# Gradient Boosting Machine with H2O

---

Michal Malohlava  Arno Candel

Edited by: Angela Bartz

---

http://h2o.ai/resources/

February 2021: Seventh Edition

Gradient Boosting Machine with H2O
by Michal Malohlava & Arno Candel
with assitance from Cliff Click, Hank Roark, & Viraj Parmar
Edited by: Angela Bartz

Published by H2O.ai, Inc.
2307 Leghorn St.
Mountain View, CA 94043



February 2021: Seventh Edition

Photos by ©H2O.ai, Inc.



Printed in the United States of America.

# Contents

# Introduction

This document describes how to use Gradient Boosting Machine (GBM) with H2O. Examples are written in R and Python. Topics include:

- installation of H2O
- basic GBM concepts
- building GBM models in H2O
- interpreting model output
- making predictions

# What is H2O?

H2O.ai is focused on bringing AI to businesses through software. Its flagship product is H2O, the leading open source platform that makes it easy for financial services, insurance companies, and healthcare companies to deploy AI and deep learning to solve complex problems. More than 9,000 organizations and 80,000+ data scientists depend on H2O for critical applications like predictive maintenance and operational intelligence. The company – which was recently named to the CB Insights AI 100 – is used by 169 Fortune 500 enterprises, including 8 of the world's 10 largest banks, 7 of the 10 largest insurance companies, and 4 of the top 10 healthcare companies. Notable customers include Capital One, Progressive Insurance, Transamerica, Comcast, Nielsen Catalina Solutions, Macy's, Walgreens, and Kaiser Permanente.

Using in-memory compression, H2O handles billions of data rows in-memory, even with a small cluster. To make it easier for non-engineers to create complete analytic workflows, H2O's platform includes interfaces for R, Python, Scala, Java, JSON, and CoffeeScript/JavaScript, as well as a built-in web interface, Flow. H2O is designed to run in standalone mode, on Hadoop, or within a Spark Cluster, and typically deploys within minutes.

H2O includes many common machine learning algorithms, such as generalized linear modeling (linear regression, logistic regression, etc.), Naïve Bayes, principal components analysis, k-means clustering, and word2vec. H2O implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. H2O also includes a Stacked Ensembles method, which finds the optimal combination of a collection of prediction algorithms using a process known as "stacking." With H2O, customers can build thousands of models and compare the results to get the best predictions.

H2O is nurturing a grassroots movement of physicists, mathematicians, and computer scientists to herald the new wave of discovery with data science by collaborating closely with academic researchers and industrial data scientists. Stanford university giants Stephen Boyd, Trevor Hastie, and Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. And with hundreds of meetups over the past several years, H2O continues to remain a word-of-mouth phenomenon.

**Try it out**

- Download H2O directly at `http://h2o.ai/download`.
- Install H2O's R package from CRAN at `https://cran.r-project.org/web/packages/h2o/`.
- Install the Python package from PyPI at `https://pypi.python.org/pypi/h2o/`.

**Join the community**

- To learn about our training sessions, hackathons, and product updates, visit `http://h2o.ai`.
- To learn about our meetups, visit `https://www.meetup.com/topics/h2o/all/`.
- Have questions? Post them on Stack Overflow using the **h2o** tag at `http://stackoverflow.com/questions/tagged/h2o`.
- Have a Google account (such as Gmail or Google+)? Join the open source community forum at `https://groups.google.com/d/forum/h2ostream`.
- Join the chat at `https://gitter.im/h2oai/h2o-3`.

# Installation

H2O requires Java; if you do not already have Java installed, install it from `https://java.com/en/download/` before installing H2O.

The easiest way to directly install H2O is via an R or Python package.

## Installation in R

To load a recent H2O package from CRAN, run:

```
install.packages("h2o")
```

**Note**: The version of H2O in CRAN may be one release behind the current version.

For the latest recommended version, download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in R" tab.
4. Copy and paste the commands into your R session.

After H2O is installed on your system, verify the installation:

```
library(h2o)

#Start H2O on your local machine using all available cores.
#By default, CRAN policies limit use to only 2 cores.
h2o.init(nthreads = -1)

#Get help
?h2o.glm
?h2o.gbm
?h2o.deeplearning

#Show a demo
demo(h2o.glm)
demo(h2o.gbm)
demo(h2o.deeplearning)
```

## Installation in Python

To load a recent H2O package from PyPI, run:

```
pip install h2o
```

To download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in Python" tab.
4. Copy and paste the commands into your Python session.

After H2O is installed, verify the installation:

```
import h2o

# Start H2O on your local machine
h2o.init()

# Get help
help(h2o.estimators.glm.H2OGeneralizedLinearEstimator)
help(h2o.estimators.gbm.H2OGradientBoostingEstimator)
help(h2o.estimators.deeplearning.
    H2ODeepLearningEstimator)

# Show a demo
h2o.demo("glm")
h2o.demo("gbm")
h2o.demo("deeplearning")
```

## Pointing to a Different H2O Cluster

The instructions in the previous sections create a one-node H2O cluster on your local machine.

To connect to an established H2O cluster (in a multi-node Hadoop environment, for example) specify the IP address and port number for the established cluster using the `ip` and `port` parameters in the `h2o.init()` command. The syntax for this function is identical for R and Python:

```
h2o.init(ip = "123.45.67.89", port = 54321)
```

## Example Code

R and Python code for the examples in this document are available here: `https://github.com/h2oai/h2o-3/tree/master/h2o-docs/src/booklets/v2_2015/source/GBM_Vignette_code_examples`

## Citation

To cite this booklet, use the following:

Click, C., Malohlava, M., Parmar, V., Roark, H., and Candel, A. (Feb 2021). *Gradient Boosting Machine with H2O*. http://h2o.ai/resources/.

# Overview

A GBM is an ensemble of either regression or classification tree models. Both are forward-learning ensemble methods that obtain predictive results using gradually improved estimations.

Boosting is a flexible nonlinear regression procedure that helps improve the accuracy of trees. Weak classification algorithms are sequentially applied to the incrementally changed data to create a series of decision trees, producing an ensemble of weak prediction models.

While boosting trees increases their accuracy, it also decreases speed and user interpretability. The gradient boosting method generalizes tree boosting to minimize these drawbacks.

## Summary of Features

H2O's GBM functionalities include:

- supervised learning for regression and classification tasks
- distributed and parallelized computation on either a single node or a multi-node cluster
- fast and memory-efficient Java implementations of the algorithms
- the ability to run H2O from R, Python, Scala, or the intuitive web UI (Flow)
- automatic early stopping based on convergence of user-specified metrics to user-specified relative tolerance
- stochastic gradient boosting with column and row sampling (per split and per tree) for better generalization
- support for exponential families (Poisson, Gamma, Tweedie) and loss functions in addition to binomial (Bernoulli), Gaussian and multinomial distributions, such as Quantile regression (including Laplace)
- grid search for hyperparameter optimization and model selection
- model export in plain Java code for deployment in production environments

- additional parameters for model tuning (for a complete listing of parameters, refer to the **Model Parameters** section.)

Gradient Boosting Machine (also known as gradient boosted models) sequentially fit new models to provide a more accurate estimate of a response variable in supervised learning tasks such as regression and classification. Although GBM is known to be difficult to distribute and parallelize, H2O provides an easily distributable and parallelizable version of GBM in its framework, as well as an effortless environment for model tuning and selection.

# Theory and Framework

Gradient boosting is a machine learning technique that combines two powerful tools: gradient-based optimization and boosting. Gradient-based optimization uses gradient computations to minimize a model's loss function in terms of the training data.

Boosting additively collects an ensemble of weak models to create a robust learning system for predictive tasks. The following example considers gradient boosting in the example of $K$-class classification; the model for regression follows a similar logic. The following analysis follows from the discussion in Hastie et al (2010) at `http://statweb.stanford.edu/~tibs/ElemStatLearn/`.

**GBM for classification**

1. Initialize $f_{k0} = 0, k = 1, 2, \ldots, K$
2. For $m = 1$ to $M$

    a. Set $p_k(x) = \frac{e^{f_k(x)}}{\sum_{l=1}^{K} e^{f_l(x)}}$ for all $k = 1, 2 \ldots, K$

    b. For $k = 1$ to $K$

        i. Compute $r_{ikm} = y_{ik} - p_k(x_i), i = 1, 2, \ldots, N$

        ii. Fit a regression tree to the targets $r_{ikm}, i = 1, 2, \ldots, N$, giving terminal regions $R_{jkm}, 1, 2, \ldots, J_m$

        iii. Compute

$$\gamma_{jkm} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{jkm}} (r_{ikm})}{\sum_{x_i \in R_{jkm}} |r_{ikm}|(1 - |r_{ikm}|)}, j = 1, 2, \ldots, J_m$$

iv. Update $f_{km}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jkm} I(x \in R_{jkm})$

3. Output $f_k(x) = f_{kM}(x), k = 1, 2, \ldots, K$

---

In the above algorithm for multi-class classification, H2O builds $k$-regression trees: one tree represents each target class. The index, $m$, tracks the number of weak learners added to the current ensemble. Within this outer loop, there is an inner loop across each of the $K$ classes.

Within this inner loop, the first step is to compute the residuals, $r_{ikm}$, which are actually the gradient values, for each of the $N$ bins in the CART model. A regression tree is then fit to these gradient computations. This fitting process is distributed and parallelized. Details on this framework are available at `http://h2o.ai/blog/2013/10/building-distributed-gbm-h2o/`.

The final procedure in the inner loop is to add the current model to the fitted regression tree to improve the accuracy of the model during the inherent gradient descent step. After $M$ iterations, the final "boosted" model can be tested out on new data.

## Distributed Trees

H2O's implementation of GBM uses distributed trees. H2O overlays trees on the data by assigning a tree node to each row. The nodes are numbered and the number of each node is stored as `Node_ID` in a temporary vector for each row. H2O makes a pass over all the rows using the most efficient method (not necessarily numerical order).

A local histogram using only local data is created in parallel for each row on each node. The histograms are then assembled and a split column is selected to make the decision. The rows are re-assigned to nodes and the entire process is repeated.

With an initial tree, all rows start on node 0. An in-memory MapReduce (MR) task computes the statistics and uses them to make an algorithmically-based decision, such as lowest mean squared error (MSE). In the next layer in the tree (and the next MR task), a decision is made for each row: if X $<$ 1.5, go right in the tree; otherwise, go left. H2O computes the stats for each new leaf in the tree, and each pass across all the rows builds the entire layer.

Each bin is inspected as a potential split point. The best split point is selected after evaluating all bins. For example, for a hundred-column dataset that uses twenty bins, there are 2000 (20x100) possible split points.

Each layer is computed using another MR task: a tree that is five layers deep requires five passes. Each tree level is fully data-parallelized. Each pass builds a per-node histogram in the MR call over one layer in the tree. During each pass, H2O analyzes the tree level and decides how to build the next level. In another pass, H2O reassigns rows to new levels by merging the two passes and then builds a histogram for each node. Each per-level histogram is done in parallel.

Scoring and building is done in the same pass. Each row is tested against the decision from the previous pass and assigned to a new leaf, where a histogram is built. To score, H2O traverses the tree and obtains the results. The tree is compressed to a smaller object that can still be traversed, scored, and printed.

Since the GBM algorithm builds each tree one level at a time, H2O is able to quickly run the entire level in parallel and distributed. Model building for large datasets can be sped up significantly by adding more CPUs or more compute nodes. Note that the communication requirements can be large for deep trees (not common for GBMs though) and can lead to slow model build times. The computing cost is based on a number of factors, including the final count of leaves in all trees. Depending on the dataset, the number of leaves can be difficult to predict. The maximum number of leaves is $2^d$, where $d$ represents the tree depth.

## Treatment of Factors

If the training data contains columns with categorical levels (factors), then these factors are split by assigning an integer to each distinct categorical level, then binning the ordered integers according to the user-specified number of bins `nbins_cats` (which defaults to 1024 bins), and then picking the optimal split point among the bins.

To specify a model that considers all factors individually (and perform an optimal group split, where every level goes in the right direction based on the training response), specify `nbins_cats` to be at least as large as the number of factors. For users familiar with R, values greater than 1024 are supported, but might increase model training time. (Note that 1024 represents the maximum number of levels supported in R; H2O has a limit of 10 million levels.)

The value of `nbins_cats` for categorical factors has a much greater impact on the generalization error rate than `nbins` for real- or integer-valued columns (where higher values mainly lead to more accurate numerical split points). For columns with many factors, a small `nbins_cats` value can add randomness to the split decisions (since the factor levels get grouped together somewhat arbitrarily), while large values can lead to perfect splits, resulting in overfitting.

## Key Parameters

In the above example, an important user-specified value is $N$, which represents the number of bins used to partition the data before the tree's best split point is determined. To model all factors individually, specify high $N$ values, but this will slow down the modeling process. For shallow trees, the total count of bins across all splits is kept at 1024 for numerical columns (so that a top-level split uses 1024, but a second-level split uses 512 bins, and so forth). This value is then maxed with the input bin count.

Specify the depth of the trees ($J$) to avoid overfitting. Increasing $J$ results in larger variable interaction effects. Large values of $J$ have also been found to have an excessive computational cost, since Cost $= \#\text{columns} \cdot N \cdot K \cdot 2^J$. Lower values generally have the highest performance.

Models with $4 \leq J \leq 8$ and a larger number of trees $M$ reflect this generalization. Grid search models can be used to tune these parameters in the model selection process. For more information, refer to **Grid Search for Model Comparison**.

To control the learning rate of the model, specify the `learn_rate` constant, which is actually a form of regularization. Shrinkage modifies the algorithm's update of $f_{km}(x)$ with the scaled addition $\nu \cdot \sum_{j=1}^{J_m} \gamma_{jkm} I(x \in R_{jkm})$, where the constant $\nu$ is between 0 and 1.

Smaller values of $\nu$ learn more slowly and need more trees to reach the same overall error rate but typically result in a better model, assuming that $M$ is constant. In general, $\nu$ and $M$ are inversely related when the error rate is constant. However, despite the greater rate of training error with small values of $\nu$, very small ($\nu < 0.1$) values typically lead to better generalization and performance on test data.

## Convergence-based Early Stopping

One nice feature for finding the optimal number of trees is early stopping based on convergence of a user-specified metric. By default, it uses the metrics on the validation dataset, if provided. Otherwise, training metrics are used.

- To stop model building if misclassification improves (goes down) by less than one percent between individual scoring events, specify `stopping_rounds=1`, `stopping_tolerance=0.01` and `stopping_metric="misclassification"`.
- To stop model building if the logloss on the validation set does not improve at all for 3 consecutive scoring events, specify a `validation_frame`, `stopping_rounds=3`, `stopping_tolerance=0` and `stopping_metric="logloss"`.
- To stop model building if the simple moving average (window length 5) of the AUC improves (goes up) by less than 0.1 percent for 5 consecutive scoring events, specify `stopping_rounds=5`, `stopping_tolerance=0.001` and `stopping_metric="AUC"`.
- To not stop model building even after metrics have converged, disable this feature with `stopping_rounds=0`.
- To compute the best number of trees with cross-validation, simply specify `stopping_rounds>0` as in the examples above, in combination with `nfolds>1`, and the main model will pick the ideal number of trees from the convergence behavior of the `nfolds` cross-validation models.

## Time-based Early Stopping

To stop model training after a given amount of seconds, specify `max_runtime_secs > 0`. This option is also available for grid searches and models with cross-validation. Note: The model(s) will likely end up with fewer trees than specified by `ntrees`.

## Stochastic GBM

Stochastic GBM is a way to improve generalization by sampling columns (per split) and rows (per tree) during the model building process. To control the sampling ratios use `sample_rate` for rows (per tree), `col_sample_rate_per_tree` for columns per tree and `col_sample_rate` for columns per split. All three parameters must range from 0 to 1, and default to 1.

## Distributions and Loss Functions

Distributions and loss functions are tightly coupled. By specifying the distribution, the loss function is automatically selected as well. For exponential families such as Poisson, Gamma, Tweedie, the canonical logarithmic link function is used.

For example, to predict the 80-th percentile of the petal length of the Iris dataset in R, use the following:

**Example in R**

```
library(h2o)
h2o.init(nthreads = -1)
train.hex <- h2o.importFile("https://h2o-public-test-
    data.s3.amazonaws.com/smalldata/iris/iris_wheader.
    csv")
splits <- h2o.splitFrame(train.hex, 0.75, seed=1234)
gbm <- h2o.gbm(x=1:3, y="petal_len",
        training_frame=splits[[1]],
        distribution="quantile", quantile_alpha=0.8)
h2o.predict(gbm, splits[[2]])
```

To predict the 80-th percentile of the petal length of the Iris dataset in Python, use the following:

**Example in Python**

```
import h2o
from h2o.estimators.gbm import
    H2OGradientBoostingEstimator
h2o.init()
train = h2o.import_file("https://h2o-public-test-data.
    s3.amazonaws.com/smalldata/iris/iris_wheader.csv")
splits = train.split_frame(ratios=[0.75], seed=1234)
gbm = H2OGradientBoostingEstimator(distribution="
    quantile", quantile_alpha=0.8)
gbm.train(x=range(0,2), y="petal_len", training_frame=
    splits[0])
print(gbm.predict(splits[1]))
```

# Use Case: Airline Data Classification

Download the Airline dataset from: `https://github.com/h2oai/h2o-2/blob/master/smalldata/airlines/allyears2k_headers.zip` and save the .csv file to your working directory.

## Loading Data

Loading a dataset in R or Python for use with H2O is slightly different from the usual methodology because the datasets must be converted into `H2OParsedData` objects. For this example, download the toy weather dataset from `https://github.com/h2oai/h2o-2/blob/master/smalldata/weather.csv`.

Load the data to your current working directory in your R Console (do this for any future dataset downloads), and then run the following command.

**Example in R**

```
library(h2o)
h2o.init()
weather.hex <- h2o.uploadFile(path  = h2o:::.h2o.
    locate("smalldata/junit/weather.csv"), header  =
    TRUE, sep = ",", destination_frame = "weather.hex"
    )

# Get a summary of the data
summary(weather.hex)
```

Load the data to your current working directory in Python (do this for any future dataset downloads), and then run the following command.

**Example in Python**

```
import h2o
h2o.init()
weather_hex = h2o.import_file("http://h2o-public-test-
    data.s3.amazonaws.com/smalldata/junit/weather.csv"
    )

# Get a summary of the data
weather_hex.describe()
```

## Performing a Trial Run

Load the Airline dataset into H2O and select the variables to use to predict the response. The following example models delayed flights based on the departure's scheduled day of the week and day of the month.

**Example in R**

```
# Load the data and prepare for modeling
airlines.hex <- h2o.uploadFile(path = h2o:::.h2o.
    locate("smalldata/airlines/allyears2k_headers.zip"
    ), header = TRUE, sep = ",", destination_frame = "
    airlines.hex")

# Generate random numbers and create training,
    validation, testing splits
r <- h2o.runif(airlines.hex)
air_train.hex <- airlines.hex[r  < 0.6,]
air_valid.hex <- airlines.hex[(r >= 0.6) & (r < 0.9),]
air_test.hex  <- airlines.hex[r  >= 0.9,]

myX <- c("DayofMonth", "DayOfWeek")

# Now, train the GBM model:
air.model <- h2o.gbm(y = "IsDepDelayed", x = myX,
          distribution="bernoulli",
          training_frame = air_train.hex,
          validation_frame = air_valid.hex,
          ntrees=100, max_depth=4, learn_rate=0.1)
```

**Example in Python**

```
# Now, train the GBM model:
from h2o.estimators.gbm import
    H2OGradientBoostingEstimator

# Load the data and prepare for modeling
airlines_hex = h2o.import_file("http://h2o-public-test
    -data.s3.amazonaws.com/smalldata/airlines/
    allyears2k_headers.zip")

```

```
# Generate random numbers and create training,
    validation, testing splits
r = airlines_hex.runif()    # Random UNIForm numbers,
    one per row
air_train_hex = airlines_hex[r  < 0.6]
air_valid_hex = airlines_hex[(r >= 0.6) & (r < 0.9)]
air_test_hex  = airlines_hex[r  >= 0.9]

myX = ["DayofMonth", "DayOfWeek"]

air_model = H2OGradientBoostingEstimator(
               distribution='bernoulli', ntrees=100,
               max_depth=4, learn_rate=0.1)
air_model.train(x=myX, y="IsDepDelayed",
                training_frame=air_train_hex)
```

Since it is meant just as a trial run, the model contains only 100 trees. In this trial run, no validation set was specified, so by default, the model evaluates the entire training set. To use n-fold validation, specify an n-folds value (for example, `nfolds=5`).

Let's run again with row and column sampling:

**Example in R**

```
# Load the data and prepare for modeling
airlines.hex <- h2o.uploadFile(path = h2o:::.h2o.
    locate("smalldata/airlines/allyears2k_headers.zip"
    ), header = TRUE, sep = ",", destination_frame = "
    airlines.hex")

# Generate random numbers and create training,
    validation, testing splits
r <- h2o.runif(airlines.hex)
air_train.hex <- airlines.hex[r  < 0.6,]
air_valid.hex <- airlines.hex[(r >= 0.6) & (r < 0.9),]
air_test.hex  <- airlines.hex[r  >= 0.9,]

myX <- c("DayofMonth", "DayOfWeek")

# Now, train the GBM model:
air.model <- h2o.gbm(
```

```
          y = "IsDepDelayed", x = myX,
          distribution="bernoulli",
          training_frame = air_train.hex,
          validation_frame = air_valid.hex,
          ntrees=100, max_depth=4, learn_rate=0.1,
          sample_rate=0.6, col_sample_rate=0.7)
```

**Example in Python**

```
# Now, train the GBM model:
from h2o.estimators.gbm import
    H2OGradientBoostingEstimator

# Load the data and prepare for modeling
airlines_hex = h2o.import_file("http://h2o-public-test
    -data.s3.amazonaws.com/smalldata/airlines/
    allyears2k_headers.zip")

# Generate random numbers and create training,
    validation, testing splits
r = airlines_hex.runif()   # Random UNIForm numbers,
    one per row
air_train_hex = airlines_hex[r  < 0.6]
air_valid_hex = airlines_hex[(r >= 0.6) & (r < 0.9)]
air_test_hex  = airlines_hex[r  >= 0.9]

myX = ["DayofMonth", "DayOfWeek"]

air_model = H2OGradientBoostingEstimator(
                distribution='bernoulli', ntrees=100,
                max_depth=4, learn_rate=0.1,
                sample_rate=0.6, col_sample_rate=0.7)
air_model.train(x=myX, y="IsDepDelayed",
                training_frame=air_train_hex)
```

## Extracting and Handling the Results

Now, extract the parameters of the model, examine the scoring process, and make predictions on the new data.

**Example in R**

```
# Examine the performance of the trained model
air.model

# View the specified parameters of your GBM model
air.model@parameters
```

**Example in Python**

```
# View the specified parameters of your GBM model
air_model.params

# Examine the performance of the trained model
air_model
```

The first command (`air.model`) returns the trained model's training and validation errors. After generating a satisfactory model, use the `h2o.predict()` command to compute and store predictions on the new data, which can then be used for further tasks in the interactive modeling process.

**Example in R**

```
# Perform classification on the held out data
prediction = h2o.predict(air.model, newdata=air_test.
    hex)

# Copy predictions from H2O to R
pred = as.data.frame(prediction)

head(pred)
```

**Example in Python**

```
# Perform classification on the held out data
pred = air_model.predict(air_test_hex)

pred.head()
```

## Web Interface

H2O users have the option of using an intuitive web interface for H2O, Flow. After loading data or training a model, point your browser to your IP address and port number (e.g., localhost:12345) to launch the web interface. In the web UI, click ADMIN > JOBS to view specific details about your model or click DATA > LIST ALL FRAMES to view all current H2O frames.

## Variable Importances

The GBM algorithm automatically calculates variable importances. The model output includes the absolute and relative predictive strength of each feature in the prediction task. To extract the variable importances from the model:

- **In R**: Use `h2o.varimp(air.model)`
- **In Python**: Use `air_model.varimp(return_list=True)`

To view a visualization of the variable importances using the web interface, click the MODEL menu, then select LIST ALL MODELS. Click the INSPECT button next to the model, then select OUTPUT - VARIABLE IMPORTANCES.

## Supported Output

The following algorithm outputs are supported:

- **Regression**: Mean Squared Error (MSE), with an option to output variable importances or a Plain Old Java Object (POJO) model
- **Binary Classification**: Confusion Matrix or Area Under Curve (AUC), with an option to output variable importances or a Java POJO model
- **Classification**: Confusion Matrix (with an option to output variable importances or a Java POJO model)

## Java Models

To access Java code to use to build the current model in Java, click the PREVIEW POJO button at the bottom of the model results. This button generates a POJO model that can be used in a Java application independently of H2O. If the model is small enough, the code for the model displays within the GUI; larger models can be inspected after downloading the model.

To download the model:

1. Open the terminal window.
2. Create a directory where the model will be saved.
3. Set the new directory as the working directory.
4. Follow the `curl` and `java compile` commands displayed in the instructions at the top of the Java model.

For more information on how to use an H2O POJO, refer to the **POJO Quick Start Guide** at `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/pojo-quickstart.rst`.

## Grid Search for Model Comparison

### Cartesian Grid Search

To run a Cartesian hyper-parameter grid search in R, use the following:

**Example in R**

```
ntrees_opt <- c(5,10,15)
maxdepth_opt <- c(2,3,4)
learnrate_opt <- c(0.1,0.2)
hyper_parameters <- list(ntrees=ntrees_opt,
    max_depth=maxdepth_opt, learn_rate=learnrate_opt)

grid <- h2o.grid("gbm", hyper_params = hyper_
   parameters,
    y = "IsDepDelayed", x = myX, distribution="
       bernoulli",
    training_frame = air_train.hex, validation_frame =
        air_valid.hex)
```

To run a Cartesian hyper-parameter grid search in Python, use the following:

**Example in Python**

```
#Define parameters for gridsearch
ntrees_opt = [5,10,15]
max_depth_opt = [2,3,4]
learn_rate_opt = [0.1,0.2]
hyper_parameters = {"ntrees": ntrees_opt, "max_depth": max_depth_opt,
    "learn_rate":learn_rate_opt}

from h2o.grid.grid_search import H2OGridSearch
gs = H2OGridSearch(H2OGradientBoostingEstimator, hyper_params=hyper_parameters)
gs.train(x=myX, y="IsDepDelayed", training_frame=air_train_hex,
    validation_frame=air_valid_hex)
```

This example specifies three different tree numbers, three different tree sizes, and two different shrinkage values. This grid search model effectively trains eighteen different models over the possible combinations of these parameters.

Of course, sets of other parameters can be specified for a larger space of models. This allows for more subtle insights in the model tuning and selection process, especially during inspection and comparison of the trained models after the grid search process is complete. To decide how and when to choose different parameter configurations in a grid search, refer to **Model Parameters** for parameter descriptions and suggested values.

To view the results of the grid search, use the following:

**Example in R**

```
# print out all prediction errors and run times of the models
grid
# print out the auc for all of the models
grid_models <- lapply(grid@model_ids, function(model_id) { model = h2o.getModel(model_id) })
for (i in 1:length(grid_models)) {
  print(sprintf("auc: %f", h2o.auc(grid_models[[i]])))
}
```

**Example in Python**

```
# print out all prediction errors and run times of the models
gs

# print out the auc for all of the models
for g in gs:
    print g.model_id + " auc: " + str(g.auc())
```

## Random Grid Search

If the search space is too large (i.e., you don't want to restrict the parameters too much), you can also let the Grid Search make random model selections for you. Just specify how many models (and/or how much training time) you want, and a seed to make the random selection deterministic:

**Example in R**

```
ntrees_opt <- seq(1,100)
maxdepth_opt <- seq(1,10)
learnrate_opt <- seq(0.001,0.1,0.001)
hyper_parameters <- list(ntrees=ntrees_opt,
    max_depth=maxdepth_opt, learn_rate=learnrate_opt)
search_criteria = list(strategy = "RandomDiscrete",
    max_models = 10, max_runtime_secs=100, seed
        =123456)

grid <- h2o.grid("gbm", hyper_params = hyper_
   parameters,
    search_criteria = search_criteria,
    y = "IsDepDelayed", x = myX, distribution="
        bernoulli",
    training_frame = air_train.hex, validation_frame =
         air_valid.hex)
```

**Example in Python**

```
#Define parameters for gridsearch
ntrees_opt = range(0,100,1)
max_depth_opt = range(1,20,1)
learn_rate_opt = [s/float(1000) for s in range(1,101)]
hyper_parameters = {"ntrees": ntrees_opt,
    "max_depth":max_depth_opt, "learn_rate":
       learn_rate_opt}
search_criteria = {"strategy":"RandomDiscrete",
    "max_models":10, "max_runtime_secs":100, "seed"
       :123456}

from h2o.grid.grid_search import H2OGridSearch
gs = H2OGridSearch(H2OGradientBoostingEstimator,
    hyper_params=hyper_parameters, search_criteria=
       search_criteria)
gs.train(x=myX, y="IsDepDelayed", training_frame=
   air_train_hex,
    validation_frame=air_valid_hex)
```

# Model Parameters

This section describes the functions of the parameters for GBM.

- `x`: A vector containing the names of the predictors to use while building the GBM model.
- `y`: A character string or index that represents the response variable in the model.
- `training_frame`: An `H2OFrame` object containing the variables in the model.
- `validation_frame`: An `H2OFrame` object containing the validation dataset used to construct confusion matrix. If blank, the training data is used by default.
- `nfolds`: Number of folds for cross-validation.
- `ignore_const_cols`: A boolean indicating if constant columns should be ignored. The default is `TRUE`.

- `ntrees`: A non-negative integer that defines the number of trees. The default is 50.
- `max_depth`: The user-defined tree depth. The default is 5.
- `min_rows`: The minimum number of rows to assign to the terminal nodes. The default is 10.
- `max_abs_leafnode_pred`: Limits the maximum absolute value of a leaf node prediction. The default is Double.MAX_VALUE.
- `pred_noise_bandwidth`: The bandwidth (sigma) of Gaussian multiplicative noise N(1,sigma) for tree node predictions. If this parameter is specified with a value greater than 0, then every leaf node prediction is randomly scaled by a number drawn from a Normal distribution centered around 1 with a bandwidth given by this parameter. The default is 0 (disabled).
- `categorical_encoding`: Specify one of the following encoding schemes for handling categorical features:
  - `auto`: Allow the algorithm to decide (default)
  - `enum`: 1 column per categorical feature
  - `one_hot_explicit`: N+1 new columns for categorical features with N levels
  - `binary`: No more than 32 columns per categorical feature
  - `eigen`: $k$ columns per categorical feature, keeping projections of one-hot-encoded matrix onto $k$-dim eigen space only
- `nbins`: For numerical columns (real/int), build a histogram of at least the specified number of bins, then split at the best point The default is 20.
- `nbins_cats`: For categorical columns (enum), build a histogram of the specified number of bins, then split at the best point. Higher values can lead to more overfitting. The default is 1024.
- `nbins_top_level`: For numerical columns (real/int), build a histogram of (at most) this many bins at the root level, then decrease by factor of two per level.
- `seed`: Seed containing random numbers that affects sampling.
- `sample_rate`: Row sample rate (from 0.0 to 1.0).

- `sample_rate_per_class`: Specifies that each tree in the ensemble should sample from the full training dataset using a per-class-specific sampling rate rather than a global sample factor (as with `sample_rate`. (from 0.0 to 1.0).
- `col_sample_rate`: Column sample rate (per split) (from 0.0 to 1.0).
- `col_sample_rate_change_per_level`: Specifies to change the column sampling rate as a function of the depth in the tree.
- `min_split_improvement`: The minimum relative improvement in squared error reduction in order for a split to happen.
- `col_sample_rate_per_tree`: Column sample rate per tree (from 0.0 to 1.0).
- `learn_rate`: An integer that defines the learning rate. The default is 0.1 and the range is 0.0 to 1.0.
- `learn_rate_annealing`: Reduces the `learn_rate` by this factor after every tree.
- `distribution`: The distribution function options: `AUTO`, `bernoulli`, `multinomial`, `gaussian`, `poisson`, `gamma`, `laplace`, `quantile`, `huber`, or `tweedie`. The default is `AUTO`.
- `score_each_iteration`: A boolean indicating whether to score during each iteration of model training. The default is `FALSE`.
- `score_tree_interval`: Score the model after every so many trees. Disabled if set to 0.
- `fold_assignment`: Cross-validation fold assignment scheme, if `fold_column` is not specified. The following options are supported: `AUTO`, `Random`, `Stratified` or `Modulo`.
- `fold_column`: Column with cross-validation fold index assignment per observation.
- `offset_column`: Specify the offset column. **Note**: Offsets are per-row bias values that are used during model training. For Gaussian distributions, they can be seen as simple corrections to the response (y) column. Instead of learning to predict the response (y-row), the model learns to predict the (row) offset of the response column. For other distributions, the offset corrections are applied in the linearized space before applying the inverse link function to get the actual response values.

- `weights_column`: Specify the weights column. **Note**: Weights are per-row observation weights. This is typically the number of times a row is repeated, but non-integer values are supported as well. During training, rows with higher weights matter more, due to the larger loss function pre-factor.
- `balance_classes`: Balance training data class counts via over or undersampling for imbalanced data. The default is `FALSE`.
- `max_hit_ratio_k`: (for multi-class only) Maximum number (top K) of predictions to use for hit ratio computation. To disable, enter `0`. The default is 10.
- `r2_stopping`:
  `r2_stopping` is no longer supported and will be ignored if set. Please use
  `stopping_rounds`,
  `stopping_metric` and
  `stopping_tolerance` instead.
- `stopping_rounds`: Early stopping based on convergence of `stopping_metric`. Stop if simple moving average of length k of the `stopping_metric` does not improve for k:=`stopping_rounds` scoring events. Can only trigger after at least 2k scoring events. To disable, specify `0`.
- `stopping_metric`: Metric to use for early stopping (`AUTO`: `logloss` for classification, `deviance` for regression). Can be any of `AUTO`, `deviance`, `logloss`, `misclassification`, `lift_top_gain`, `MSE`, `AUC`, and `mean_per_class_error`.
- `stopping_tolerance`: Relative tolerance for metric-based stopping criterion Relative tolerance for metric-based stopping criterion (stop if relative improvement is not at least this much).
- `max_runtime_secs`: Maximum allowed runtime in seconds for model training. Use 0 to disable.
- `build_tree_one_node`: Specify if GBM should be run on one node only; no network overhead but fewer CPUs used. Suitable for small datasets. The default is `FALSE`.
- `quantile_alpha`: Desired quantile for quantile regression (from 0.0 to 1.0) when `distribution = "quantile"`. The default is 0.5 (median, same as `distribution = "laplace"`).

- `tweedie_power`: A numeric specifying the power for the Tweedie function when `distribution = "tweedie"`. The default is 1.5.
- `huber_alpha`: Specify the desired quantile for Huber/M-regression (the threshold between quadratic and linear loss). This value must be between 0 and 1.
- `checkpoint`: Enter a model key associated with a previously-trained model. Use this option to build a new model as a continuation of a previously-generated model.
- `keep_cross_validation_predictions`: Specify whether to keep the predictions of the cross-validation models. The default is `FALSE`.
- `keep_cross_validation_fold_assignment`: Specify whether to preserve the fold assignment. The default is `FALSE`.
- `class_sampling_factors`: Desired over/under-sampling ratios per class (in lexicographic order). If not specified, sampling factors will be automatically computed to obtain class balance during training. Requires `balance_classes`.
- `max_after_balance_size`: Maximum relative size of the training data after balancing class counts; can be less than 1.0. The default is 5.
- `model_id`: The unique ID assigned to the generated model. If not specified, an ID is generated automatically.

# Acknowledgments

We would like to acknowledge the following individuals for their contributions to this booklet: Cliff Click, Hank Roark, Viraj Parmar, and Jessica Lanford.

# References

Cliff Click. **Building a Distributed GBM on H2O**, 2013. URL http://h2o.ai/blog/2013/10/building-distributed-gbm-h2o/

Thomas Dietterich and Eun Bae Kong. **Machine Learning Bias, Statistical Bias, and Statistical Variance of Decision Tree Algorithms**. 1995. URL http://www.iiia.csic.es/~vtorra/tr-bias.pdf

Jane Elith, John R. Leathwick, and Trevor Hastie. **A Working Guide to Boosted Regression Trees**. *Journal of Animal Ecology*, 77(4):802–813, 2008. URL http://onlinelibrary.wiley.com/doi/10.1111/j.1365-2656.2008.01390.x/abstract

Jerome H. Friedman. **Greedy Function Approximation: A Gradient Boosting Machine**. *Annals of Statistics*, 29:1189–1232, 1999. URL http://statweb.stanford.edu/?jhf/ftp/trebst.pdf

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. **Discussion of Boosting Papers**. *Annual Statistics*, 32:102–107, 2004. URL http://web.stanford.edu/?hastie/Papers/boost_discussion.pdf

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. **Additive Logistic Regression: a Statistical View of Boosting**. *Annals of Statistics*, 28:2000, 1998. URL http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aos/1016218223

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. **The Elements of Statistical Learning**. Springer, New York, 2001. URL http://statweb.stanford.edu/?tibs/ElemStatLearn/printings/ESLII_print10.pdf

H2O.ai Team. **H2O website**, 2021. URL http://h2o.ai

H2O.ai Team. **H2O documentation**, 2021. URL http://docs.h2o.ai

H2O.ai Team. **H2O GitHub Repository**, 2021. URL https://github.com/h2oai

H2O.ai Team. **H2O Datasets**, 2021. URL http://data.h2o.ai

H2O.ai Team. **H2O JIRA**, 2021. URL https://jira.h2o.ai

H2O.ai Team. **H2Ostream**, 2021. URL https://groups.google.com/d/forum/h2ostream

H2O.ai Team. **H2O R Package Documentation**, 2021. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html

# Authors

**Michal Malohlava**

Michal is a geek, developer, Java, Linux, programming languages enthusiast developing software for over 10 years. He obtained PhD from the Charles University in Prague in 2012 and post-doc at Purdue University. He participated in design and development of various systems including SOFA and Fractal component systems or jPapabench control system. Follow him on Twitter: @MMalohlava

**Arno Candel**

Arno is the Chief Architect of H2O, a distributed and scalable open-source machine learning platform and the main author of H2O Deep Learning. Arno holds a PhD and Masters summa cum laude in Physics from ETH Zurich, Switzerland. He has authored dozens of scientific papers and is a sought-after conference speaker. Arno was named 2014 Big Data All-Star by Fortune Magazine. Follow him on Twitter: @ArnoCandel.

**Angela Bartz**

Angela is the doc whisperer at H2O.ai. With extensive experience in technical communication, she brings our products to life by documenting the many features and functionality of H2O. Having worked for companies both large and small, she is an expert at understanding her audience and translating complex ideas to user-friendly content. Follow her on Twitter: @abartztweet.