observations) and a testing frame (30% of observations), which we name `train` and `test`, respectively. We also change the current frame to `train`. For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and see [H2OML] **H2O setup**.

```
. h2o init
  (output omitted)

. _h2oframe put, into(credit)
Progress (%): 0 100
```

```
. _h2oframe split credit, into(train test) split(0.7 0.3) rseed(19)
. _h2oframe change train
```

We use random forest binary classification with 3-fold cross-validation to fit a model, and we specify `h2orseed()` for reproducibility. Because our goal is to compare ROC and precision–recall curves, we do not implement tuning. We store the estimation results by using the `h2omlest store` command.

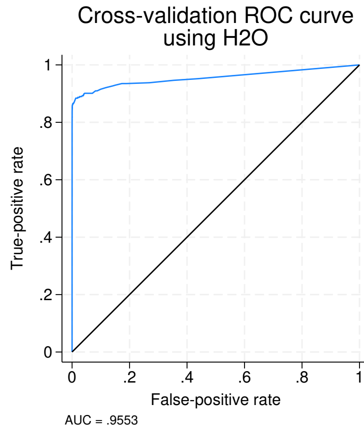
```
. h2oml rfbinclass fraud v1-v28 amount, h2orseed(19) cv(3, modulo)
Progress (%): 0 0.4 1.4 4.5 10.4 17.4 25.0 29.4 34.4 38.4 43.0 50.4 56.4 62.0
> 66.5 70.9 75.0 76.4 83.4 88.9 96.4 100
Random forest binary classification using H2O
Response: fraud
Frame:                                   Number of observations:
  Training: train                                    Training = 199,612
                                             Cross-validation = 199,612
Cross-validation: Modulo                 Number of folds      =       3
Model parameters
Number of trees      =   50
             actual =   50
Tree depth:                              Pred. sampling value =      -1
           Input max =   20              Sampling rate        =    .632
                 min =   19              No. of bins cat.     =   1,024
                 avg = 19.9              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	.0057128	.0054806
Mean class error	.0890433	.0904708
AUC	.940396	.9553414
AUCPR	.8348062	.8391036
Gini coefficient	.8807921	.9106828
MSE	.0004454	.0004531
RMSE	.0211043	.0212871

```
. h2omlest store RF
```

Now we plot the ROC curve by using the h2omlgraph roc command.

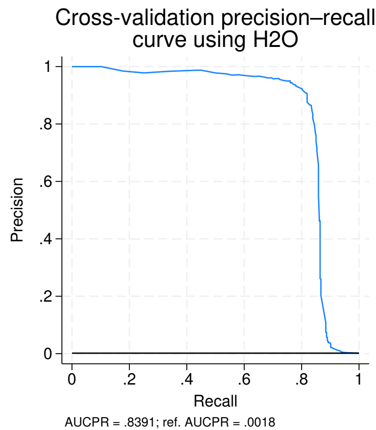
```
. h2omlgraph roc
```



As expected, the ROC curve fails to capture the imbalance in the response and shows good performance of the model.

On the other hand, the precision–recall curve, plotted below, shows an abrupt decrease in performance closer to the right side.

```
. h2omlgraph prcurve
```



The abrupt drop in precision when recall is greater than 0.8 suggests that the model's ability to distinguish between positive and negative classes diminishes substantially at certain thresholds.

The horizontal black line in the graph is the reference line. The reference line of the precision–recall curve is determined by the proportion of positive classes in the response (the ratio of the number of positives and the total number of observations). It corresponds to the model that always predicts a positive class.

Note that the h2omlgraph prcurve command by default plotted the precision and recall values based on cross-validation because the cv() option was specified and cross-validation was performed during estimation.

▷ Example 2: Comparing models using the precision–recall curve

In example 1, we plotted the precision–recall curve for random forest binary classification. In practice, the precision–recall curve is often used to compare the performance of different models and methods on a testing frame. In this example, we compare the precision–recall curves for the random forest method and the gradient boosting machine (GBM) method.

We use the h2omlpostestframe command to set the testing frame for the random forest model estimated in example 1.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

Then we perform gradient boosting binary classification and store the estimation results.

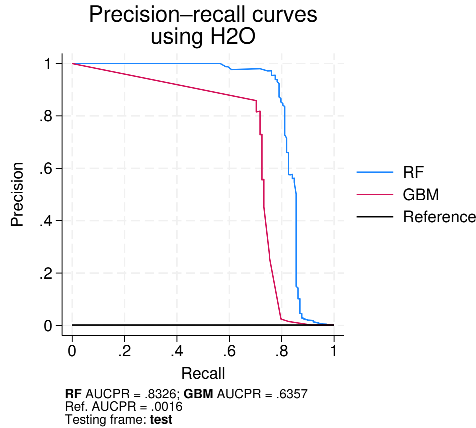
```
. h2oml gbbinclass fraud v1-v28 amount, h2orseed(19) cv(3, modulo)
Progress (%): 0 2.9 17.4 33.0 51.4 62.9 74.5 82.9 100
Gradient boosting binary classification using H2O
Response: fraud
Loss:     Bernoulli
Frame:                                    Number of observations:
  Training: train                                    Training = 199,612
                                             Cross-validation = 199,612
Cross-validation: Modulo                  Number of folds     =       3
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 =   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	Cross-validation
Log loss	.0069067	.0213072
Mean class error	.0932605	.1597576
AUC	.9220793	.8142659
AUCPR	.8075749	.5743456
Gini coefficient	.8441585	.6285319
MSE	.0004101	.0009271
RMSE	.0202519	.0304475

```
. h2omlest store GBM
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

To compare GBM and random forest, with default hyperparameters, we use `h2omlgraph prcurve` with the `models()` option.

```
. h2omlgraph prcurve, models(RF GBM)
```



Based on the graph above, random forest performs better than GBM.

References

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.

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.

Also see

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

[H2OML] **h2omlgraph roc** — Produce ROC curve plot

h2omlgraph roc — Produce ROC curve plot

Description	Quick start	Menu	Syntax
Options	Remarks and examples	Also see	

Description

`h2omlgraph roc` plots the receiver operating characteristic (ROC) curve after binary classification performed by `h2oml gbbinclass` and `h2oml rfbinclass`. With binary classification, the predicted probability for each observation is compared with a threshold value to determine whether the observation is predicted to be in the positive class or the negative class. Thus, for different threshold values, different numbers of observations are classified as positive and negative. The ROC curve allows us to evaluate the tradeoff between the true-positive rate (TPR) and false-positive rate (FPR) by plotting these metrics for a variety of threshold values.

The curve produced by plotting TPR versus FPR is useful for evaluating model performance. A large area under the curve (AUC) indicates that the model has a high true-positive rate and low false-positive rate.

Quick start

Plot the ROC curve

```
h2omlgraph roc
```

Same as above, but report results based on the validation data

```
h2omlgraph roc, valid
```

Same as above, but remove the reference line

```
h2omlgraph roc, valid norefline
```

Menu

Statistics > H2O machine learning

Syntax

h2omlgraph roc [, *options*]

options	Description
Main	
models(*namelist*)	specify the name or a list of names of stored estimation results
savedata(*filename*[, replace])	save plot data to *filename*
Plot options	
rlopts(*line_options*)	affect rendition of reference line
norefline	suppress plotting reference line
lineopts(*line_options*)	affect rendition of all ROC curves
line#opts(*line_options*)	affect rendition of the ROC curve for model #
twoway_options	any options other than by() documented in [G-3] ***twoway_options***
train	specify that the TPR and FPR be reported using training results
valid	specify that the TPR and FPR be reported using validation results
cv	specify that the TPR and FPR be reported using cross-validation results
test	specify that the TPR and FPR be computed using the testing frame
test(*framename*)	specify that the TPR and FPR be computed using data in testing frame *framename*
frame(*framename*)	specify that the TPR and FPR be computed using data in H2O frame *framename*
framelabel(*string*)	label frame as *string* in the output

train, valid, cv, test, test(), frame(), and framelabel() do not appear in the dialog box.

Options

Main

models(*namelist*) specifies the name or the list of the names of the stored estimation results for which the ROC curves are plotted. For each model, the displayed curve corresponds to the default frame of that model when a postestimation frame has not been set with h2omlpostestframe.

savedata(*filename*[, replace]) saves the plot data to a Stata data file (.dta file). replace specifies that filename be overwritten if it exists.

Plot options

rlopts(*line_options*) affects the rendition of the reference line. See [G-3] ***line_options***.

norefline suppresses plotting the reference line. The 45-degree reference line is the ROC curve that is expected if predictions are a random guess. The area between the ROC curve for the model and the reference line indicates how much better the model performs over a random guess.

lineopts(*line_options*) affects the rendition of all ROC curves. See [G-3] ***line_options***.

line#opts(*line_options*) affects the rendition of the ROC curve for model #. See [G-3] ***line_options***.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding by(). These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with h2omlgraph roc but are not shown in the dialog box:

train, valid, cv, test, test(), and frame() specify the H2O frame for which TPR and FPR are reported. Only one of train, valid, cv, test, test(), or frame() is allowed.

train specifies that TPR and FPR be reported using training results. This is the default when neither validation nor cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe.

valid specifies that TPR and FPR be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. valid may be specified only when the validframe() option is specified with h2oml *gbm* or h2oml *rf*.

cv specifies that TPR and FPR be reported using cross-validation results. This is the default when cross-validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. cv may be specified only when the cv or cv() option is specified with h2oml *gbm* or h2oml *rf*.

test specifies that TPR and FPR be computed on the testing frame specified with h2omlpostest-frame. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set with h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that TPR and FPR be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, h2omlpostestframe provides a more convenient and computationally efficient process for doing this.

frame(*framename*) specifies that TPR and FPR be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output. This option is not allowed with the cv option.

Remarks and examples

ROC curves graphically illustrate how well a model performs in terms of the TPR and FPR.

After binary classification, the predicted probability for each observation is compared with a threshold value to determine whether the observation is predicted to be in the positive class or the negative class. Observations with probabilities greater than the threshold are classified as positive, and the remaining observations are classified as negative. Different threshold values lead to different predicted classes. Therefore, as the threshold changes, the numbers of true positives and false positives also change.

The ROC curve plots the TPR on the y axis and FPR on the x axis, where each metric is computed across a range of threshold values. This is useful for evaluating model performance. When the area under the ROC curve is large (close to 1), the model has a high TPR and low FPR.

▷ Example 1: Basic example

To best understand the ROC curve, we can find it helpful to first consider the TPR and FPR for individual threshold values. Below, we use the `h2omlestat threshmetric` command to obtain these metrics for three different threshold values.

```
. h2omlestat threshmetric, threshold(0)
Metrics for specific threshold using H2O
Training frame: auto
────────────────────────────────────────
Threshold                  │
                     Input │          0
                  Computed │          0
───────────────────────────┼────────────
Metric                     │
                        F1 │      .4583
                        F2 │       .679
                      F0.5 │      .3459
                  Accuracy │      .2973
                 Precision │      .2973
                    Recall │          1
               Specificity │          0
       Min. class accuracy │          0
      Mean class accuracy  │         .5
            True negatives │          0
           False negatives │          0
            True positives │         22
           False positives │         52
        True-negative rate │          0
       False-negative rate │          0
        True-positive rate │          1
       False-positive rate │          1
                       MCC │          0
────────────────────────────────────────
```

A threshold of 0 produces a TPR of 1 and an FPR of 1.

```
. h2omlestat threshmetric, threshold(0.1)
Metrics for specific threshold using H2O
Training frame: auto
```

Threshold	
Input	.1
Computed	.125
Metric	
F1	.7
F2	.8333
F0.5	.6034
Accuracy	.7568
Precision	.5526
Recall	.9545
Specificity	.6731
Min. class accuracy	.6731
Mean class accuracy	.8138
True negatives	35
False negatives	1
True positives	21
False positives	17
True-negative rate	.6731
False-negative rate	.0455
True-positive rate	.9545
False-positive rate	.3269
MCC	.5739

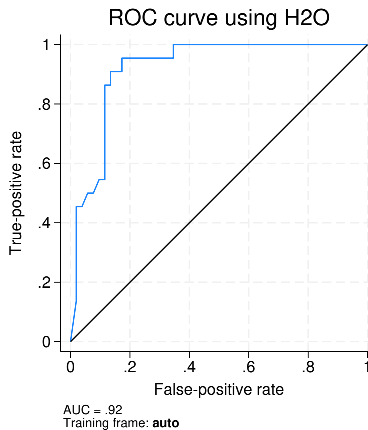
A threshold of 0.1 produces a TPR of 0.9545 and an FPR of 0.3269.

```
. h2omlestat threshmetric, threshold(1)
Metrics for specific threshold using H2O
Training frame: auto
```

Threshold	
Input	1
Computed	1
Metric	
F1	.2308
F2	.163
F0.5	.3947
Accuracy	.7297
Precision	.75
Recall	.1364
Specificity	.9808
Min. class accuracy	.1364
Mean class accuracy	.5586
True negatives	51
False negatives	19
True positives	3
False positives	1
True-negative rate	.9808
False-negative rate	.8636
True-positive rate	.1364
False-positive rate	.0192
MCC	.2368

A threshold of 1 produces a TPR of 0.1364 and an FPR of 0.0192.

If we repeat the same exercise with more threshold values and graph the corresponding TPRs and FPRs, the resulting curve is the ROC curve in the graph below.



The black reference line is the ROC curve for a method that randomly classifies with probability equal to 0.5. Therefore, a model that has a ROC curve that lies below the reference line performs worse than a random guess. Similarly, the further a model's ROC curve lies above the reference line, the better the model performs over a random guess.

We can also use ROC curves to compare models. The ROC curve located closest to the upper-left corner has the best performance. If ROC curves of two models overlap, then the higher AUC may indicate a better performance. In `h2omlgraph roc`, we can compare models by specifying the `models()` option with the names of two or more stored results.

◁

▷ Example 2: ROC for one model

In this example, we plot and interpret the ROC curve after performing random forest binary classification.

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. We use the `_h2oframe split` command to randomly split the `auto` frame into a training frame (80% of observations) and a testing frame (20% of observations), which we name `train` and `test`, respectively. We also change the current frame to `train`. 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/auto
(1978 automobile dataset)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe split auto, into(train test) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

Next we perform random forest binary classification with 3-fold cross-validation and store the estimation results by using the `h2omlest store` command.

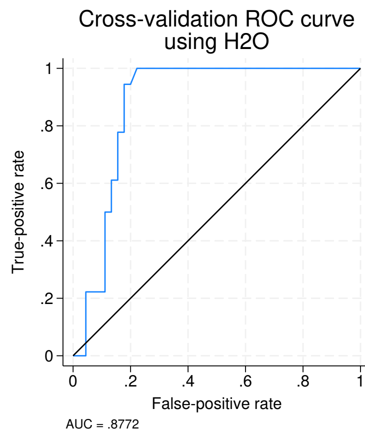
```
. global predictors price mpg trunk weight length
. h2oml rfbinclass foreign $predictors, h2orseed(19) cv(3, modulo)
Progress (%): 0 78.5 100
Random forest binary classification using H2O
Response: foreign
Frame:                                  Number of observations:
  Training: train                                  Training =     63
                                           Cross-validation =     63
Cross-validation: Modulo                Number of folds      =      3
Model parameters
Number of trees        =  50
              actual =  50
Tree depth:                             Pred. sampling value =     -1
           Input max =  20              Sampling rate        =   .632
                 min =   4              No. of bins cat.     =  1,024
                 avg = 5.3              No. of bins root     =  1,024
                 max =   8              No. of bins cont.    =     20
Min. obs. leaf split =   1              Min. split thresh.   = .00001
Metric summary
```

Metric	Training	Cross-validation
Log loss	.8986088	.4191571
Mean class error	.1166667	.1166667
AUC	.8851852	.8771605
AUCPR	.590704	.5771737
Gini coefficient	.7703704	.754321
MSE	.1331692	.144763
RMSE	.3649235	.3804774

```
. h2omlest store RF
```

Finally, we plot the ROC curve by using the `h2omlgraph roc` command.

```
. h2omlgraph roc
```



Because the cv() option was specified and cross-validation was performed during the estimation, the default reported results correspond to the metrics calculated using cross-validation. The closer the curve is to the upper-left corner, the better the performance. This model performs substantially better than the reference line corresponding to random guessing.

Example 3: Comparing models using ROC

In example 2, we plotted the ROC curve for the random forest binary classification. In practice, the ROC curve is often used to compare the performance of different models on a testing frame. In this example, we compare the ROC curve for the random forest method with the one for the gradient boosting machine (GBM) method.

We use the h2omlpostestframe command to set the testing frame for the random forest model estimated in example 2.

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

Then we perform gradient boosting binary classification, set the testing frame for this model, and store the estimation results.

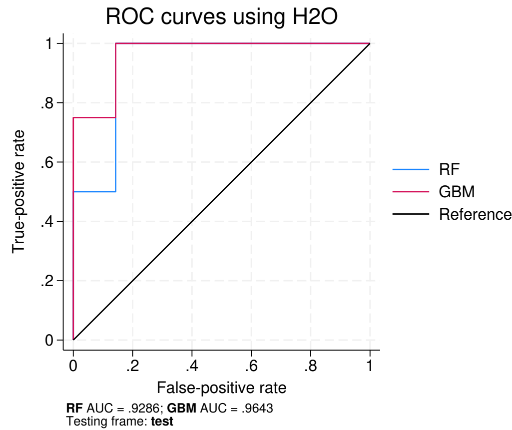
```
. h2oml gbbinclass foreign $predictors, h2orseed(19) cv(3, modulo)
Progress (%): 0 100
Gradient boosting binary classification using H2O
Response: foreign
Loss:     Bernoulli
Frame:                                      Number of observations:
  Training: train                                      Training =     63
                                               Cross-validation =     63
Cross-validation: Modulo                    Number of folds      =      3
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 = 3.5                  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	Cross-validation
Log loss	.0931244	.2803522
Mean class error	.0111111	.0666667
AUC	.9975309	.9259259
AUCPR	.9938208	.7733418
Gini coefficient	.9950617	.8518519
MSE	.0211802	.096305
RMSE	.1455344	.3103305

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
. h2omlest store GBM
```

To compare the ROC curves of the GBM and random forest models, with default hyperparameters, we use h2omlgraph roc with the models() option.

```
. h2omlgraph roc, models(RF GBM)
```



Based on the graph above, GBM performs better than random forest.

Also see

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

h2omlgraph scorehistory — Produce score history plot

Description Quick start Menu Syntax
Options Remarks and examples Also see

Description

`h2omlgraph scorehistory` plots the evolution of a performance metric (a score) as the number of trees grows in a machine learning model fit using either `h2oml` *gbm* or `h2oml` *rf*. The performance metric is based on the training set. If validation was specified during estimation, the performance metric on the validation set is also plotted. If cross-validation was specified during estimation, the performance metric based on the cross-validation results and based on the training on cross-validation results is also plotted.

Quick start

Plot the score history

```
h2omlgraph scorehistory
```

Same as above, but show the best score reference line

```
h2omlgraph scorehistory, bsline
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlgraph scorehistory` [, *options*]

options	Description
Main	
`metric(`*metric*`)`	specify the metric (score) to be plotted
`table`	display results as a table
`savedata(`*filename*[, `replace`]`)`	save plot data to *filename*
Plot options	
`bsline`	plot the best score reference line
`bslineopts(`*line_options*`)`	affect rendition of the best score reference line
`lineopts(`*line_options*`)`	affect rendition of all training, validation, and cross-validation curves
`trainlineopts(`*line_options*`)`	affect rendition of training curve
`validlineopts(`*line_options*`)`	affect rendition of validation curve
`cvtrainlineopts(`*line_options*`)`	affect rendition of the training on cross-validation curve
`cvlineopts(`*line_options*`)`	affect rendition of cross-validation curve
`nocvtrainsd`	do not plot the standard deviation band for the training on cross-validation curve
`cvtrainsdopts(`*area_options*`)`	affect rendition of the standard deviation band for the training on cross-validation curve
`nocvsd`	do not plot the standard deviation band for the cross-validation curve
`cvsdopts(`*area_options*`)`	affect rendition of the standard deviation band for the cross-validation curve
twoway_options	any options other than `by()` documented in [G-3] ***twoway_options***
`trainopts(`*line_options*`)`	synonym for `trainlineopts()`
`validopts(`*line_options*`)`	synonym for `validlineopts()`
`cvtrainopts(`*line_options*`)`	synonym for `cvtrainlineopts()`
`cvopts(`*line_options*`)`	synonym for `cvlineopts()`

Options

Main

`metric(`*metric*`)` specifies the metric to be plotted. The allowed options are the following:

After regression: `deviance`, `rmse`, and `mae`.

After binary classification: `logloss`, `misclassification`, `auc`, `aucpr`, and `rmse`.

After multiclass classification: `logloss`, `misclassification`, and `rmse`.

`deviance` is the default metric for regression. `logloss` is the default metric for binary and multiclass classification.

`table` displays results as a table. The table is suppressed by default.

`savedata(`*filename*[, `replace`]`)` saves the plot data to a Stata data file (`.dta` file). `replace` specifies that *filename* be overwritten if it exists.

Plot options

`bsline` plots the best score reference line for the training, validation, or cross-validation curve. The best score corresponds to the optimal training score (the optimal metric) if neither validation nor cross-validation is performed during estimation. When validation or cross-validation is performed, the best score corresponds to the optimal validation or cross-validation score, respectively.

`bslineopts(`*line_options*`)` affects rendition of the best score reference line. For options, see [G-3] ***line_options***.

`lineopts(`*line_options*`)` affects the rendition of both training and validation curves when `validframe()` is specified during estimation or the rendition of training, training on cross-validation, and cross-validation curves when `cv()` is specified during estimation. If neither `validframe()` nor `cv()` is specified, only training curve is affected. See [G-3] ***line_options***.

`trainlineopts(`*line_options*`)` affects the rendition of the training curve. See [G-3] ***line_options***.

`validlineopts(`*line_options*`)` affects the rendition of the validation curve when `validframe()` is specified during estimation. See [G-3] ***line_options***.

`cvtrainlineopts(`*line_options*`)` affects the rendition of the training on cross-validation curve when `cv()` is specified during estimation. During k-fold cross-validation, the training data are separated into k folds, from which $k-1$ are used for training and 1 for prediction. The training on cross-validation curve plots the average across the k cross-validation iterations of the metrics computed on the training data (from $k-1$ folds). See [G-3] ***line_options***.

`cvlineopts(`*line_options*`)` affects the rendition of the cross-validation curve when `cv()` is specified during estimation. See [G-3] ***line_options***.

`nocvtrainsd` suppresses plotting the standard deviation band for the mean training on cross-validation curve. The standard deviation band is included by default.

`cvtrainsdopts(`*area_options*`)` affects rendition of the standard deviation band for mean training on cross-validation metrics. See [G-3] ***area_options***.

`nocvsd` suppresses plotting the standard deviation band for the mean cross-validation curve.

`cvsdopts(`*area_options*`)` affects rendition of the standard deviation band for the mean cross-validation curve. See [G-3] ***area_options***.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding `by()`. These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

`trainopts(`*line_options*`)` is a synonym for `trainlineopts()`.

`validopts(`*line_options*`)` is a synonym for `validlineopts()`.

`cvtrainopts(`*line_options*`)` is a synonym for `cvtrainlineopts()`.

`cvopts(`*line_options*`)` is a synonym for `cvlineopts()`.

Remarks and examples

We assume you have read [H2OML] **Intro**.

Overfitting occurs when a machine learning model fits the training data too well. This harms the ability of the model to generalize to new data, increasing the generalization error. Underfitting occurs when performance can be improved by increasing complexity of the model by modifying the hyperparameters.

The score history curve, also known as the learning curve, is a useful graphical tool for examining the overfitting or underfitting of a model. It plots a performance metric (a score) as a function of the number of trees and allows you to evaluate the optimal number of trees.

▷ Example 1: Over- and underfitting with score history

Consider churn.dta, described in example 1 of [H2OML] **h2oml** and where the goal is to build a predictive model that will predict the best behavior of a customer who is more likely to churn or retain the company's services.

We start by opening the churn dataset 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. We use the _h2oframe split command to randomly split the churn frame into a training frame (80% of observations) and a validation frame (20% of observations), which we name train and valid, respectively. We also change the current frame to train. 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/churn
(Telco customer churn data)

. h2o init
  (output omitted)

. _h2oframe put, into(churn)
Progress (%): 0 100

. _h2oframe split churn, into(train valid) split(0.8 0.2) rseed(19)

. _h2oframe change train
```

Next we define a global macro, predictors, to store predictors, and perform gradient boosting binary classification with 200 trees.

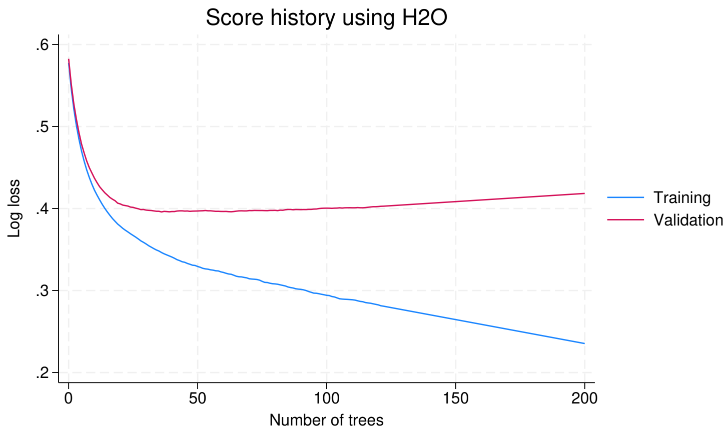
```
. global predictors latitude longitude tenuremonths monthlycharges
> totalcharges gender seniorcitizen partner dependents phoneservice
> multiplelines internetserv onlinesecurity onlinebackup deviceprotect
> techsupport streamtv streammovie contract paperlessbill paymethod

. h2oml gbbinclass churn $predictors, validframe(valid) ntrees(200) h2orseed(19)
Progress (%): 0 5.9 19.4 56.0 100

Gradient boosting binary classification using H2O

Response: churn
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training:   train                                Training =  5,643
  Validation: valid                              Validation =  1,400

Model parameters

Number of trees       = 200             Learning rate       =     .1
             actual = 200               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	.2353826	.4184287
Mean class error	.0982787	.2314265
AUC	.9692747	.8515924
AUCPR	.9264498	.6724044
Gini coefficient	.9385495	.7031848
MSE	.0679986	.1370254
RMSE	.2607655	.3701694

Next we plot the score history curve by using the `h2omlgraph scorehistory` command.

```
. h2omlgraph scorehistory
Training frame:   train
Validation frame: valid
```



We can see that when the number of trees is fewer than 10, learning and generalization behave similarly. In other words, the log loss is similar for the training and validation data. For these small numbers of trees, the log-loss metric is large; the model is underfitting the training data, and performance can be improved. However, when the number of trees exceeds 40, the log-loss metric for the validation data starts to increase. Generalization stops improving, even though the training metrics continue to improve. This indicates that the model learns patterns specific to training data that cannot be extended to new data points. At this stage, the model is overfitting.

▷ Example 2: Score history with cross-validation

In example 1, we used a validation frame during estimation. When cross-validation is used, the `h2omlgraph scorehistory` command provides not only the score history curves for cross-validation but also standard deviation bands for quantifying uncertainty.

We open `auto.dta` in Stata and then put it into an H2O frame. Because we are focused on evaluating cross-validation, we do not split the data into training and testing sets as we typically would in practice.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

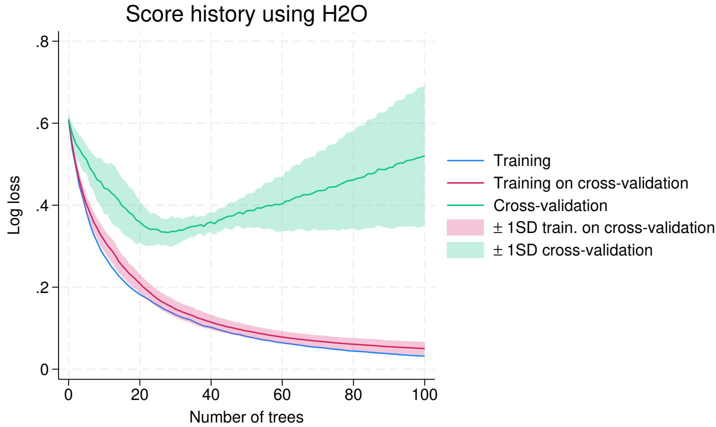
We perform gradient boosting binary classification with 3-fold cross-validation and use 100 trees.

```
. h2oml gbbinclass foreign price length weight trunk mpg, h2orseed(19)
> cv(3, modulo) ntrees(100)
Progress (%): 0 40.7 100
Gradient boosting binary classification using H2O
Response: foreign
Loss:     Bernoulli
Frame:                                     Number of observations:
  Training: auto                                      Training =     74
                                              Cross-validation =     74
Cross-validation: Modulo                   Number of folds     =      3
Model parameters
Number of trees       = 100                Learning rate       =     .1
               actual = 100                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 = 4.3                 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	Cross-validation
Log loss	.0319483	.5174966
Mean class error	0	.1153846
AUC	1	.9143357
AUCPR	1	.802104
Gini coefficient	1	.8286713
MSE	.0050191	.1460853
RMSE	.0708458	.3822111

Next we plot the score history using the `h2omlgraph scorehistory` command.

```
. h2omlgraph scorehistory
Training frame: auto
```



The band representing the cross-validation standard deviation, displayed in green, has an hourglass-like shape. The uncertainty is greater at the beginning, where the model is underfitting. It then narrows in the regions where the model's performance is likely to generalize well before widening again at the end, where the model overfits the training data.

Also see

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

h2omlgraph shapsummary — Produce SHAP beeswarm plot

Description Quick start Menu Syntax
Options Remarks and examples Also see

Description

h2omlgraph shapsummary produces the beeswarm plot of Shapley additive explanation (SHAP) values after regression or binary classification performed by h2oml gbregress, h2oml rfregress, h2oml gbbinclass, or h2oml rfbinclass. SHAP values indicate the contributions of predictors to the prediction for a given observation. The beeswarm plot allows visualization of SHAP values for many observations by placing them in a one-dimensional scatterplot for each predictor where the overlapping observations are separated (or jittered) so that each SHAP value is visible.

SHAP values are considered a unified measure for variable importance and machine learning model explanation. For an overview of SHAP values, see *Remarks and examples* in [H2OML] **h2omlgraph shapvalues**.

Quick start

Plot SHAP summary

```
h2omlgraph shapsummary
```

Same as above, but plot the summary for predictors x1, x2, and x3

```
h2omlgraph shapsummary x1-x3
```

Plot the summary for the top 5 highest SHAP-important predictors

```
h2omlgraph shapsummary, top(5)
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlgraph shapsummary` [*predictors*] [, *options*]

options	Description
Main	
`top(#)`	display the top # highest SHAP-important predictors; default is `top(20)`
`samples(#)`	specify the number of observations to be randomly sampled to estimate the SHAP approximation; default is `samples(1000)`
`rseed(#)`	set random-number seed to #
`savedata(`*filename*[, `replace`]`)`	save plot data to *filename*
Plot options	
`norefline`	suppress vertical reference line identifying the origin
`rlopts(`*line_options*`)`	affect rendition of reference line
`startcolor(`*colorstyle*`)`	determine starting color for the color legend
`endcolor(`*colorstyle*`)`	determine ending color for the color legend
`jitter(#)`	affect the magnitude of jitter of overlapped observations
twoway_options	any option other than `by()` documented in [G-3] ***twoway_options***
`train`	specify that the SHAP summary be reported using training results
`valid`	specify that the SHAP summary be reported using validation results
`test`	specify that the SHAP summary be computed using testing frame
`test(`*framename*`)`	specify that the SHAP summary be computed using data in testing frame *framename*
`frame(`*framename*`)`	specify that the SHAP summary be computed using data in H2O frame *framename*
`framelabel(`*string*`)`	label frame as *string* in the output

`train`, `valid`, `test`, `test()`, `frame()`, and `framelabel()` do not appear in the dialog box.

Options

Main

`top(#)` specifies the number of highest SHAP-important predictors to be included in the plot. Up to 20 top important predictors are included by default. `top()` is not allowed if *predictors* are specified.

`samples(#)` specifies the maximum number of observations to be randomly sampled with replacement to approximate the estimate of the contribution function. The default is `samples(1000)`.

`rseed(#)` specifies the random-number seed for reproducibility.

`savedata(`*filename*[, `replace`]`)` saves the plot data to a Stata data file (`.dta` file). `replace` specifies that *filename* be overwritten if it exists.

Plot options

`norefline` suppresses the vertical reference line identifying the origin. The line is included by default.

`rlopts(`*line_options*`)` affects the rendition of the reference line. See [G-3] ***line_options***.

`startcolor(`*colorstyle*`)` determines the starting color of the color legend. The color legend shows whether the value of the given predictor for the observation is low (starting color) or high (ending color). See [G-4] ***colorstyle***.

`endcolor(`*colorstyle*`)` determines the ending color of the color legend. The color legend shows whether the value of the given predictor for the observation is low (starting color) or high (ending color). See [G-4] ***colorstyle***.

`jitter(`#`)` adds spherical random noise to the data before plotting. # represents the size of the noise as a percentage of the graphical area.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding `by()`. These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with `h2omlgraph shapsummary` but are not shown in the dialog box:

`train`, `valid`, `test`, `test()`, and `frame()` specify the H2O frame for which SHAP summary is reported. Only one of `train`, `valid`, `test`, `test()`, or `frame()` is allowed.

`train` specifies that SHAP summary be reported using training results. This is the default when validation is not performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`.

`valid` specifies that SHAP summary be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with `h2omlpostestframe`. `valid` may be specified only when the `validframe()` option is specified with `h2oml` *gbm* or `h2oml` *rf*.

`test` specifies that SHAP summary be computed on the testing frame specified with `h2omlpostestframe`. This is the default when a testing frame is specified with `h2omlpostestframe`. `test` may be specified only after a testing frame is set by using `h2omlpostestframe`. `test` is necessary only when a subsequent `h2omlpostestframe` command is used to set a default postestimation frame other than the testing frame.

`test(`*framename*`)` specifies that SHAP summary be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, it is more computationally efficient and convenient to specify the testing frame by using `h2omlpostestframe` instead of specifying `test(`*framename*`)` with individual postestimation commands.

`frame(`*framename*`)` specifies that SHAP summary be computed using the data in H2O frame *framename*.

`framelabel(`*string*`)` specifies the label to be used for the frame in the output.

Remarks and examples

We assume you have read the introduction to explainable machine learning in *Interpretation and explanation* in [H2OML] **Intro** and [H2OML] **h2omlgraph shapvalues**.

Additional examples can be found in example 6 of [H2OML] **h2oml** and example 2 of [H2OML] **h2omlgraph shapvalues**.

SHAP values explain the predictions of a model by measuring the contribution of each predictor to those predictions. For an overview of SHAP values and how they are computed, see *Remarks and examples* in [H2OML] **h2omlgraph shapvalues**. SHAP values can be computed for each observation in the dataset. The `h2omlgraph shapvalues` command allows you to plot SHAP values for one observation at a time. The `h2omlgraph shapsummary` command discussed here provides a summary beeswarm plot for evaluating the contribution of predictors across many observations.

▷ Example 1: Interpreting a SHAP summary plot

In this example, we interpret a SHAP summary plot after performing random forest regression.

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/r19/auto
(1978 automobile data)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

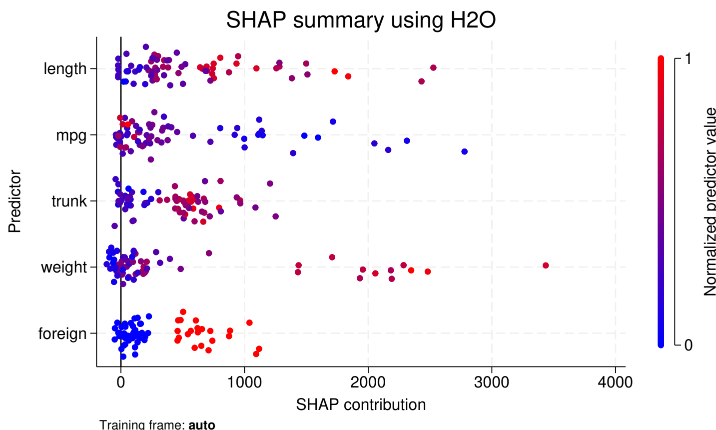
For simplicity, we save the predictor names in the global macro `predictors` in Stata. We then perform random forest regression with 100 trees and limit the maximum depth of the trees to 5.

```
. global predictors foreign mpg trunk weight length
. h2oml rfregress price $predictors, h2orseed(19) ntrees(100) maxdepth(5)
Progress (%): 0 100
Random forest regression using H2O
Response: price
Frame:                                       Number of observations:
  Training: auto                                        Training =     74
Model parameters
Number of trees       = 100
              actual = 100
Tree depth:                                  Pred. sampling value =     -1
           Input max =   5                   Sampling rate        =   .632
                 min =   2                   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 =   1                   Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Deviance	3129378
MSE	3129378
RMSE	1769.005
RMSLE	.2315556
MAE	1229.955
R-squared	.6353542

Finally, we use the `h2omlgraph shapsummary` command to plot the SHAP summary. The `samples(300)` option specifies that 300 randomly sampled observations be used, and the `rseed(19)` option is for reproducibility.

```
. h2omlgraph shapsummary, samples(300) rseed(19)
```



The summary plot is a beeswarm plot that provides a summary of how the predictors in a dataset affect the model's predictions. In the graph, for each predictor, each observation is represented as a dot. The horizontal location shows the contributed SHAP value for a specific observation. Colors show whether the predictor has high (red) or low (blue) observed values. For example, smaller observed values of weight are mostly associated with smaller SHAP contributions and a smaller predicted price. On the other hand, smaller observed values of `mpg` mostly imply larger SHAP contributions and a larger predicted price.

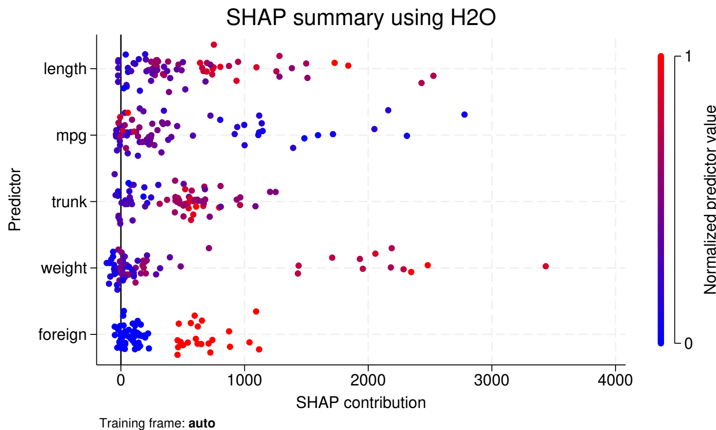
`h2omlgraph shapsummary` offers a number of options to control the look of this graph. The start color and end color for the normalized predictions can be changed by using the `scolor()` and `ecolor()` options. We can specify the `jitter()` option to control how much the observations overlap. We can also specify the `sample()` option to control the maximum number of observations to be sampled from the dataset.

▷ Example 2: Explaining voting behavior

In example 2 of [H2OML] **h2omlgraph shapvalues**, we used local SHAP explanation to study voting behavior for a specific observation. In this example, we use `h2omlgraph shapsummary` to explain voting behavior from a global perspective.

We assume that the `h2oml gbbinclass` command in example 2 of [H2OML] **h2omlgraph shapvalues** has been run to perform gradient boosting binary classification. Here we focus on the SHAP summary plot for the top 5 SHAP-important predictors.

```
. h2omlgraph shapsummary, top(5) rseed(19)
Progress (%): 0 100
```



For binary classification, the explanation is with respect to the positive class, which in our case is `vote = Yes`. We see that being young (represented by blue points for `age`) has a negative effect on the probability of voting because lower ages are mostly associated with negative SHAP contributions. The `p2000`, `p2002`, `p2004`, and `g2002` variables are indicators for voting in primary and general elections. We see that the previous voting behavior of the subjects has a substantial effect on future voting behavior.

Also see

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

[H2OML] **h2omlgraph shapvalues** — Produce SHAP values plot for individual observations

h2omlgraph shapvalues — Produce SHAP values plot for individual observations

Description | Quick start | Menu | Syntax
Options | Remarks and examples | References | Also see

Description

h2omlgraph shapvalues plots the Shapley additive explanation (SHAP) values for an individual observation after regression or binary classification performed by h2oml gbregress, h2oml rfregress, h2oml gbbinclass, or h2oml rfbinclass. SHAP values indicate the contributions of predictors to the prediction for a given observation. SHAP values are considered a unified measure for variable importance and machine learning model explanation.

Quick start

Plot individual SHAP values for the third observation

```
h2omlgraph shapvalues, obs(3)
```

Same as above, but use H2O frame myframe and predictors x1, x2, and x3

```
h2omlgraph shapvalues x1-x3, obs(3) frame(myframe)
```

Same as above, but instead of x1, x2, and x3, plot the top 4 SHAP-important predictors

```
h2omlgraph shapvalues, obs(3) frame(myframe) top(4)
```

Same as above, but save the result in the shapval3.dta file

```
h2omlgraph shapvalues, obs(3) frame(myframe) top(4) ///
   savedata(shapval3, replace)
```

Menu

Statistics > H2O machine learning

Syntax

`h2omlgraph shapvalues` [*predictors*] `, obs(#)` [*options*]

options	Description
Main	
* `obs(#)`	specify the observation number for which SHAP will be computed
`impplot`	plot SHAP values as zero-based importance—as deviations from zero rather than deviations from average prediction
`top(#)`	display the top # highest SHAP-important predictors; default is `top(20)`
`savedata(`*filename*[`, replace`]`)`	save plot data to *filename*
Plot options	
`norefline`	suppress reference line at zero for zero-based importance
`rlopts(`*line_options*`)`	affect rendition of reference line for zero-based importance
`nopredline`	suppress prediction line
`predlineopts(`*line_options*`)`	affect rendition of prediction line
`nopredlabel`	suppress label of prediction line
`predlabelopts(`*textbox_options*`)`	affect labeling of prediction line
`nobiasline`	suppress bias line
`biaslineopts(`*line_options*`)`	affect rendition of bias line that identifies the expected model prediction
`nobiaslabel`	suppress label of bias line
`biaslabelopts(`*textbox_options*`)`	affect labeling of bias line
`noboundarylines`	suppress boundary lines for SHAP contribution bars
`boundarylineopts(`*line_options*`)`	affect rendition of boundary lines for SHAP contribution bars
`novaluelabel`	suppress labels of SHAP values
`valuelabelopts(`*label_opts*`)`	affect labeling of SHAP values
`poscolor(`*colorstyle*`)`	affect color for positive SHAP values
`negcolor(`*colorstyle*`)`	affect color for negative SHAP values
`bar#opts(`*bar_opts*`)`	affect rendition of the bar for the #th SHAP-important predictor
`baropts(`*bar_opts*`)`	affect rendition of all bars for the SHAP plot
`barwidth(#)`	specify the bar width; default is `barwidth(0.9)`
Y axis, X axis, Titles, Legend, Overall	
twoway_options	any option other than `by()` documented in [G-3] ***twoway_options***
`train`	specify that SHAP values be reported using training results
`valid`	specify that SHAP values be reported using validation results
`test`	specify that SHAP values be computed using testing frame
`test(`*framename*`)`	specify that SHAP values be computed using data in testing frame *framename*
`frame(`*framename*`)`	specify that SHAP values be computed using data in H2O frame *framename*
`framelabel(`*string*`)`	label frame as *string* in the output

*`obs()` is required.

`train`, `valid`, `test`, `test()`, `frame()`, and `framelabel()` do not appear in the dialog box.

Options

Main

`obs(#)` specifies the observation number for which SHAP will be computed. # must be a positive integer. `obs()` is required.

`impplot` plots SHAP values as deviations from zero rather than deviations from the average model prediction. `impplot` is not allowed with any of options `predlineopts()`, `predlabelopts()`, `biaslineopts()`, `biaslabelopts()`, `valuelabelopts()`, or `boundarylineopts()`.

`top(#)` specifies the number of highest SHAP-important predictors to be included in the plot. Up to 20 top important predictors are included by default. `top()` is not allowed if *predictors* are specified.

`savedata(`*filename*[`, replace`]`)` saves the plot data to a Stata data file (`.dta` file). `replace` specifies that *filename* be overwritten if it exists.

Plot options

`norefline` suppresses the reference line at zero when zero-based importance is plotted. `norefline` may be specified with only option `impplot`. The reference line is included by default.

`rlopts(`*line_options*`)` affects the rendition of the reference line at zero for zero-based importance. `rlopts()` must be specified with the option `impplot`. See [G-3] ***line_options***.

`nopredline` suppresses prediction line identifying the predicted value for regression or the predicted probability for classification. When gradient boosting machine is used, the predicted values correspond to the raw predictions of the model before applying the inverse link function.

`predlineopts(`*line_options*`)` affects rendition of prediction line. See [G-3] ***line_options***. `predlineopts()` is not allowed with `impplot`.

`nopredlabel` suppresses the label for prediction line.

`predlabelopts(`*textbox_options*`)` affects labeling of prediction line. See [G-3] ***textbox_options***. `predlabelopts()` is not allowed with `impplot`.

`nobiasline` suppresses bias line identifying the expected model response—the contribution of the model without any predictors. When gradient boosting machine is used, the bias value corresponds to the raw prediction of the model before applying the inverse link function.

`biaslineopts(`*line_options*`)` affects rendition of bias line. See [G-3] ***line_options***. `biaslineopts()` is not allowed with `impplot`.

`nobiaslabel` suppresses the label for the bias line.

`biaslabelopts(`*textbox_options*`)` affects labeling of bias line. See [G-3] ***textbox_options***. `biaslabelopts()` is not allowed with `impplot`.

`noboundarylines` suppresses the boundary lines for the SHAP contribution bars.

`boundarylineopts(`*line_options*`)` affects the rendition of the lines on the boundaries of the bars for the SHAP contributions. `boundarylineopts()` is not allowed with `impplot`. See [G-3] ***line_options***.

`novaluelabel` suppresses labeling of the SHAP contributions for each predictor.

`valuelabelopts(`*label_opts*`)` affects labeling of the SHAP values for each predictor. See [G-3] ***marker_label_options***. The labels are numbers that show the SHAP values. `valuelabel()` is not allowed with `impplot`.

poscolor(*colorstyle*) affects the bar color of the positive SHAP contributions. See [G-4] ***colorstyle***.

negcolor(*colorstyle*) affects the bar color of the negative SHAP contributions. See [G-4] ***colorstyle***.

bar#opts(*bar_opts*) affects rendition of the bar for the SHAP-important predictor #. In an h2omlgraph shapvalues plot, the order of the predictors is based on SHAP importance. The predictor with largest magnitude of SHAP values will be the first and so on. For example, to change the rendition of the bar for the third-ranked predictor, we need to specify bar3opts(). See [G-2] **graph twoway bar**.

baropts(*bar_opts*) affects rendition of all bars for the SHAP plot. See [G-2] **graph twoway bar**.

barwidth(#) specifies the width of the bar. The default is barwidth(0.9).

Y axis, X axis, Titles, Legend, Overall

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding by(). These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

The following options are available with h2omlgraph shapvalues but are not shown in the dialog box:

train, valid, test, test(), and frame() specify the H2O frame for which SHAP values are reported. Only one of train, valid, test, test(), or frame() is allowed.

train specifies that SHAP values be reported using training results. This is the default when validation is not performed during estimation and when a postestimation frame has not been set with h2omlpostestframe.

valid specifies that SHAP values be reported using validation results. This is the default when validation is performed during estimation and when a postestimation frame has not been set with h2omlpostestframe. valid may be specified only when the validframe() option is specified with h2oml *gbm* or h2oml *rf*.

test specifies that SHAP values be computed on the testing frame specified with h2omlpostestframe. This is the default when a testing frame is specified with h2omlpostestframe. test may be specified only after a testing frame is set by using h2omlpostestframe. test is necessary only when a subsequent h2omlpostestframe command is used to set a default postestimation frame other than the testing frame.

test(*framename*) specifies that SHAP values be computed using data in testing frame *framename* and is rarely used. This option is most useful when running a single postestimation command on the named frame. If multiple postestimation commands are to be run on the same test frame, it is more computationally efficient and convenient to specify the testing frame by using h2omlpostestframe instead of specifying test(*framename*) with individual postestimation commands.

frame(*framename*) specifies that SHAP values be computed using the data in H2O frame *framename*.

framelabel(*string*) specifies the label to be used for the frame in the output.

Remarks and examples

We assume you have read the introduction to explainable machine learning in *Interpretation and explanation* in [H2OML] **Intro**.

SHAP values are used to explain the predictions of a model by measuring the contribution of each predictor to those predictions. Specifically, for a given prediction, the SHAP value measures the contribution of a predictor to the deviation of that prediction from a base prediction, typically from the average prediction our model makes (Štrumbelj and Kononenko 2010, 2013; Lundberg and Lee 2017).

In a traditional linear regression with no interaction terms, the computation of SHAP has a simple closed-form solution. For example, the contribution of predictor X_1 to the prediction is simply the estimated coefficient on X_1 multiplied by the observed value x_{1i}. However, for a typical machine learning model, no such coefficients are available, so computing the contributions requires an alternative approach.

In this entry, we focus on local SHAP explanation, which allows us to explain the effect of predictors for one observation at a time. The `h2omlgraph shapvalues` command plots this type of local SHAP values. For global SHAP explanations, the `h2omlgraph shapsummary` command uses the Kernel SHAP algorithm (Lundberg and Lee 2017) and produces a beeswarm plot that summarizes how each predictor affects predictions across many observations.

For intuition on SHAP values, suppose we have trained a machine learning model, such as random forest, to predict the price of a car using three predictors: mileage (M), number of accidents (A), and the presence of add-on features (F). A new car then arrives with mileage equal to 6,000 miles, a history of 1 accident, and with add-on features. In the `h2omlgraph shapvalues` command, we specify the observation number for this new car with the `obs()` option. Finally, suppose the predicted price for the car is $32,000 and the average predicted price for all cars is $29,000. Our goal then is to measure the contribution of each predictor (M, A, and F) to the $32,000 − $29,000 = $3,000 by which the predicted price of the new car deviates from the average predicted price.

The general idea of SHAP values is to imagine that the three predictors collaborate with each other to achieve the predicted value. For example, suppose for the newly arrived car we start by adding the predictor M into our model and observe that it contributes $7,000 to the prediction, then add the number of accidents A predictor and see that it contributes −$5,000. Finally, the presence of add-on features F contributes $1,000 to the so-called coalition of predictors $\{\mathrm{M}, \mathrm{A}\}$. The contribution of all predictors then adds up to the $3,000, the deviation we computed above. Unfortunately, the contribution of each predictor depends on the order at which it enters the model; that is, it depends on the coalition of the previously entered predictors. Notice that the coalition S of predictors that entered the model before M could be one of four:

$$\mathrm{S} \in \{\{\emptyset\}, \{\mathrm{A}\}, \{\mathrm{F}\}, \{\mathrm{A}, \mathrm{F}\}\}$$

And there are eight possible coalitions of predictors:

$$\mathrm{C} = \{\mathrm{M}, \mathrm{A}, \mathrm{F}\} : \{\emptyset\}, \{\mathrm{M}\}, \{\mathrm{A}\}, \{\mathrm{F}\}, \{\mathrm{M}, \mathrm{A}\}, \{\mathrm{M}, \mathrm{F}\}, \{\mathrm{A}, \mathrm{F}\}, \{\mathrm{M}, \mathrm{A}, \mathrm{F}\}$$

Therefore, the SHAP contribution of M is a weighted average of the differences of contributions of a coalition with M, denoted $v_x(\mathrm{S} \cup \mathrm{M})$, and a coalition excluding M, denoted $v_x(\mathrm{S})$, for each possible scenario of S. Here $v_x(\mathrm{S})$ is defined as a conditional expectation of the prediction given the observed values of predictors in the coalition S,

$$v_x(\mathrm{S}) = E(\widehat{f}(\mathbf{x})|\mathbf{x}_\mathrm{S})$$

where $\widehat{f}(\mathbf{x})$ is the prediction for a specific observation $\mathbf{x}$. For more details, see Lundberg and Lee (2017) and Aas, Jullum, and Løland (2021).

For machine learning methods, there is no simple form for the weighted average and with many predictors, direct computation becomes intractable. Therefore, H2O uses the TREESHAP algorithm, introduced in Lundberg, Erion, and Lee (2018), which is an efficient procedure for the exact computation of the SHAP values.

SHAP values have desirable properties (Molnar 2022, chap. 9). For instance, the efficiency property is

$$\hat{f}(\mathbf{x}) = \phi_0 + \sum_{j=1}^{p} \phi_j$$

where $\phi_0 = E\{\hat{f}(\mathbf{x})\}$ is the average predicted contribution and ϕ_j, $j = 1, \ldots, p$ is the SHAP value of each predictor. The prediction for each observation is the sum of the average prediction plus the SHAP values for all predictors.

We can also define SHAP predictor importance (Molnar 2022, chap. 9.6), which is based on the idea that important predictors are associated with large absolute SHAP values. Thus, the global importance for predictors $j = 1, \ldots, p$ can be computed by averaging their absolute SHAP values over the observations

$$I_j = \frac{1}{N} \sum_{i=1}^{n} |\phi_j^{(i)}|$$

In `h2omlgraph shapvalues`, you can specify that only a given number of highest SHAP-important predictors to be included in the graph with the `top()` option.

▷ Example 1: Interpreting SHAP values

In this example, we interpret SHAP values after performing random forest regression.

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/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe put, into(auto)
Progress (%): 0 100

. _h2oframe change auto
```

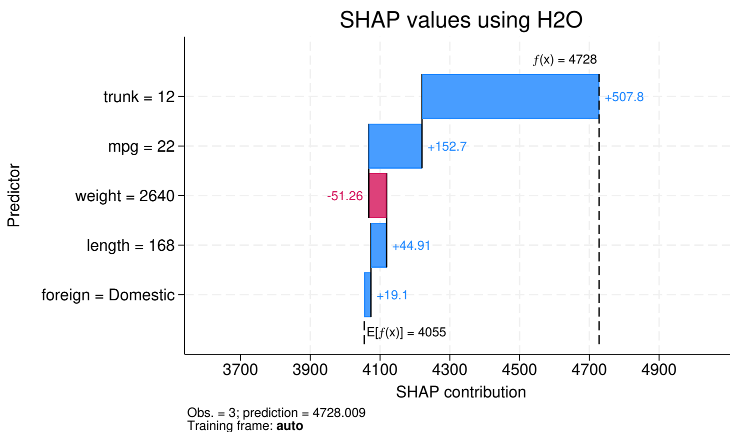
For simplicity, we save the predictor names in the global macro `predictors` in Stata. We then perform random forest regression with 100 trees and limit the maximum depth of the trees to 5.

```
. global predictors foreign mpg trunk weight length
. h2oml rfregress price $predictors, h2orseed(19) ntrees(100) maxdepth(5)
Progress (%): 0 100
Random forest regression using H2O
Response: price
Frame:                                   Number of observations:
  Training: auto                                    Training =     74
Model parameters
Number of trees       = 100
               actual = 100
Tree depth:                              Pred. sampling value =     -1
           Input max =   5               Sampling rate        =   .632
                 min =   2               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 =   1               Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Deviance	3129378
MSE	3129378
RMSE	1769.005
RMSLE	.2315556
MAE	1229.955
R-squared	.6353542

Finally, we use the `h2omlgraph shapvalues` command to plot SHAP values for the third observation.

```
. h2omlgraph shapvalues, obs(3)
```

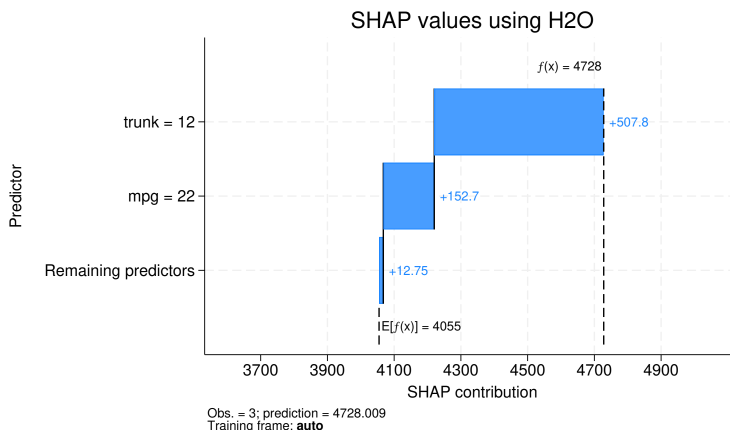


In this case, the predicted car price is 4728. We wish to explain the contribution of each predictor to this predicted price. In the plot, the contributions are plotted bottom to top, starting from the baseline value, which is the average prediction of 4055. We can see from the top blue bar that `trunk = 12` has a

positive SHAP value, which means it increases the predicted price. On the other hand, weight = 2640 has a negative contribution to the predicted price as indicated by the red bar in the center of the graph. The sum of the bars in the plot is equal to the difference of the predicted price and the bias term 4728 − 4055.

If we wish to display contributions of a subset of predictors, for example, trunk and mpg, the plot can be customized to show contributions of this subset by specifying the names of the predictors in the h2omlgraph shapvalues command.

```
. h2omlgraph shapvalues trunk mpg, obs(3)
```



In this case, the bottom bar in the plot shows the total contribution of the remaining predictors. The order of the predictors is determined based on the magnitude of their SHAP values.

◁

▷ Example 2: Explaining voting behavior

In this example, we consider the social pressure dataset described in example 1 of [H2OML] ***h2oml rf***. The goal is to explain how the predictors affect the probability of voting in the August 2006 primary election. As with most explainable machine learning methods, caution is advised when interpreting the results.

We start by opening the simulated socialpressure.dta dataset in Stata and then putting it into an H2O frame.

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

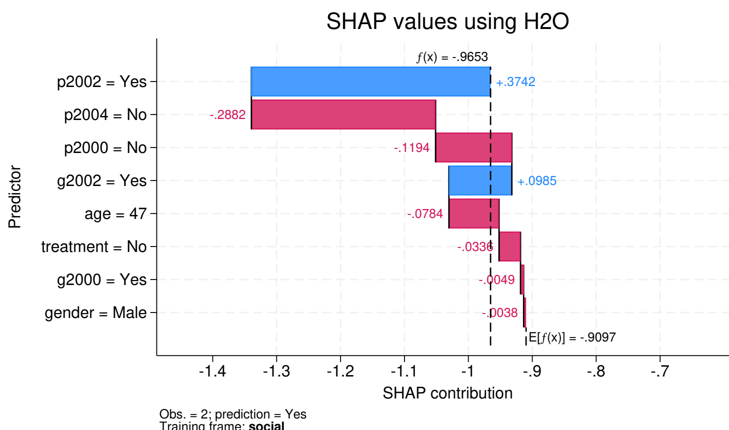
For convenience, we create a global macro, `predictors`, in Stata that contains the predictor names and perform gradient boosting binary classification with a learning rate of 0.05, a maximum tree depth of 6, and 70 trees.

```
. global predictors gender g2000 g2002 p2000 p2002 p2004 treatment age

. h2oml gbbinclass voted $predictors, h2orseed(19) lrate(0.05)
> maxdepth(6) ntrees(70)

Progress (%): 0 2.8 22.8 50.0 98.5 100

Gradient boosting binary classification using H2O

Response: voted
Loss:     Bernoulli
Frame:                                  Number of observations:
  Training: social                                 Training = 229,461

Model parameters

Number of trees      =  70              Learning rate       =     .05
              actual =  70              Learning rate decay =       1
Tree depth:                             Pred. sampling rate =       1
           Input max =   6              Sampling rate       =       1
                 min =   6              No. of bins cat.    =   1,024
                 avg = 6.0              No. of bins root    =   1,024
                 max =   6              No. of bins cont.   =      20
Min. obs. leaf split =  10              Min. split thresh.  =  .00001

Metric summary
```

Metric	Training
Log loss	.5695804
Mean class error	.3907184
AUC	.6771573
AUCPR	.4761226
Gini coefficient	.3543147
MSE	.1934469
RMSE	.439826

We display SHAP values for the second observation of the dataset by using the `h2omlgraph shapvalues` command with the option `obs(2)`. The option `xlabel()` improves the display of the figure by setting the range of the x axis to a convenient interval.

```
. h2omlgraph shapvalues, obs(2) xlabel(-1.4(0.1)-0.7)
```



The second observation corresponds to a male who voted in the primary election, so our goal is to explain why the prediction of his vote is "Yes" based on predictors. We can see that the subject being male has a very small effect on the probability of voting. On the other hand, as expected, voting in the primary election in 2002 (p2002) has a substantial positive effect on the probability of voting.

Note that the reported SHAP values after `h2oml gbbinclass` are reported as raw predictions. To interpret these values as probabilities, we need to apply the inverse logit transformation to the values shown in the graph. Similarly, for SHAP values reported after `h2oml gbregress` with a loss other than Gaussian, an appropriate transformation may be needed for interpretation. Nonetheless, the graph still allows us to infer the direction and magnitude of the predictions directly.

References

Aas, K., M. Jullum, and A. Løland. 2021. Explaining individual predictions when features are dependent: More accurate approximations to Shapley values. *Artificial Intelligence* 298: art. 103502. https://doi.org/10.1016/j.artint.2021.103502.

Lundberg, S. M., G. G. Erion, and S. Lee. 2018. Consistent individualized feature attribution for tree ensembles. arXiv:1802.03888 [cs.LG], https://doi.org/10.48550/arXiv.1802.03888.

Lundberg, S. M., and S. Lee. 2017. "A unified approach to interpreting model predictions". In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, vol. 30: 4768–4777. Red Hook, NY: Curran Associates.

Molnar, C. 2022. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. 2nd ed. https://christophm.github.io/interpretable-ml-book.

Štrumbelj, E., and I. Kononenko. 2010. An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research* 11: 1–18.

———. 2013. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems* 41: 647–665. https://doi.org/10.1007/s10115-013-0679-x.

Also see

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

[H2OML] **h2omlgraph shapsummary** — Produce SHAP beeswarm plot

h2omlgraph varimp — Produce variable importance plot

Description　Quick start　Menu　Syntax
Options　Remarks and examples　References　Also see

Description

`h2omlgraph varimp` plots the variable importance after `h2oml` *gbm* and `h2oml` *rf*. Variable importance for ensemble decision tree methods, such as random forest and gradient boosting machine, measures the relative influence of a predictor to the predictive performance of the model.

Quick start

Plot the variable importance

```
h2omlgraph varimp
```

Same as above, but plot the top 5 important predictors

```
h2omlgraph varimp, top(5)
```

Plot scaled importance of predictors

```
h2omlgraph varimp, scaled
```

Plot variable importance as a dot graph

```
h2omlgraph varimp, dot
```

Same as above, but save the graph data

```
h2omlgraph varimp, dot savedata(varimp)
```

Menu

Statistics > H2O machine learning

Syntax

h2omlgraph varimp [, *options*]

options	Description
Main	
top(#)	plot the top # important predictors; default is top(10)
proportion	plot the proportional contribution of the importance of each predictor; the default
relative	plot relative influence of each predictor
scaled	plot scaled importance of each predictor
table	display results as a table
savedata(*filename*[, replace])	save plot data to *filename*
Plot options	
bar	plot variable importance as a bar plot; the default
baropts(*bar_opts*)	affect rendition of the bar plot
dot	plot variable importance as a dot plot
dotopts(*dot_opts*)	affect rendition of the dot plot
valuelabel	display variable importance values
valuelabelopts(*label_opts*)	affect the labeling of important values
twoway_options	any options other than by() documented in [G-3] ***twoway_options***

Options

Main

top(#) plots the top # important predictors. The default is top(10).

proportion, relative, and scaled specify the type of the variable importance contribution to be plotted.

proportion plots the proportional contribution of the importance of each predictor. It is calculated by dividing the importance of each predictor by the total sum of the importance of all predictors. proportion is the default.

relative plots the importance, which is the relative influence of each predictor.

scaled plots the scaled importance. It is calculated by dividing the importance of each predictor by the largest importance score of the predictors.

Only one of proportion, relative, or scaled is allowed.

table displays results as a table. The table is suppressed by default.

savedata(*filename*[, replace]) saves the plot data to a Stata data file (.dta file). replace specifies that *filename* be overwritten if it exists.

Plot options

bar plots the variable importance as a bar plot. This is the default. bar is not allowed with dot.

`baropts(`*bar_opts*`)` affects rendition of the bar plot. *bar_opts* are any of the options documented in [G-2] **graph twoway bar**, excluding `horizontal` and `vertical`.

`dot` plots the variable importance as a dot plot. `dot` is not allowed with `bar`.

`dotopts(`*dot_opts*`)` affects the rendition of the dot plot. *dot_opts* are any of the options documented in [G-2] **graph twoway dot**, excluding `horizontal` and `vertical`.

`valuelabel` displays the values of the variable importance on the graph.

`valuelabelopts(`*label_opts*`)` affects the labeling of variable importance values. *label_opts* includes any of the options documented in [G-3] ***marker_label_options***, excluding `mlabel()`.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding `by()`, `horizontal`, and `vertical`. These include options for titling the graph (see [G-3] ***title_options***) and options for saving the graph to disk (see [G-3] ***saving_option***).

Remarks and examples

We assume you have read the *Interpretation and explanation* in [H2OML] **Intro**.

In a typical machine learning problem, the predictors influence on the outcome differs. Some of the predictors are more relevant than others. In decision trees, the variable importance of a predictor quantifies this relevance by accumulating the improvement of an impurity measure, such as cross-entropy or mean squared error (MSE), from the splitting of this predictor. For a single tree T, Breiman et al. (1984) propose to measure a relative importance of a predictor $\mathbf{X}_i$ by summing the square of relative improvements $\imath_j^2$ associated to all $J-1$ node splits,

$$I_i^2(T) = \sum_{j=1}^{J-1} \imath_j^2 I(v(j) = i)$$

where the split relative improvement $\imath_j$ is defined in (1) of [H2OML] **Intro** and is computed using entropy for classification and MSE for regression. $I(v(j) = i)$ is an indicator function, which takes 1 when the internal node is the predictor X_i. This measure easily extends to ensemble decision trees by taking an average over the number of trees. For example, if the ensemble decision tree method contains 100 trees ($t = 1, 2, \ldots, 100$), then

$$I_i^2 = \frac{1}{100} \sum_{t=1}^{100} I_i^2(T_t)$$

To find the importance for the variable X_i, we take the square root of the measure above.

For multiclass classification with K classes ($k = 1, 2, \ldots K$), there are K different models induced, where each model is an ensemble of classification trees. Then for the class k with 100 trees, the importance of the predictor X_i is computed by

$$I_{ik}^2 = \frac{1}{100} \sum_{t=1}^{100} I_i^2(T_{tk})$$

where T_{tk} is the tth tree for the class k.

It is common to plot the proportional contributions of importance values so that the total importance of all predictors sums to 1. This approach makes it easier to compare predictors. In the `h2omlgraph varimp` command, this is the default behavior. To plot the relative influences, you can specify the `relative` option.

One of the main limitations of variable importance based on impurity measures is their bias toward predictors with more levels. Additionally, they are not reliable when predictors are correlated.

▷ Example 1: Plotting variable importance

In this example, we plot variable importance after performing random forest binary classification.

We consider the churn dataset described in example 1 of [H2OML] **h2oml** and where the goal is to build a predictive model that will predict the best behavior of a customer who is more likely to churn or retain the company's services.

We start by opening the churn dataset 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/r19/churn
(Telco customer churn data)
. h2o init
  (output omitted)
. _h2oframe put, into(churn)
Progress (%): 0 100
. _h2oframe change churn
```

For convenience, we save the name of the predictors in the global macro predictors in Stata.

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

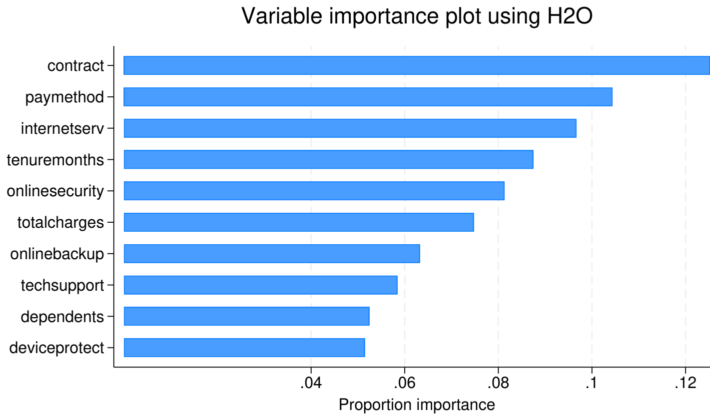
We use h2oml rfbinclass to perform random forest binary classification with 200 trees, a maximum tree depth of 3, an observation sampling rate of 0.9, and a predictor sampling value of 1. Then we use h2omlgraph varimp to plot the variable importance.

```
. h2oml rfbinclass churn $predictors, h2orseed(19) ntrees(200)
> maxdepth(3) samprate(0.9) predsampvalue(1)
Progress (%): 0 10.0 33.5 63.9 93.5 100
Random forest binary classification using H2O
Response: churn
Frame:                                  Number of observations:
  Training: churn                                   Training =  7,043
Model parameters
Number of trees      = 200
              actual = 200
Tree depth:                             Pred. sampling value =      1
           Input max =   3              Sampling rate        =     .9
                 min =   3              No. of bins cat.     =  1,024
                 avg = 3.0              No. of bins root     =  1,024
                 max =   3              No. of bins cont.    =     20
Min. obs. leaf split =   1              Min. split thresh.   = .00001
```

```
Metric summary
```

Metric	Training
Log loss	.480982
Mean class error	.2400372
AUC	.8284618
AUCPR	.6263171
Gini coefficient	.6569236
MSE	.1572825
RMSE	.3965886

```
. h2omlgraph varimp
```



The proportion of importance for the top 10 predictors is plotted. Based on this model, `contract`, `paymethod`, and `internetserv` are the three most important predictors of churn.

◁

▷ Example 2: Assessing stability of variable importance

Recent literature shows an increased attention on assessing stability of variable importance (Wang et al. 2016). In this example, we study the stability of variable importance by showing dependence of variable rankings from the predictor sampling number. That is, our goal is to vary the predictor sampling value `predsampvalue()` in random forest and explore the change in rankings of predictors based on the importance. Wang et al. (2016) implement a more extensive study and use rank-based tests to quantify stability. Our example is limited only to graphical comparison.

In the previous example, we specified a predictor sampling value of 1. Here we will compare this with the results using three other values. For convenience, we save a list of possible `predsampvalues` in the local macro `sratelist` in Stata.

```
. local sratelist 1 -1 10 -2
```

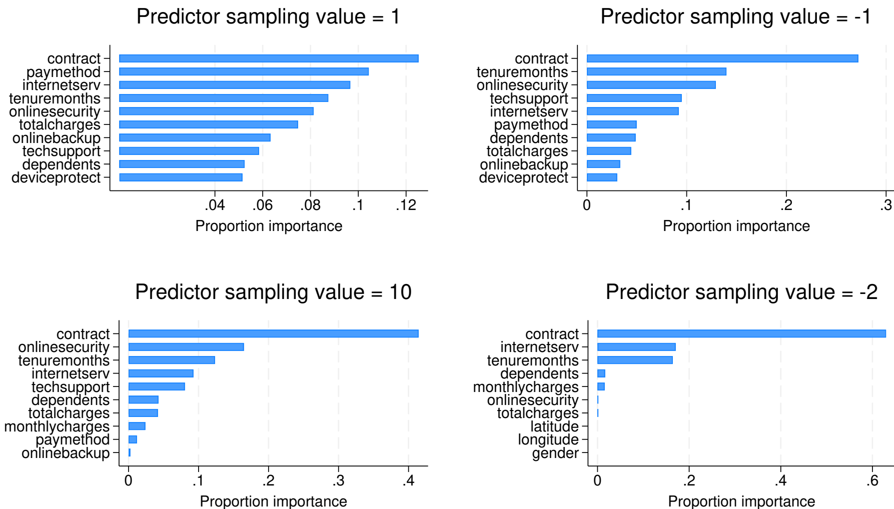
Next we use a loop to perform random forest binary classification with the predictor sampling values of $\{1, -1, 10, -2\}$, iteratively specifying each of these values in the `predsampvalue()` option of `h2oml rfbinclass`. We plot the variable importance after each estimation by using the `h2omlgraph varimp` command. Note that `predsampvalue(-2)` corresponds to selecting all predic-

tors, and `predsampvalue(-1)` corresponds to selecting the square root of the number of predictors. In `h2omlgraph varimp`, we also specify the option `saving()` to save the graphs and the option `title()` to provide a title for each graph.

```
. local i = 1
. foreach rate in `sratelist'{
  2.         quietly h2oml rfbinclass churn $predictors, h2orseed(19)
>         ntrees(200) maxdepth(3) samprate(0.9) predsampvalue(`rate')
  3.         h2omlgraph varimp, saving(imp`i', replace)
>         title("Predictor sampling value = `rate'")
  4.         local i = `i' + 1
  5. }
file imp1.gph saved
file imp2.gph saved
file imp3.gph saved
file imp4.gph saved
```

Finally, we display the saved graphs by using the `graph combine` command in Stata.

```
. graph combine imp1.gph imp2.gph imp3.gph imp4.gph
```



As the predictor sampling value changes, except for the `contract` predictor, the ranking of the importance of predictors changes substantially, indicating instability in the variable importance measure. In practice, this instability can be explained as follows: For smaller numbers of sampled predictors, predictors with smaller effects are assigned greater importance. Conversely, for larger numbers of sampled predictors, such as when all predictors are sampled with `predsampvalue(-2)`, the random forest focuses on highly influential predictors, resulting in only a few predictors considered important.

◁

References

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Boca Raton, FL: Chapman and Hall/CRC.

Wang, L., C. S. McMahan, M. G. Hudgens, and Z. P. Qureshi. 2016. A flexible, computationally efficient method for fitting the proportional hazards model to interval-censored data. *Biometrics* 72: 222–231. https://doi.org/10.1111/biom.12389.

Also see

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

h2omlpostestframe — Specify frame for postestimation analysis

Description Quick start Menu Syntax
Options Remarks and examples Stored results Also see

Description

h2omlpostestframe is a convenience command for setting an H2O frame to be used by h2oml postestimation commands to report results after h2oml *gbm* and h2oml *rf*. h2omlpostestframe does not physically change the current frame to the specified frame; see _h2oframe change.

h2omlpostestframe affects all but the following postestimation commands: h2omlestat gridsummary, h2omlselect, h2omlexplore, h2omlestat cvsummary, h2omlgraph varimp, h2omlgraph scorehistory, and h2omltree.

Quick start

Specify a generic frame named mytest to be used by postestimation commands, and label it as "Testing" in the output

```
h2omlpostestframe mytest
```

Specify a predefined validation frame to be used by postestimation commands

```
h2omlpostestframe _valid
```

Specify a frame named auto and label it

```
h2omlpostestframe auto, label(Auto dataset)
```

Switch back to the default frame specific to each postestimation command

```
h2omlpostestframe _default
```

Menu

Statistics > H2O machine learning

Syntax

Specify generic frame to be used by postestimation commands to report the results

h2omlpostestframe *framename* [, notest label(*string*)]

Specify prespecified frame to be used by postestimation commands to report the results

h2omlpostestframe *frametype* [, label(*string*)]

frametype	Description
_default	default frame; varies across commands
_train	training frame
_valid	validation frame
* _cv	cross-validation “frame”

* _cv does not correspond to an actual H2O frame; it is not applicable for some postestimation commands. See *Remarks and examples*.

label() is not allowed with _default or _cv.

Options

Options

notest specifies that the generic frame should not be considered a testing frame. By default, the specified frame is assumed to be a testing frame. This frame will be used whenever option test is specified with h2oml postestimation commands that support this option. However, if option notest is specified with h2omlpostestframe, then option test may not be used with the postestimation commands.

label(*string*) labels frame as *string* in the output.

Remarks and examples

The h2omlpostestframe command is designed to simplify machine learning postestimation analysis. If neither the cv() nor validframe() option is specified during estimation, the h2oml postestimation commands perform computations using the training frame. If the validframe() option is specified, they use the validation frame. And if the cv() option is specified, they use the cross-validation results for computation.

Sometimes, we may want to use a different frame for postestimation analysis such as a testing frame. The h2oml postestimation commands support options that allow you to specify a different frame. Alternatively, we can use the h2omlpostestframe command to specify the desired frame once for all postestimation analyses. By default, the specified frame is assumed to be a testing frame and thus will be labeled correspondingly in the output. You can use the notest option to suppress this and use the label() option to provide your own frame label.

Instead of a generic frame name, we can also specify _train, _valid, or _cv with the h2omlpostestframe command to use the respective training, validation, or cross-validation results for all postestimation analyses, provided the appropriate options were specified during estimation. The _cv specification does not correspond to an actual H2O frame and is not supported by h2omlpredict, h2omlgraph pdp, h2omlgraph ice, h2omlgraph shapvalues, and h2omlgraph shapsummary postestimation commands.

At any point during your postestimation analyses, you can specify _default to switch back to using the default frame, which is specific to each postestimation command.

Below, we demonstrate various uses of h2omlpostestframe on auto.dta.

▷ Example 1: Using h2omlpostestframe

Suppose we want to perform various postestimation analyses using the testing frame. 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. We use the _h2oframe split command to randomly split the auto frame into a training frame (80%) and a testing frame (20%), which we name train and test, respectively. We also change the current frame to train. 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/auto
(1978 automobile data)
. h2o init
. _h2oframe put, into(auto)
. _h2oframe split auto, into(train test) split(0.8 0.2) rseed(19)
. _h2oframe change train
```

Next we perform random forest binary classification using cross-validation.

```
. h2oml rfbinclass foreign price mpg length, cv(3, modulo) h2orseed(19)
  (output omitted)
```

We want to use the testing frame test for all postestimation analyses. We type

```
. h2omlpostestframe test
(testing frame test is now active for h2oml postestimation)
```

The command reported that test is assumed to be a testing frame.

Now we can use any of the postestimation commands that work with a testing frame, and the test frame will be used in computations automatically:

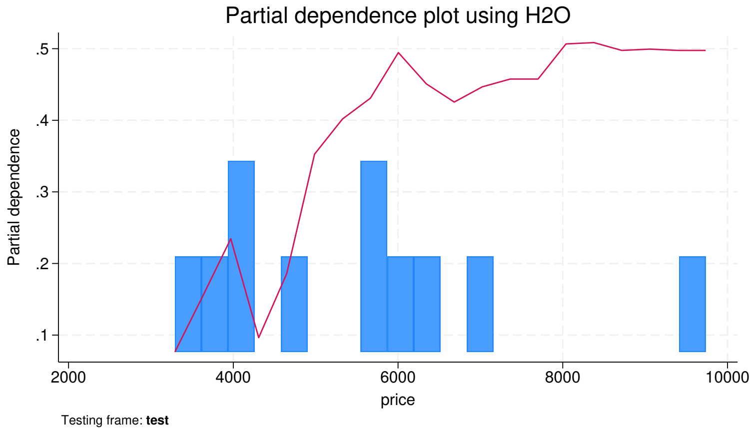
```
. h2omlestat confmatrix
Confusion matrix using H2O
Testing frame: test
```

foreign	Predicted Domestic	Foreign	Total	Error	Rate
Domestic	6	1	7	1	.143
Foreign	0	4	4	0	0
Total	6	5	11	1	.091

```
Note: Probability threshold .52 that maximizes F1 metric
      used for classification.
```

or

```
. h2omlgraph pdp price
```



And to compute predictions for the testing frame `test`, we can simply type

```
. h2omlpredict foreignhat, class
```

Note that `h2omlpostestframe` does not physically change the current frame to `test`. To access the predicted classes, we will need to change the working frame to `test` with `_h2oframe change test`.

Instead of using `h2omlpostestframe`, we could have specified the `test(test)` options with each command above. For instance, we could have typed

```
. h2omlestat confmatrix, test(test)
Confusion matrix using H2O
Testing frame: test

             |       Predicted     |
     foreign |  Domestic   Foreign |  Total   Error    Rate
-------------+---------------------+------------------------
    Domestic |         6         1 |      7       1    .143
     Foreign |         0         4 |      4       0       0
-------------+---------------------+------------------------
       Total |         6         5 |     11       1    .091
Note: Probability threshold .52 that maximizes F1 metric
      used for classification.
```

But this would require more typing.

If we need to switch back to postestimation commands using their default frames, we can specify _default instead of the frame name. For instance, because we specified the cv() option during estimation, by default, h2omlestat confmatrix would have reported the results based on cross-validation. We can still obtain these results by specifying the cv option with the command:

```
. h2omlestat confmatrix, cv

Cross-validation confusion matrix using H2O
```

foreign	Predicted Domestic	Foreign	Total	Error	Rate
Domestic	34	11	45	11	.244
Foreign	2	16	18	2	.111
Total	36	27	63	13	.206

```
Note: Probability threshold .22 that maximizes F1 metric
      used for classification.
```

Or we can use h2omlpostestframe to restore the default frame for all postestimation commands by typing

```
. h2omlpostestframe _default
(cross-validation results are now active for h2oml postestimation)
```

We can also specify one of the predefined frames with h2omlpostestframe to be used for h2oml postestimation analysis: _train to use the training frame, _valid to use the validation frame when the validframe() option is specified during estimation, and _cv to use cross-validation results when the cv() option is specified during estimation. For instance, we can type

```
. h2omlpostestframe _train
(training frame train is now active for h2oml postestimation)
```

The above is also equivalent to specifying the train option with h2omlestat confmatrix:

```
. h2omlestat confmatrix, train
  (output omitted)
```

Also, because we previously used h2omlpostestframe to define a testing frame, we can use the test option with the postestimation commands that support this option to obtain results for the testing frame:

```
. h2omlestat confmatrix, test

Confusion matrix using H2O
Testing frame: test
```

foreign	Predicted Domestic	Foreign	Total	Error	Rate
Domestic	6	1	7	1	.143
Foreign	0	4	4	0	0
Total	6	5	11	1	.091

```
Note: Probability threshold .52 that maximizes F1 metric
      used for classification.
```

◁

Stored results

h2omlpostestframe stores the following in r():

Macros

r(postest_frame)	name of the frame
r(postest_label)	frame label

Also see

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

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

h2omlselect — Select model after grid search

Description Quick start Menu Syntax
Remarks and examples Stored results Also see

Description

h2omlselect retrieves the fitted model with the hyperparameter configuration you select after h2oml *gbm* and h2oml *rf* perform tuning using a grid search. These estimation commands select the top-performing model, the one with the most optimal tuning performance metric, as the working model. After estimation, you can use h2omlestat gridsummary to see performance metrics for models with different hyperparameter configurations and to obtain an ID for each of these models. You can then select a different model to be the working model by using h2omlselect. h2omlselect selects and retrieves the fitted model; afterward, you can treat this model just as you would treat estimation results from the h2oml *gbm* and h2oml *rf* estimation commands. Subsequent postestimation commands are based on the selected model.

Quick start

After performing multiclass classification and obtaining the grid-search summary, select the model that has id = 2

```
h2oml rfmulticlass y x1-x20, ntrees(10(5)100) maxdepth(3(1)10)
h2omlestat gridsummary
h2omlselect id = 2
```

Menu

Statistics > H2O machine learning

Syntax

```
h2omlselect id = #
```

where # is a grid ID from h2omlestat gridsummary corresponding to the desired model configuration.

Remarks and examples

Building a machine learning model that generalizes well to new data involves choosing an appropriate method and selecting a model by tuning hyperparameters. We can perform a grid search using gradient boosting and random forest methods and then use h2omlestat gridsummary to report the hyperparameter configurations that achieve the top performance based on the specified metric. For example, you might use the log-loss metric to choose between models with 10, 20, and 30 trees. Typically, you would select the model that performs the best based on the chosen metric. However, you may want to explore different hyperparameter configurations that do not correspond to the best model, in which case you can use h2omlselect and h2omlexplore.

After you review the grid-search summary from `h2omlestat gridsummary`, you can select the model you are interested in by specifying the ID number with `h2omlselect`. Once you have selected a model with `h2omlselect`, you can treat the model in the same way you would treat results from the `h2oml` *gbm* and `h2oml` *rf* estimation commands. Postestimation commands will be based on the model selected by `h2omlselect`; for example, you could estimate variable importance for the selected model with `h2omlgraph varimp`. `h2omlselect` overwrites the previously stored estimation results, which can be recovered by refitting the original model or by storing the estimation results before running `h2omlselect` and then restoring them; see [H2OML] **h2omlest**.

▷ Example 1: Selecting the second-best model

In this example, we illustrate the use of `h2omlselect` by performing random forest binary classification with the social pressure dataset discussed in example 1 of [H2OML] ***h2oml rf***.

We start by opening the social pressure dataset 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 in an H2O frame, and `_h2oframe change` makes the specified frame the current H2O frame. We use the `_h2oframe split` command to randomly split the `social` frame into a training frame (80% of observations) and a validation frame (20% of observations), which we name `train` and `valid`, respectively. We also change the current frame to `train`. For details, see *Prepare your data for H2O machine learning in Stata* in [H2OML] **h2oml** and see [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 _split social, into(train valid) split(0.8 0.2) rseed(19)
. _h2oframe _change train
```

We define a global macro, predictors, to store the names of our predictors. We perform random forest binary classification, and we specify the maxdepth() and predsampvalue() options to tune the maximum tree depth and predictor sampling rate hyperparameters. For illustration, we use the area under the precision–recall curve (AUCPR) metric for tuning.

```
. global predictors gender g2000 g2002 p2000 p2002 p2004 treatment age
. h2oml rfbinclass voted $predictors, validframe(valid) h2orseed(19)
> ntrees(200) maxdepth(3(3)12) predsampvalue(-1, 1(2)8) tune(metric(aucpr))
Progress (%): 0 100
Random forest binary classification using H2O
Response: voted
Frame:                                   Number of observations:
  Training:   train                                Training = 183,607
  Validation: valid                              Validation =  45,854
Tuning information for hyperparameters
Method: Cartesian
Metric: AUCPR
```

Hyperparameters	Grid values Minimum	Maximum	Selected
Max. tree depth	3	12	6
Pred. sampling value	-1	7	7

```
Model parameters
Number of trees      = 200
              actual = 200
Tree depth:                              Pred. sampling value =      7
           Input max =   6               Sampling rate        =   .632
                 min =   6               No. of bins cat.     =  1,024
                 avg = 6.0               No. of bins root     =  1,024
                 max =   6               No. of bins cont.    =     20
Min. obs. leaf split =   1               Min. split thresh.   = .00001
Metric summary
```

Metric	Training	Validation
Log loss	.5724664	.5705699
Mean class error	.3935492	.3943867
AUC	.6705554	.6734867
AUCPR	.4658395	.4725543
Gini coefficient	.3411109	.3469735
MSE	.1946923	.1935647
RMSE	.4412395	.4399599

Next we obtain the grid-search summary by using the `h2omlestat gridsummary` command. This command lists the configuration of the hyperparameters we are tuning ranked by AUCPR.

```
. h2omlestat gridsummary
Grid summary using H2O
─────────────────────────────────────────
                           Pred.
      Max. tree    sampling
 ID       depth       value       AUCPR
─────────────────────────────────────────
  1           6           7    .4725543
  2           6           5    .4723736
  3           6           3    .4714554
  4           9           3    .4712076
  5           6          -1    .4708614
  6          12          -1    .4706606
  7           9          -1    .4705794
  8           9           5    .4689799
  9           9           7    .4682457
 10           9           1    .4674565
─────────────────────────────────────────
```

The top two models have very similar values of AUCPR, and they correspond to models with 7 and 5 randomly sampled predictors and a maximum tree depth of 6. As discussed in [H2OML] ***h2oml rf***, using a random sample of predictors improves the ability of the model to generalize to new data, compared with using the full set of predictors, because it introduces an additional randomness to the method. Therefore, we may prefer to continue our analysis with the second-best model.

To select the second-best model, we specify `id = 2` in `h2omlselect`.

```
. h2omlselect id = 2
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       = 200
              actual = 200
Tree depth:                                  Pred. sampling value =      5
           Input max =    6                  Sampling rate        =   .632
                 min =    6                  No. of bins cat.     =  1,024
                 avg = 6.0                   No. of bins root     =  1,024
                 max =    6                  No. of bins cont.    =     20
Min. obs. leaf split =    1                  Min. split thresh.   = .00001
Metric summary
──────────────────┬─────────────────────
           Metric │  Training  Validation
──────────────────┼─────────────────────
         Log loss │    .57237    .5704978
 Mean class error │  .3979593    .3945857
              AUC │   .671146    .6737527
            AUCPR │  .4670326    .4723736
 Gini coefficient │   .342292    .3475054
              MSE │  .1946602    .1935627
             RMSE │  .4412031    .4399576
──────────────────┴─────────────────────
```

Now we can continue our analysis using the second-best model.

Stored results

`h2omlselect` retrieves the selected fitted model and thus stores the same results as the estimation command used.

See *Stored results* in [H2OML] ***h2oml gbm*** or [H2OML] ***h2oml rf***.

Also see

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

[H2OML] **h2omlestat gridsummary** — Display grid-search summary

[H2OML] **h2omlexplore** — Explore models after grid search

h2omltree — Save decision tree DOT file and display rule set

Description   Quick start   Menu   Syntax
Options   Remarks and examples   References   Also see

Description

`h2omltree` saves the decision tree plot in a DOT file and returns the decision rules for a specified tree after the `h2oml` *gbm* and `h2oml` *rf* commands. For details on how to work with DOT files and convert them to images, see [H2OML] **DOT extension**.

Quick start

Save the plot of the second tree as a DOT file after regression

```
h2omltree, id(2) dotsaving(tree.dot)
```

Same as above, but report the returned results as a rule set, and replace the existing `tree.dot` file

```
h2omltree, id(2) dotsaving(tree.dot, replace) rule
```

Save the plot of the first tree as a DOT file after multiclass classification, and use the second class as the target (reference) class

```
h2omltree, target(2) dotsaving(classtree.dot, replace)
```

Same as above, but set the direction to horizontal with the tree built left to right

```
h2omltree, target(2) dotsaving(classtree.dot, replace direction(lr))
```

Menu

Statistics > H2O machine learning

Syntax

`h2omltree` [, *options*]

options	Description
* `target(`*class*`)`	specify the target class of the response variable after multiclass classification
`id(#)`	specify the number of the tree; default is `id(1)`
`rule`	report the result as a rule set
`dotsaving(`*filename* [, *saveopts*]`)`	specify that the graph be saved as *filename*

*`target()` is required for multiclass classification.

saveopts	Description
`replace`	overwrites the existing file if it already exists
`direction(`*diropts*`)`	sets the direction of tree layout; may be `tb` (the default), `bt`, `lr`, or `rl`
`titile(`*string*`)`	specifies the tree title in the DOT file

Options

target(*class*) specifies the target class of the response variable for which the decision tree DOT file is to be created. target() is required after multiclass classification with h2oml gbmulticlass and h2oml rfmulticlass.

id(#) specifies the number of the tree. The default is the first tree.

rule specifies that the tree results be reported as a rule set.

dotsaving(*filename*[, *saveopts*]) specifies that the tree be saved as *filename*. *saveopts* are the following:

replace specifies that, if the file already exists, it is okay to replace it.

direction(*diropts*) sets the direction of the tree layout. *diropts* may be one of the following:

tb specifies that the tree is built top to bottom; the default.

bt specifies that the tree is built bottom to top.

lr specifies that the tree is built left to right.

rl specifies that the tree is built right to left.

title(*string*) specifies the tree title in the DOT file.

Remarks and examples

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

Remarks are presented under the following headings:

Example 1: Plotting a classification tree after random forest
Example 2: Plotting a classification tree after gradient boosting machine (GBM)
Example 3: Plotting a regression tree
Example 4: Plotting a tree for multiclass classification

An additional example can be found in *Explaining classification prediction* of [H2OML] **h2oml**.

All decision tree plots in the examples below are produced using Graphviz (https://graphviz.org). See [H2OML] **DOT extension** for more information.

Example 1: Plotting a classification tree after random forest

We plot and interpret binary classification trees produced by random forest.

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.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile dataset)
. h2o init
  (output omitted)
. _h2oframe put, into(auto)
Progress (%): 0 100
. _h2oframe change auto
```

For simplicity, we save the predictor names in the global macro `predictors` in Stata. We then perform random forest binary classification with 100 trees and a maximum depth of 5.

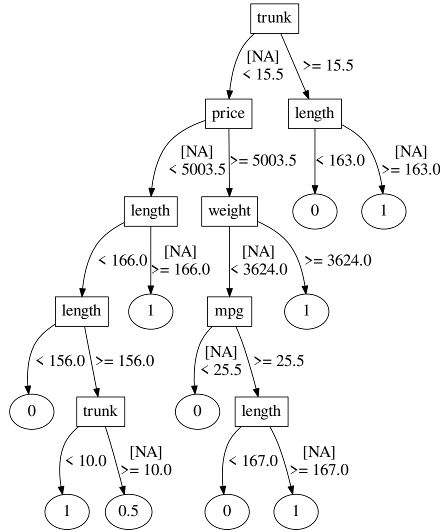
```
. global predictors price mpg trunk weight length
. h2oml rfbinclass foreign $predictors, h2orseed(19) ntrees(100) maxdepth(5)
Progress (%): 0 100
Random forest binary classification using H2O
Response: foreign
Frame:                                 Number of observations:
  Training: auto                                   Training =     74
Model parameters
Number of trees      = 100
              actual = 100
Tree depth:                            Pred. sampling value =     -1
           Input max =   5             Sampling rate        =   .632
                 min =   3             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 =   1             Min. split thresh.   = .00001
Metric summary
```

Metric	Training
Log loss	.3238765
Mean class error	.1223776
AUC	.9160839
AUCPR	.7850033
Gini coefficient	.8321678
MSE	.1089033
RMSE	.330005

Finally, we use the `h2omltree` command to save the 10th tree in the DOT file named `classtreerf.dot`.

```
. h2omltree, id(10) dotsaving(classtreerf, replace)
```

For binary classification, only the base class (the "negative" class) can be chosen as a target or reference class in H2O. In this example, this is the `Domestic` class. The tree plot shown below can be generated and saved as a PDF or another format using the information in `classtreerf.dot` and the Graphviz tool. For more details, refer to [H2OML] **DOT extension**.



The internal nodes in the tree correspond to the predictor names for which the split has occurred and the terminal nodes correspond to $P(\texttt{Domestic} = 1)$. Each internal predictor separates data based on the split. The `NA`'s on the branches indicate the split of the missing values, if any. Based on this tree, for the observations with `length` ≥ 163, the predicted probability of the car being domestic is 1.

Example 2: Plotting a classification tree after gradient boosting machine (GBM)

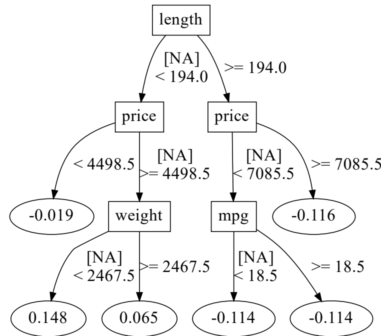
In this example, we plot a classification tree after gradient boosting binary classification. We start by running the `h2oml gbbinclass` command with options `ntrees(100)` and `maxdepth(5)`.

```
. h2oml gbbinclass foreign $predictors, h2orseed(19) ntrees(100) maxdepth(5)
```
(output omitted)

Then we use the `h2omltree` command to save the 10th tree in the DOT file named `classtreegbm.dot`

```
. h2omltree, id(10) dotsaving(classtreegbm, replace)
```

The tree below is generated from the `classtreegbm.dot` file using Graphviz.



Compared with the classification tree in *Example 1: Plotting a classification tree after random forest*, the terminal nodes of the classification tree after GBM contain negative values. This may be surprising because the expected values should be between $[0, 1]$. However, as we explain below, this is the expected behavior.

As discussed in the *Introduction* of [H2OML] ***h2oml gbm***, GBM relies on link functions to determine the loss function. For instance, in binary classification, GBM uses the logit link function. Consequently, for certain postestimation commands, such as `h2omltree` and `h2omlgraph shapvalues`, probabilities are obtained by applying the inverse link function, in this case, the inverse logit function.

For example, the predicted raw value -0.114 in the terminal node corresponds to probability 0.47153083.

```
. display invlogit(-0.114)
.47153083
```

Here the terminal nodes can be explained based on increasing or decreasing probability $P($`Domestic` $= 1)$. Thus, the highest probability corresponds to 0.148 (probability of 0.54) and occurs for the observations with length less than 194, price greater than 4498.5, and weight less than 2467.5.

Example 3: Plotting a regression tree

In this example, we create and save a DOT file and display a regression tree for random forest regression.

We start by redefining the global macro predictors. Then we perform random forest regression with 100 trees and a maximum depth of 5 for each tree.

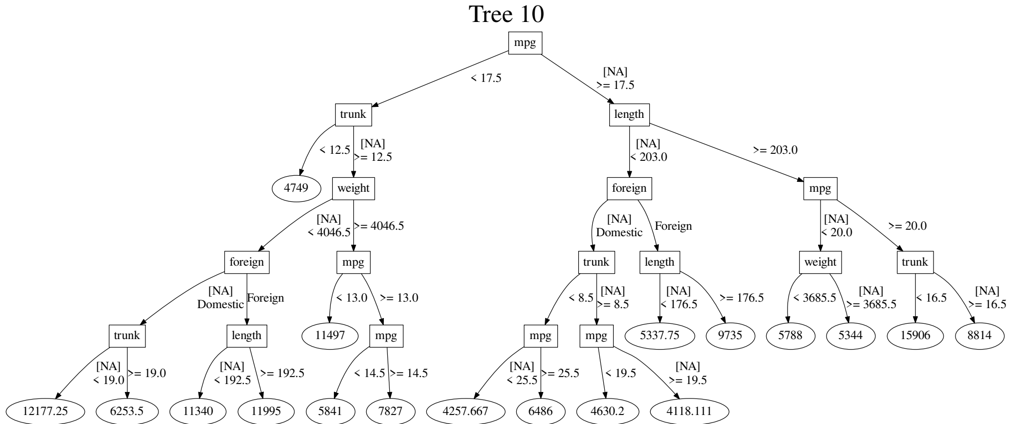
```
. global predictors foreign mpg trunk weight length

. h2oml rfregress price $predictors, h2orseed(19) ntrees(100) maxdepth(5)

Progress (%): 0 100

Random forest regression using H2O

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

Model parameters

Number of trees       = 100
             actual = 100
Tree depth:                                  Pred. sampling value =     -1
           Input max =   5                   Sampling rate        =   .632
                 min =   2                   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 =   1                   Min. split thresh.   = .00001

Metric summary
```

Metric	Training
Deviance	3129378
MSE	3129378
RMSE	1769.005
RMSLE	.2315556
MAE	1229.955
R-squared	.6353542

We save the regression tree as a DOT file by using the h2omltree command.

```
. h2omltree, id(10) dotsaving(regtreerf, replace)
```

The following tree is created from the `regtreerf.dot` file using Graphviz.



From the tree above, the predicted price for the cars with mileage per gallon less than 17.5 and trunk space less than 12.5 cu.ft. is equal to $4,749.

Example 4: Plotting a tree for multiclass classification

In this example, we create a DOT file for a tree for multiclass classification by using the `iris` dataset and random forest. 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.

```
. use https://www.stata-press.com/data/r19/iris
(Iris data)

. h2o init
  (output omitted)

. _h2oframe put, into(iris)

Progress (%): 0 100

. _h2oframe change iris
```

Next we define the global macro `predictors` to store the name of predictors and perform random forest multiclass classification.

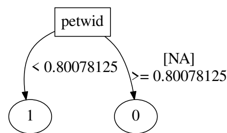
```
. global predictors seplen sepwid petlen petwid
. h2oml rfmulticlass iris $predictors, h2orseed(19) ntrees(100) maxdepth(5)
Progress (%): 0 100
Random forest multiclass classification using H2O
Response: iris                          Number of classes     =      3
Frame:                                  Number of observations:
  Training: iris                                   Training =    150
Model parameters
Number of trees      = 100
              actual = 100
Tree depth:                             Pred. sampling value =     -1
           Input max =   5              Sampling rate        =   .632
                 min =   1              No. of bins cat.     =  1,024
                 avg = 3.4              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
Log loss	.1290855
Mean class error	.06
MSE	.0370932
RMSE	.1925959

To save a tree after a multiclass classification, you must specify the option `target()` in the `h2omltree` command. Here we create a DOT file to plot the 10th tree for the class `Setosa`.

```
. h2omltree, id(10) dotsaving(mclasstreerf, replace) target(Setosa)
```

The following tree is created from the `mclasstreerf.dot` file using Graphviz.



References

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

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.

Also see

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

[H2OML] **DOT extension** — Handling DOT files

DOT extension — Handling DOT files

Description   Remarks and examples   Also see

Description

This entry provides a brief introduction to the DOT language and DOT files. These DOT files, which can be created by h2omltree, can be converted into images of decision trees.

The open source software Graphviz can be used to convert DOT files to images.

Remarks and examples

Remarks are presented under the following headings:

Install Graphviz
How to use Graphviz and DOT language
Modifying the DOT file

Install Graphviz

Graphviz is available for most operating systems. For the steps to download and install Graphviz, see https://graphviz.org/download/. If prompted during installation, you can allow Graphviz to be installed on the system path so that Graphviz commands can be issued from the terminal and issued from the Command window of Stata using the shell command. For the rest of this entry, we assume that Graphviz is installed.

How to use Graphviz and DOT language

Instead of providing extensive details of DOT language, we will explain by example and focus on options that are relevant to our goal.

First, we open the 1978 automobile data (auto.dta) in Stata and then put the data into an H2O frame. Recall that h2o init initiates an H2O cluster, _h2oframe put loads the current Stata dataset in an H2O frame, and _h2oframe change makes the specified frame the current H2O frame.

```
. use https://www.stata-press.com/data/r19/auto
(1978 automobile data)

. h2o init
  (output omitted)

. _h2oframe _put, into(auto)

. _h2oframe _change auto
```

Next, we perform gradient boosting regression and specify h2orseed(19) for reproducibility.

```
. h2oml gbregress price make mpg, h2orseed(19)
  (output omitted)
```

Finally, we use the h2omltree command to save the second tree in a file called example.dot.

```
. h2omltree, id(2) dotsaving(example.dot, replace)
```

The code below is the content of the example.dot file. You can look through the content of DOT files using your preferred text editor.

```
digraph G {
rankdir = TB
/* Level 0 */
{
"Node_0" [shape=box, fontsize=20, label="mpg"]
}
/* Level 1 */
{
"Node_9" [fontsize=20, label="286.207"]
"Node_2" [shape=box, fontsize=20, label="mpg"]
}
/* Level 2 */
{
"Node_3" [shape=box, fontsize=20, label="mpg"]
"Node_10" [fontsize=20, label="-172.209"]
}
/* Level 3 */
{
"Node_11" [fontsize=20, label="-125.564"]
"Node_6" [shape=box, fontsize=20, label="mpg"]
}
/* Level 4 */
{
"Node_12" [fontsize=20, label="15.111"]
"Node_13" [fontsize=20, label="-78.548"]
}
/* Edges */
"Node_0" -> "Node_9" [fontsize=20, label="< 17.5
"]
"Node_0" -> "Node_2" [fontsize=20, label="[NA]
>= 17.5
"]
"Node_2" -> "Node_3" [fontsize=20, label="[NA]
< 27.0
"]
"Node_2" -> "Node_10" [fontsize=20, label=">= 27.0
"]
"Node_3" -> "Node_11" [fontsize=20, label="< 20.5
"]
"Node_3" -> "Node_6" [fontsize=20, label="[NA]
>= 20.5
"]
"Node_6" -> "Node_12" [fontsize=20, label="[NA]
< 23.5
"]
"Node_6" -> "Node_13" [fontsize=20, label=">= 23.5
"]
fontsize=40
labelloc="t"
label = "Tree 2"
}
```

The file provides information about nodes of each level in the tree. For example, `Node_2` and `Node_9` belong to level 1. By default, the file provides information about the shape of the node, font size, and label. Those entries can be modified and other options can be added to describe the node. The `Edges` section in the file provides information about the structure of the tree, that is, which nodes are connected and how.

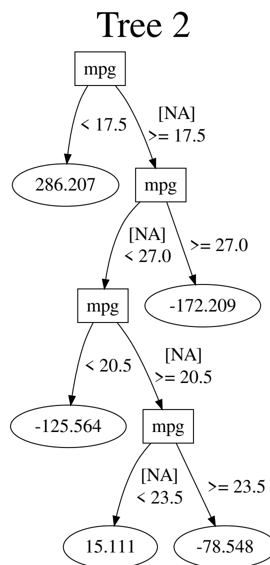
To create a PDF file with a diagram of this tree with Graphviz, we type in Stata

```
. shell dot -Tpdf example.dot -o example.pdf
```

and to create the diagram as a PNG image, we type

```
. shell dot -Tpng example.dot -o example.png
```

The `shell` command of Stata allows you to send commands to the operating system. For details, see [D] **shell**. The resulting tree is shown below.

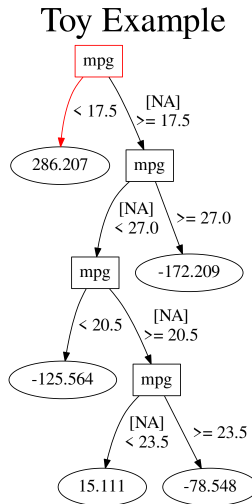


Modifying the DOT file

Having a DOT file gives us the flexibility to modify the tree based on our preference. For example, in the code below, we change the title to "Toy Example", the contour of the Node_0 to red, and the color of the left edge emanating from the Node_0 also to red. Note that the title also can be changed using the title() option in h2omltree. Changes are highlighted in bold.

```
digraph G {
rankdir = TB
/* Level 0 */
{
"Node_0" [shape=box, fontsize=20, label="mpg", color = "red"]
}
/* Level 1 */
{
"Node_9" [fontsize=20, label="286.207"]
"Node_2" [shape=box, fontsize=20, label="mpg"]
}
/* Level 2 */
{
"Node_3" [shape=box, fontsize=20, label="mpg"]
"Node_10" [fontsize=20, label="-172.209"]
}
/* Level 3 */
{
"Node_11" [fontsize=20, label="-125.564"]
"Node_6" [shape=box, fontsize=20, label="mpg"]
}
/* Level 4 */
{
"Node_12" [fontsize=20, label="15.111"]
"Node_13" [fontsize=20, label="-78.548"]
}
/* Edges */
"Node_0" -> "Node_9" [fontsize=20, label="< 17.5
", color = "red"]
"Node_0" -> "Node_2" [fontsize=20, label="[NA]
>= 17.5
"]
"Node_2" -> "Node_3" [fontsize=20, label="[NA]
< 27.0
"]
"Node_2" -> "Node_10" [fontsize=20, label=">= 27.0
"]
"Node_3" -> "Node_11" [fontsize=20, label="< 20.5
"]
"Node_3" -> "Node_6" [fontsize=20, label="[NA]
>= 20.5
"]
"Node_6" -> "Node_12" [fontsize=20, label="[NA]
< 23.5
"]
"Node_6" -> "Node_13" [fontsize=20, label=">= 23.5
"]
fontsize=40
labelloc="t"
label = "Toy Example"
}
```

The following plot depicts the changes.



Also see

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

encode_option — Encoding schemes for categorical predictors

Description Syntax Option Reference Also see

Description

The `encode()` option specifies the encoding scheme to use for categorical predictors in machine learning models implemented by the `h2oml` *gbm* and `h2oml` *rf* commands. The encoding scheme determines how a machine learning method splits categorical predictors, which can affect model performance. This entry introduces encoding schemes for categorical predictors that are available in H2O and that may be selected via the `encode()` option. For more details, see https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/algo-params/categorical_encoding.html. For an introduction to predictor encoding, see Kuhn and Johnson (2020).

Syntax

command ... [, ... `encode(`*encode_type*`)` ...]

command is one of `h2oml gbregress`, `h2oml gbbinclass`, `h2oml gbmulticlass`, `h2oml rfregress`, `h2oml rfbinclass`, or `h2oml rfmulticlass`.

encode_type	Description
`enum`	map labels of categorical predictors to integers; the default
`enumfreq`	map labels for 10 most frequent levels of each categorical predictor to integers; combine all other levels to an 11th integer
`onehotexplicit`	generate a binary predictor for each level of each categorical predictor
`binary`	convert levels of categorical predictors into binary digit representation
`eigen`	generate new predictors for a categorical predictor based on eigenvalues of the one-hot-encoding matrix
`label`	map labels of categorical predictors to integers; ensure order is preserved
`sortbyresponse`	map levels of categorical predictors to integers; order by average response within levels

Option

`encode(`*encode_type*`)` specifies the H2O encoding scheme to be used for categorical predictors. The selected encoding scheme does not modify the existing H2O frame. The predictors generated by the encoding scheme are entirely virtual; they are created at the algorithmic level rather than at the memory level. Therefore, they cannot be accessed directly. However, it can be helpful to think of the predictors as physically generated.

encode_type may be one of `enum`, `enumfreq`, `onehotexplicit`, `binary`, `eigen`, `label`, or `sortbyresponse`.

`enum` maps the labels of categorical predictors into integers, which are then used by the machine learning method for splitting decisions. For example, if a categorical predictor has the levels {cat, dog, horse, cow, turtle, unicorn}, then the `enum` option maps those levels to $\{0, 1, 2, 3, 4, 5\}$. The machine learning method may split the levels as $\{0, 2, 4\}$ and $\{1, 3, 5\}$. This is the default scheme.

`enumfreq` reduces the levels of each categorical predictor to the 10 most frequent levels. All other levels, if any, are grouped into a separate 11th level. This option is useful when the number of levels of categorical predictors is very large and some of the categories are very rare and might not provide useful information. In reporting postestimation results, this option adds suffix `.top_10_levels` to the names of the categorical predictors.

`onehotexplicit` internally generates a new binary predictor for each level of each categorical predictor. For example, if a categorical predictor has the observations {cat, dog, cat, cat, dog, unicorn, unicorn}, then three new predictors will be generated with cat $= \{1, 0, 1, 1, 0, 0, 0\}$, dog $= \{0, 1, 0, 0, 1, 0, 0\}$, and unicorn $= \{0, 0, 0, 0, 0, 1, 1\}$. This is the most well-known encoding scheme in machine learning. In reporting postestimation results, this option adds suffix `.`*level* to the names of the categorical predictors, where *level* corresponds to the class of the predictor, including missing values, which are labeled as class `NA`.

`binary` converts the levels of each categorical predictor into binary digits, with each binary digit representing a new separate predictor. The encoding process begins by assigning a numeric value to each level of the categorical predictor, starting from 1. For example, the observations of the categorical predictor {cat, dog, cat, cat, dog, unicorn, unicorn} are converted to the sequence $\{1, 2, 1, 1, 2, 3, 3\}$. The binary code for each numeric value is then determined, with 1 being represented by 01, 2 by 10, and 3 by 11. Then the observations are converted to the binary code $\{01, 10, 01, 01, 10, 11, 11\}$, with the digits of the binary number forming separate predictors. In our example, there are two new encoded predictors: $\{0, 1, 0, 0, 1, 1, 1\}$ and $\{1, 0, 1, 1, 0, 1, 1\}$. Binary encoding is useful when the number of categories is very large. However, H2O limits the number of new encoded predictors to 32. In reporting postestimation results, this option adds suffix `:#` to the names of the categorical predictors, where # varies from 1 to the maximum number of newly generated predictors. In the above example, the maximum number of generated predictors is 2.

`eigen` generates k new projected predictors per categorical predictor, such that the projections of the matrix generated from one-hot-encoding of the categorical predictor is in k-dimensional eigenspace. Currently, H2O uses $k = 1$. For details, see https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/algo-params/categorical_encoding.html. In reporting postestimation results, this option adds suffix `.Eigen` to the names of the categorical predictors.

`label` maps the labels of categorical predictors into integers, ensuring that the ordinal nature of each encoded predictor is preserved. For example, if an encoded predictor has values $\{0, 1, 2, 3, 4, 5\}$, a possible split could be $\{0, 1, 2\}$ and $\{3, 4, 5\}$, but not $\{0, 3, 4\}$ and $\{1, 2, 5\}$.

`sortbyresponse` maps the levels of categorical predictors into integers according to the ascending order of the average value of the response for each level. Thus, the level with the lowest average response value is assigned to 0, the level with second-lowest average response is assigned to 1, and so on.

Reference

Kuhn, M., and K. Johnson. 2020. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Boca Raton, FL: CRC Press.

Also see

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

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

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

Description Syntax Options References Also see

Description

The h2oml *gbm* and h2oml *rf* estimation commands allow you to specify which metric is to be used for tuning and for early stopping. In addition, h2omlestat gridsummary allows you to specify a metric for reporting; h2omlestat confmatrix allows you to specify a metric for selecting an optimal threshold for classifying predictions; and h2omlgraph scorehistory allows you to specify a metric for the y axis of the graph. In each case, you may specify the metric via a metric() option or suboption. The allowed list of metrics for each command is documented here. Available metrics vary depending on whether regression, binary classification, or multiclass classification is performed.

Syntax

In h2oml *gbm and* h2oml *rf*

command ... [, ... tune(metric(*metric*) ...)]

or

command ... [, ... stop(#, metric(*metric*) ...)]

In h2omlestat gridsummary

h2omlestat gridsummary ... [, ... metric(*metric*) ...]

In h2omlestat confmatrix

h2omlestat confmatrix ... [, ... metric(*metric_conf*) ...]

In h2omlgraph scorehistory

h2omlgraph scorehistory ... [, ... metric(*metric_score*) ...]

command is one of h2oml gbregress, h2oml gbbinclass, h2oml gbmulticlass, h2oml rfregress, h2oml rfbinclass, or h2oml rfmulticlass.

metric	Description
reg_metric	metric for regression (h2oml gbregress and h2oml rfregress)
binclass_metric	metric for binary classification (h2oml gbbinclass and h2oml rfbinclass)
multiclass_metric	metric for multiclass classification (h2oml gbmulticlass and h2oml rfmulticlass)

reg_metric	Description
* deviance	deviance
* mse	mean squared error
* rmse	root mean squared error
* rmsle	root mean squared logarithmic error
* mae	mean absolute error
r2	coefficient of determination

* indicates metrics allowed for stopping.

binclass_metric	Description
* logloss	logarithmic loss
f1	F_1 score
f2	F_2 score
fhalf	$F_{0.5}$ score
accuracy	number of correct predictions as a ratio of all predictions made
precision	proportion of correct predictions in predictions of positive class
recall	proportion of correct predictions of positive class
specificity	proportion of correct predictions in the negative class
* misclassification	number of observations incorrectly classified divided by the total number of observations
* meanclasserror	mean of per-class error rates
maxclasserror	maximum of per-class error rates
meanclassaccuracy	mean of per-class accuracy
misclasscount	total count of misclassification per class
* auc	area under the ROC curve
* aucpr	area under the precision–recall curve
* mse	mean squared error
* rmse	root mean squared error
misclasserror	synonym for misclassification
meanpcerr	synonym for meanclasserror
maxpcerr	synonym for maxclasserror
meanpcacc	synonym for meanclassaccuracy
misclasscnt	synonym for misclasscount

* indicates metrics allowed for stopping.

multiclass_metric	Description
* logloss	logarithmic loss metric
accuracy	number of correct predictions as a ratio of all predictions made
* misclassification	number of observations incorrectly classified divided by the total number of observations
* meanclasserror	mean of per-class error rates
maxclasserror	maximum of per-class error rates
meanclassaccuracy	mean of per-class accuracy
misclasscount	total count of misclassification per class
* mse	mean squared error
* rmse	root mean squared error
meanpcerr	synonym for meanclasserror
maxpcerr	synonym for maxclasserror
meanpcacc	synonym for meanclassaccuracy
misclasscnt	synonym for misclasscount

* indicates metrics allowed for stopping.

metric_conf	Description
f1	F_1 score
f2	F_2 score
fhalf	$F_{0.5}$ score
accuracy	number of correct predictions as a ratio of all predictions made
precision	proportion of correct predictions in predictions of positive class
recall	proportion of correct predictions of positive class
specificity	proportion of correct predictions in the negative class
minclassaccuracy	minimum of per-class accuracy
meanclassaccuracy	mean of per-class accuracy
tn	true negative; the number of correct predictions of the negative class
fn	false negative; the number of incorrect predictions of the negative class
tp	true positive; the number of correct predictions of the positive class
fp	false positive; the number of incorrect predictions of the positive class

tnr	true-negative rate; synonym for specificity
fnr	false-negative rate; the proportion of incorrect predictions in negative class
tpr	true-positive rate; synonym for recall
fpr	false-positive rate; the proportion of incorrect predictions in positive class
mcc	Matthews correlation coefficient
meanpcacc	synonym for meanclassaccuracy
tneg	synonym for tn
fneg	synonym for fn
tpos	synonym for tp
fpos	synonym for fp
tnegrate	synonym for tnr
fnegrate	synonym for fnr
tposrate	synonym for tpr
fposrate	synonym for fpr
mccorr	synonym for mcc

metric_score	Description
reg_metric_score	metric for regression (h2oml gbregress and h2oml rfregress)
binclass_metric_score	metric for binary classification (h2oml gbbinclass and h2oml rfbinclass)
multiclass_metric_score	metric for multiclass classification (h2oml gbmulticlass and h2oml rfmulticlass)

reg_metric_score	Description
deviance	deviance
rmse	root mean squared error
mae	mean absolute error

binclass_metric_score	Description
logloss	logarithmic loss
misclassification	number of observations incorrectly classified divided by the total number of observations
auc	area under the ROC curve
aucpr	area under the precision–recall curve
rmse	root mean squared error
misclasserror	synonym for misclassification

multiclass_metric_score	Description
logloss	logarithmic loss
misclassification	number of observations incorrectly classified divided by the total number of observations
rmse	root mean squared error
misclasserror	synonym for misclassification

Options

Options are presented under the following headings:

Metrics for regression
Metrics for classification
Additional classification metrics

Metrics are divided into those for regression and those for classification (binary and multiclass).

Metrics for regression

In the metric formulas, the ith observation is denoted by y_i, the predicted value by $\hat{y}$, the mean by $\overline{y}$, and the total number of observations by n.

`deviance` requests the deviance, which is a measurement of goodness-of-fit of the model.

With `h2oml rfregress` or with `h2oml gbregress` and the Gaussian loss, the deviance, D, is defined as

$$D = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

which is equivalent to the mean squared error (MSE).

With `h2oml gbregress` and the Tweedie loss, the deviance is defined as

$$D = \sum_{i=1}^{n} \left[\frac{\{\max(y,0)\}^{2-p}}{(1-p)(2-p)} - \frac{y(\hat{y})^{1-p}}{1-p} + \frac{(\hat{y})^{2-p}}{2-p} \right]$$

where p is the parameter in Tweedie and specified as `power()` in `h2oml` *gbm*.

With `h2oml gbregress` and the Poisson loss, the deviance is defined as

$$D = -2 \sum_{i=1}^{n} \left\{ y_i \log(\frac{y_i}{\hat{y}_i}) - (y_i - \hat{y}_i) \right\}$$

With `h2oml gbregress` and the Laplace loss, the deviance is defined as

$$D = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

which is equivalent to the mean absolute error (MAE).

`mse` requests the MSE, which is the average of the squared errors. MSE can be represented as a sum of the variance and the square of the bias. It imposes larger penalties on larger errors. Thus, it is sensitive to outliers. The formula is

$$\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

`rmse` requests the root mean squared error (RMSE). Unlike the MSE, the units of RMSE are the same as the units of the response variable, which provides a useful interpretation when the size of the error is of interest. The formula is

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

`rmsle` requests the root mean squared logarithmic error (RMSLE), which is the ratio between the logarithm actual values and the logarithm of predicted values. The RMSLE is recommended when underprediction of the model is worse than the overprediction. The formula is

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left\{\ln\left(\frac{y_i+1}{\widehat{y}_i-1}\right)\right\}^2}$$

`mae` requests the MAE, which is the average of the absolute value of the error. The units of MAE are the same as the units of the response, and it is robust to outliers. A smaller MAE indicates a better performance. The formula is

$$\frac{1}{n}\sum_{i=1}^{n}|y_i-\widehat{y}_i|$$

`r2` requests the R^2, also known as the coefficient of determination. R^2 is the proportion of the variance of a response that is explained by the predictors. Because the estimated variance depends on the given dataset, we do not advise the comparison of R^2 across different datasets. The best R^2 score is 1, and it can be negative because a model can predict arbitrarily poorly. The estimated R^2 is defined as

$$1-\frac{\sum_{i=1}^{n}(y_i-\widehat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\overline{y})^2}$$

Metrics for classification

For binary classification, suppose that y_i takes two possible values $\{0,1\}$, where 0 and 1 correspond to negative and positive classes, respectively. The predicted probability for the positive class and observation i is denoted by $\widehat{p}_i$ and the predicted class by $\widehat{y}_i$.

For multiclass classification, the number of classes is denoted by K and $y_{ik} = 1$ if the observation i belongs to the class k and 0 otherwise. The predicted probability for the observation i and class k is denoted by $\widehat{p}_{ik}$.

`logloss` requests log loss (logarithmic loss). The goal of the log loss is to estimate the closeness of the model's predicted probabilities to the actual values of the response variable. That is, log loss indicates the ability of the model to assign higher predicted probabilities to observations in the positive class and smaller probabilities to observations in the negative class. Log loss may take any nonnegative value. For binary classification, it is defined as

$$-\frac{1}{n}\sum_{i=1}^{n}y_i\ln(\widehat{p}_i)+(1-y_i)\ln(1-\widehat{p}_i)$$

For multiclass classification, it is defined as

$$-\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}y_{ik}\ln(\widehat{p}_{ik})$$

`f1`, `f2`, and `fhalf` are F_β scores and are functions of recall and precision. The F_β scores are defined as

$$F_\beta=(1+\beta^2)\frac{\text{precision}\times\text{recall}}{\beta^2(\text{precision}+\text{recall})}$$

where $\beta > 0$ is chosen such that recall is considered β times as important as precision. Here precision and recall are defined as in the descriptions of the `precision` and `recall` options.

`f1` requests F_1.

`f2` requests F_2, which is the harmonic mean of precision and recall.

`fhalf` requests $F_{0.5}$.

`accuracy` requests the accuracy, which is the ratio of the number of correct predictions to the total number of all predictions made. The accuracy metric is not recommended for imbalanced data (Bradley 1997; Huang and Ling 2005). For example, for a sample with 100 observations such that 96 belong to positive and 4 to negative classes, the accuracy score for a model that predicts the positive class for all observations is 0.96, which is misleading. The formula is

$$\frac{\text{tp} + \text{tn}}{\text{tp} + \text{tn} + \text{fp} + \text{fn}}$$

where tn and tp are the numbers of true negatives and true positives (correct predictions) and where fn and fp are the numbers of false negatives and false positives (incorrect predictions).

For multiclass classification, `accuracy_k` denotes the estimated accuracy for the class k.

`precision` requests the precision, which is the proportion of observations correctly predicted to be in the positive class out of all observations predicted to be in the positive class. Precision is a biased metric; it fails to account for the performance in negative classes (Powers 2011). The formula is

$$\frac{\text{tp}}{\text{tp} + \text{fp}}$$

`recall` requests the recall, also known as the sensitivity or the true-positive rate. It is the proportion of observations correctly predicted to be in the positive class out of all observations that actually belong to the positive class. Recall is a biased metric; it fails to account for the performance in negative classes (Powers 2011). The formula is

$$\frac{\text{tp}}{\text{tp} + \text{fn}}$$

`specificity` requests the specificity, also known as the true-negative rate. It is the proportion of correct predictions in the negative class. The formula is

$$\frac{\text{tn}}{\text{tn} + \text{fn}}$$

`misclassification` requests the misclassification, which is the proportion of the predictions that are false. It is equal to

$$1 - \text{accuracy}$$

For multiclass classification, the misclassification error for the class k is defined as

$$1 - \text{accuracy}_k$$

`misclasserror` is a synonym for `misclassification`.

`meanclasserror` requests the mean of the per-class misclassification errors. The misclassification error in class k is estimated by $1 - \text{accuracy}_k$, where accuracy_k is the accuracy for the class k. Then for K classes, the `meanclasserror` is

$$\frac{1}{K}\sum_{k=1}^{K}(1 - \text{accuracy}_k)$$

`meanpcerr` is a synonym for `meanclasserror`.

`maxclasserror` requests the maximum per-class misclassification error. For K classes, it is defined as

$$\max_{k=1,\ldots,K}\{1 - \text{accuracy}_k\}$$

`maxpcerr` is a synonym for `maxclasserror`.

`minclassaccuracy` requests the minimum per-class accuracy. For K classes, it is defined as

$$\min_{k=1,\ldots,K}\{\text{accuracy}_k\}$$

`meanclassaccuracy` requests the mean of the per-class accuracies. For K classes, it is defined as

$$\frac{1}{K}\sum_{k=1}^{K}\text{accuracy}_k$$

`meanpcacc` is a synonym for `meanclassaccuracy`.

`misclasscount` requests the total number of observations that a model has incorrectly classified. For the binary classification, it is defined as

$$\sum_{i=1}^{n} 1(y_i \neq \widehat{y}_i)$$

where $1(\cdot)$ is an indicator function and $\widehat{y}_i$ is the predicted class.

For the multiclass classification, it is defined as

$$\sum_{i=1}^{n}\sum_{k=1}^{K} 1(y_{ik} \neq \widehat{y}_{ik})$$

`misclasscnt` is a synonym for `misclasscount`.

`auc` requests the area under the curve (AUC), which measures the ability of the classification model to distinguish between true positives and false positives. A higher value indicates a better classifier. A classifier with an AUC score of 0.5 is no better than a random guess. H2O uses the trapezoidal rule to approximate the area under the receiver operating characteristic (ROC) curve. The ROC curve plots the recall against the false-positive rate. For imbalanced data, AUC is preferred more than accuracy (Bradley 1997) but less recommended than the area under the precision–recall curve (AUCPR) or the Matthews correlation coefficient (MCC).

For multiclass classification with the number of classes equal to K, there exist several variations of the AUC score.

The one-versus-one AUC (OVO AUC) calculates the AUC score for all pairwise combinations of classes. The computation of this metric requires fitting one binary classification per class pair. Thus, there are $K \times (K-1)/2$ binary classifiers.

The one-versus-rest AUC (OVR AUC) calculates the AUC score for one class with the rest of the classes. The computation of this metric requires fitting one binary classifier per class, where a given class is regarded as the "positive" class and the remaining classes are regarded as the "negative" class.

The macro average OVR AUC is a uniform weighted average of all OVR AUCs.

$$\frac{1}{K}\sum_{k=1}^{K}\text{AUC}(k, K_{-k})$$

where K is the number of classes and $\text{AUC}(j, K_{-j})$ is the AUC with class j as the positive class and the rest of classes K_{-j} as the negative class.

The weighted average OVR AUC calculates the prevalence weighted average of all OVR AUCs, where the prevalence of class k, $p(k)$, is the number of observations in class k.

$$\frac{1}{\sum_{k=1}^{K} p(k)}\sum_{k=1}^{K} p(k)\text{AUC}(k, K_{-k})$$

The macro average OVO AUC is a uniformly weighted average of all OVO AUCs

$$\frac{2}{K}\sum_{k=1}^{K}\sum_{j\neq k}^{K}\frac{1}{2}\{\text{AUC}(k, j) + \text{AUC}(j, k)\}$$

The weighted average OVO AUC is a prevalence weighted average of all OVO AUCs.

$$\frac{2}{\sum_{k=1}^{K}\sum_{j\neq k}^{K} p(k\cup j)}\sum_{k=1}^{K}\sum_{j\neq k}^{K} p(k\cup j)\frac{1}{2}\{\text{AUC}(k, j) + \text{AUC}(j, k)\}$$

`aucpr` requests the AUCPR. It is a weighted average of precision, where the weights are determined by recall at the threshold. By construction, AUCPR is more sensitive to true-positive, false-positive, and false-negative rates than AUC. Thus, it is more suitable for highly imbalanced data.

For multiclass classification, AUCPR metrics are defined similarly to the corresponding AUC metrics.

`tn` requests the true-negative metric, tn, which is the number of correct predictions of the negative class.

`tneg` is a synonym for `tn`.

`fn` requests the false-negative metric, fn, which is the number of incorrect predictions of the negative class.

`fneg` is a synonym for `fn`.

`tp` requests the true-positive metric, tp, which is the number of correct predictions of the positive class.

`tpos` is a synonym for `tp`.

`fp` requests the false-positive metric, fp, which is the number of incorrect predictions of the positive class.

`fpos` is a synonym for `fp`.

`tnr` requests the true-negative rate, which is the same as specificity.

`tnegrate` is a synonym for `tnr`.

`fnr` requests the false-negative rate, which is the proportion of incorrect predictions in the positive class. The formula is

$$\frac{\text{fn}}{\text{tp} + \text{fn}}$$

`fnegrate` is a synonym for `fnr`.

`tpr` requests the true-positive rate, which is the same as recall.

`tposrate` is a synonym for `tpr`.

`fpr` requests the false-positive rate, which is the proportion of incorrect predictions in the negative class. The formula is

$$\frac{\text{fp}}{\text{tn} + \text{fp}}$$

`fposrate` is a synonym for `fpr`.

`mcc` requests the MCC, which measures how well a binary classifier detects true and false positives, and true and false negatives. The MCC provides correlation between the actual and predicted values.

$$\frac{\text{tp} \times \text{tn} - \text{fp} \times \text{fn}}{\sqrt{(\text{tp} + \text{fp})(\text{tp} + \text{fn})(\text{tn} + \text{fp})(\text{tn} + \text{fn})}}$$

`mccorr` is a synonym for `mcc`.

Additional classification metrics

Below, we provide definitions for additional metrics that are reported by H2OML commands for classification but that need not be specified via the `metric()` option.

Gini coefficient. Often referred to as the Gini index, this estimates the "purity" of a dataset in classification problems. For a binary classification, the Gini coefficient is calculated as

$$\text{Gini} = 1 - (p_1^2 + p_2^2)$$

where p_1 and p_2 are the proportions of class 1 and 2, respectively.

$\mathbf{R^2}$ for classification. This represents the degree to which the predicted probability and the actual class move together. The best R^2 score is 1, and it can be negative because a model can predict arbitrarily poorly. For binary classification, the estimated R^2 is defined as

$$1 - \frac{\sum_{i=1}^{n}(y_i - \widehat{p}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{p}_i)^2}$$

For multiclass classification, it is defined as

$$1 - \frac{\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \widehat{p}_{ik})^2}{\sum_{i=1}^{n}\sum_{k=1}^{K}(y_i - \overline{p}_{ik})^2}$$

MSE for classification. This is the average of the squared errors, where error is the difference between the predicted probability and the actual class. For binary classification, the formula is

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{p}_i)^2$$

For multiclass classification, it is

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \widehat{p}_{ik})^2$$

RMSE for classification. This is the square root of MSE.

References

Bradley, A. P. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30: 1145–1159. https://doi.org/10.1016/S0031-3203(96)00142-2.

Huang, J., and C. X. Ling. 2005. Using AUC and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering* 17: 299–310. https://doi.org/10.1109/TKDE.2005.50.

Powers, D. M. W. 2011. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies* 2: 37–63.

Also see

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

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

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

[H2OML] **h2omlestat gridsummary** — Display grid-search summary

[H2OML] **h2omlestat confmatrix** — Display confusion matrix

[H2OML] **h2omlgraph scorehistory** — Produce score history plot

H2O option mapping — Mapping of H2OML estimation options to H2O

Description   Also see

Description

The H2OML suite of commands in Stata provides a wrapper for H2O. To facilitate the transition and clear up a potential ambiguity that you might encounter, in this entry we provide a mapping of `h2oml` *gbm* and `h2oml` *rf* option names in Stata to the H2O option names available in H2O GBM and H2O random forest. For options corresponding to hyperparameter tuning and grid search (via `h2oml`'s `tune()` option), we refer you to documentation for H2O tuning.

H2OML in Stata	H2O
* `loss()`	`distribution`
`validframe()`	`validation_frame`
`cv(#)`	`nfolds`
`cv(`*cvmethod*`)`	`fold_assignment`
`cv(`*varname*`)`	`fold_column`
`h2orseed()`	`seed`
`encode()`	`categorical_encoding`
`stop(#)`	`stopping_rounds`
`stop(metric())`	`stopping_metric`
`stop(tolerance)`	`stopping_tolerance`
`maxtime()`	`max_runtime_secs`
`scoreevery()`	`score_tree_interval`
* `monotone()`	`monotone_constraints`
`ntrees()`	`ntrees`
* `lrate()`	`learn_rate` (GBM option)
* `lratedecay()`	`learn_rate_annealing`
`maxdepth()`	`max_depth`
`minobsleaf()`	`min_rows`
* `predsamprate()`	`col_sample_rate`
† `predsampvalue()`	`mtries`
`samprate()`	`sample_rate`
`minsplitthreshold()`	`min_split_improvement`
`binscat()`	`nbins_cats`
`binsroot()`	`nbins_top_level`
`binscont()`	`nbins`
`tune(grid(`*gridspec*`))`	`strategy`
`tune(maxmodels())`	`max_models`

* indicates that the option is available only for GBM.

† indicates that the option is available only for random forest.

Also see

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

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

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

Description Also see

Description

Reproducibility is an important consideration in all scientific research, data analyses, and machine learning experiments. The goal is ensure that repeating the same analysis under the same conditions will yield identical results. In H2O, reproducibility can be affected by randomness in data splitting, model training, and the design of the machine learning method.

Below, we provide a list of guidelines to help you ensure that your analysis and results are reproducible. For more details, see H2O's reproducibility page.

1. **Control data splitting:** If you split the dataset into multiple datasets, such as training and testing sets, by using the `_h2oframe split` command, control the randomness of the splitting by setting the random-number seed with the `rseed()` option. For example, you might type

```
. _h2oframe split mydata, into(train test) split(0.8 0.2) rseed(19)
```

2. **Set a seed when fitting a model:** Gradient boosting machine (GBM) and random forest methods use random-number generation for various operations throughout estimation and grid search. For example, the observation sampling rate and column sampling rate set by the `samprate()` and `colsamprate()` options in the commands for GBM use a seed for sampling. To ensure reproducibility, set a seed via the `h2orseed()` option for both the model and the grid search. For example, you might type

```
. h2oml gbregress y x1 x2, h2orseed(19) ntrees(10(4)20)
> tune(grid(random, h2orseed(20)))
```

3. **Make sure hyperparameters are the same in every execution:** For reproducibility, the hyperparameters of the model, such as those set by the `maxdepth()`, `samprate()`, `minobsleaf()`, and other hyperparameter options, should be identical in each execution of the estimation command.

4. **Be careful with early stopping:** Early stopping, specified by the `stop()` option in GBM and random forest commands, stops the training process early when the model performance does not improve. Even though early stopping may prevent overfitting and significantly improve execution time, it is a potential source of nonreproducibility. By default, during training H2O determines an interval T, and the model performance is scored only after T trees are added to the model. In each execution of the estimation command, this default interval T can vary, which affects the scoring of the model performance, and the training may stop at different times. To ensure that the scoring of the model is consistent throughout multiple executions, specify the `scoreevery()` option with early stopping. For example, you might type

```
. h2oml gbregress y x1 x2, h2orseed(19) ntrees(100) stop(3) scoreevery(1)
```

5. **Control parallelism:** The number of machine cores, the specified number of threads during cluster initialization, and the parallelism level determine how a dataset is partitioned in memory (referred to as "chunks" by H2O) and affect the estimation of various methods, such as GBM. While H2O leverages parallelism to improve training time, this can introduce some randomness when running on multiple threads and cores.

You can limit parallelism during cluster initialization by specifying the desired number of threads using the nthread() option in the h2o.init command. For example, you can type

```
. h2o init, nthread(1)
```

However, even though nthreads() is closely related to the number of cores, in H2O this does not determine how it partitions the dataset into chunks, as this depends on the number of cores available on the machine. If the number of chunks varies, the order of operations executed by H2O will also differ. As a result, certain numeric operations may produce slightly different outcomes depending on the order of operations. This can lead to small variations in metrics sensitive to ordering, such as AUC, AUCPR, etc, when the same model with the same parameters is run in a machine with different number of cores.

The reproducibility issues described above also apply when you choose to enable parallel model building during grid search to reduce computational time. For example,

```
. h2oml gbregress y x1 x2, h2orseed(19) ntrees(100(50)200) tune(parallel(0))
```

6. **Use the same version of H2O:** A different version of H2O may contain slight differences in implementation of the method, which can affect the reproducibility. To avoid discrepancies, ensure that the same version of H2O is used each time the command is executed. The version of H2O in Stata can be checked by using the h2o query command. In the output below, the H2O version is 3.46.0.6. For details on how to download and set up H2O, see [H2OML] **H2O setup**.

```
. h2o query
Cluster is running at http://127.0.0.1:54321.
--------------------------------------------------------------------------------
H2O cluster uptime:        1 hour 0 mins
H2O cluster timezone:      America/Chicago
H2O data parsing timezone: UTC
H2O cluster version:       3.46.0.6
H2O cluster version age:   3 months
H2O cluster total nodes:   1
H2O cluster free memory:   6.892 Gb
H2O cluster total cores:   28
H2O cluster allowed cores: 28
H2O cluster status:        locked, healthy
H2O connection url:        http://127.0.0.1:54321
--------------------------------------------------------------------------------
```

Also see

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

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

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

Glossary

bagging. A model agnostic procedure that generates perturbation of the dataset by random and independent drawings (Breiman 1996).

base learner. A learner whose error rate is only slightly better than random guessing.

beeswarm plot. A type of data visualization used to display the individual data points as dots such that the points do not overlap, resulting in a "swarm" of points. This type of plot is used by `h2omlgraph shapsummary`.

bias-variance tradeoff. This controls the tension between learning and generalization. The tradeoff concerns how to lower generalization error by reducing the bias and variance of the machine learning methods. For details, see *Fundamentals of machine learning* in [H2OML] **Intro**.

black box method. A machine learning method that is difficult to interpret by design. For example, linear models and decision trees belong to the class of interpretable models, but ensemble methods, and neural networks are considered black box methods.

boosting. A model agnostic deterministic procedure that generates perturbation of the dataset by sequentially reweighting it (Freund and Schapire 1997).

categorical encoding. A process of transforming categorical predictors into numerical representations so that they can be used in machine learning models. For details, see [H2OML] ***encode_option***.

classification. A type of supervised machine learning task where the goal is to predict the category or class of a response based on predictors.

classifier. A machine learning method that is designed for classification. When the response variable in the supervised learning method is categorical, then the method implements classification.

DOT language. A plain-text graph description language used in the Graphviz software.

ensemble method. A mechanism that forms a smart committee of incompetent but carefully selected members to solve a machine learning problem. For details, see *Ensemble methods* in [H2OML] **Intro**.

explainable method. A technique used in machine learning that enables explaining the predictions of a model.

feature. Same as *predictor*.

fitting. A process of training a model on data by adjusting its hyperparameters to improve performance.

generalization. A process where the model not only performs well on the training data but also generalizes to new (testing) data.

generalization error. A quantitative measure of how well a machine learning model can predict outcomes for new (testing) data. Generalization error is the expected error on new data (the testing set).

grid search. A process of evaluating different hyperparameter configurations in the hyperparameter space to find the best configuration that improves performance of a model.

hyperparameter. A parameter whose value is adjusted to control and improve the training process.

hyperparameter space. Possible values and ranges of the hyperparameters.

hyperparameter tuning. A process where the hyperparameters of a model are optimized to improve performance.

impurity measure. A measure to quantify the goodness of fit of a split in the regression or classification trees.

k-fold cross-validation. A process of splitting a dataset into k parts. For each of k iterations, it uses one part for validation and the remaining $k-1$ parts as a training subset for model fitting.

learn. In the machine learning context, learning refers to the process when a model uses data to adjust its parameters to increase prediction accuracy.

learner. A machine learning method such as random forest and gradient boosting machine used for learning.

majority-vote rule. A classification rule that returns a class that is the most commonly occurring one among the predictors. Majority-vote rule is used in bagging and random forest to predict the class.

manifold hypothesis. The manifold hypothesis states that the observed high-dimensional data lie on a low-dimensional manifold.

metric scoring. A process of evaluating the performance of a machine learning algorithm by using a specified metric.

model agnostic. A methodology whose implementation does not directly require a particular model.

model selection. The process of building an optimal model by exploring a range of possible hyperparameters and selecting the ones that result in the best-performing model.

one-hot encoding. A process that decomposes categories of a categorical predictor into binary variables.

optimism bias. Bias that occurs when a sufficiently complex machine learning model memorizes the patterns in the training data.

out-of-bag observations. Observations that are not used to grow the tree after bootstrap.

overfitting. A process of fitting a machine learning method too well on the training data so the method fails to generalize to testing data. For details, see *Fundamentals of machine learning* in [H2OML] **Intro**.

performance metric. A quantitative measure used to evaluate the performance of a model.

pessimistic bias. Bias that occurs when the validation set is small and the machine learning model fails to reach its full capacity.

predictive modeling. A process of developing a model that generates accurate predictions.

predictor importance. The degree to which a predictor influences the model's predictions.

predictors. The inputs for a machine learning model. In classical statistics, these may be referred to as independent variables, covariates, x variables, or predictors. In machine learning literature, they are also referred to as features.

proportion predictor importance. A type of predictor importance calculated by dividing the importance of each predictor by the total sum of the importance of all predictors.

pruning. A process to optimize hyperparameters for regression and classification trees (Breiman et al. 1984).

response. The outputs for a machine learning model. In classical statistics, these may be referred to as dependent variables, y variables, or outcomes. In machine learning literature, they are also referred to as targets.

root node. A node in the graph or tree that does not have parents. For details, see *Decision trees* in [H2OML] **Intro**.

scaled predictor importance. A type of predictor importance calculated by dividing the importance of each predictor by the largest importance score of the predictors.

stopping criteria. In growing decision trees, the stopping criteria determine what will be used to halt the additional splitting of the node. Examples of stopping criteria are the depth of the tree, minimum number of observations in each tree, etc.

stump. A decision tree with depth equal to one. Stumps are weak learners.

supervised learning. A type of machine learning in which a method is trained on data where there is an associated response for each observation. Linear regression, random forest, and gradient boosting machine are examples of supervised learning.

surrogate model. An explainable model that approximates the prediction of the machine learning model.

target. See *response*.

terminal node. A node in the graph that does not have children. For details, see *Decision trees* in [H2OML] **Intro**.

testing set. New data used to estimate the generalization error of the machine learning method.

three-way holdout. A process of splitting the dataset into three parts: training, validation, and testing datasets. This method is used to evaluate model performance.

training set. Data used to train a machine learning method.

tuning budget. Time or computational resources allocated for hyperparameter tuning.

two-way holdout. A process of splitting the dataset into two parts: training and testing datasets. This method is used to evaluate model performance.

underfitting. Underfitting occurs when a machine learning model is not complex enough to capture the hidden patterns of the data, resulting in poor performance on the training and testing data.

unsupervised learning. A type of machine learning where there is no response variable.

validation dataset. A subset of data separated during the training process of a machine learning model and used to evaluate the model's performance during hyperparameter tuning.

variable importance. See *predictor importance*.

weak learner. See *base learner*.

References

Breiman, L. 1996. Bagging predictors. *Machine Learning* 24: 123–140. https://doi.org/10.1007/BF00058655.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Boca Raton, FL: Chapman and Hall/CRC.

Freund, Y., and R. E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences* 55: 119–139. https://doi.org/10.1006/jcss.1997.1504.

Subject and author index

See the combined subject index and the combined author index in the *Stata Index*.