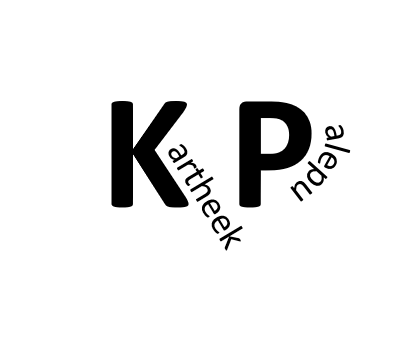


*https://www.h2o.ai*

**R - DOCUMENTATION**



1. **Overview**

* Written in **JAVA**.
* Can be implemented on top of **HADOOP** (internally it runs Map/Reduce).
* Uses Java Fork/Join for **multi-threading**.
* Uses distributed Key/Value store to access and reference data.
* Gives **REST API** call for **R** and **Python**.
* **Advantages**
  + **Fast**
  + **Distributed**
  + **In-Memory**
  + **Scalable**

1. **Installation**
   * <http://h2o-release.s3.amazonaws.com/h2o/rel-ueno/3/index.html>

*# The following two commands remove any previously installed H2O packages for R.*

if ("package:h2o" %in% search()) { detach("package:h2o", unload=TRUE) }

if ("h2o" %in% rownames(installed.packages())) { remove.packages("h2o") }

*# Next, we download packages that H2O depends on.*

if (! ("methods" %in% rownames(installed.packages()))) { install.packages("methods") }

if (! ("statmod" %in% rownames(installed.packages()))) { install.packages("statmod") }

if (! ("stats" %in% rownames(installed.packages()))) { install.packages("stats") }

if (! ("graphics" %in% rownames(installed.packages()))) { install.packages("graphics") }

if (! ("RCurl" %in% rownames(installed.packages()))) { install.packages("RCurl") }

if (! ("jsonlite" %in% rownames(installed.packages()))) { install.packages("jsonlite") }

if (! ("tools" %in% rownames(installed.packages()))) { install.packages("tools") }

if (! ("utils" %in% rownames(installed.packages()))) { install.packages("utils") }

*# Now we download, install and initialize the H2O package for R.*

install.packages("h2o", type="source", repos=(c("http://h2o-release.s3.amazonaws.com/h2o/rel-ueno/3/R")))

library(h2o)

1. **Basic Commands**
   * library(h2o)
     + load h2o in R
   * h2o.init(nthreads = -1, max\_mem\_size = '8G')
     + nthreads = -1 means it uses all cores in all CPUs
     + max\_mem\_size = memory to be used for H2O
     + **Note:** Use all CPUs and full RAM and don't run any browsers while H2O operations is happening.
   * h2o.clusterInfo()
     + prints the cluster status and information
   * h2o.shutdown(prompt = F)
     + Shutsdown the cluster
     + prompt = F won't ask for confirmation
   * as.h2o(d)
     + converts d to a h2o frame (where d = data.table or data.frame)
   * h2o.ls()
     + to check objects in h2o environment
   * h2o.importFile("full-path-of-the-file", sep = ",")
     + similar to read.csv
   * h2o.exportFile(object, "full-path-for-the-file")
     + similar to write.csv
   * h2o.merge(x, y)
   * h2o.rbind(x, y)
   * h2o.cbind(x, y)
   * h2o.table(x, y)
   * h2o.group\_by
   * h2o.runinf(100)
   * h2o.rm("***object-in-quotes***")
   * h2o.which(x == 1)
   * h2o.ifelse
   * h2o.strsplit
   * h2o.tolower
   * h2o.toupper
   * h2o.trim
     + trims leading and trailing white spaces in a string
   * h2o.gsub
   * h2o.impute(data, "column-a", method = mean/mode/median, by = c("column-b", "column-c"))
     + the above command imputes missing values in "column-a" with mean or median or mode grouped by "column-b", "column-c"
   * h2o.splitFrame(data, ratios = 0.5)
     + splits data into 2 parts of 0.5 each
     + if ratios = c(0.75, 0.15), then splits data into 3 parts 75%, 15% and 10% (train, test and validation)
2. **Modelling Commands [Explained in later sections]**
   * h2o.glm
   * h2o.gbm
   * h2o.randomForest
   * h2o.naiveBayes
   * h2o.deeplearning
   * h2o.kmeans
   * h2o.betweenss
   * h2o.centers
   * h2o.predict
   * h2o.accuracy
   * h2o.confusionMatrix
   * h2o.performance
   * h2o.mse
   * h2o.grid
3. **GLM (Generalized Linear Models)**

* Extension of traditional linear models.
* Scales well for large datasets.
* Handles classification and Prediction by using family.
* Handles Gaussian, Poisson and various other family of distributions of data.
* **In case of Linear Regression:**
  + - where,
    - Y = Target (independent variable) and
    - = Feature vector
    - = Parameter vector
    - = Intercept
    - = noise, i.e. Gaussian random variable
  + This assumes:
    - Normality of error term
    - Constant variance
    - Additivity of covariates
* Thus, we need more flexible model (Hence, we use GLM)
  + It allows variance to vary
  + Allows non-linear relationship between response and covariates.
* **Aim**:
  + For Classification: To maximize log-likelihood over parameter (B).
    - i.e.
  + For Regression: To minimize MSE.
* GLM also uses regularization and introduces parametric regularization penalty to prevent over fitting.
  + i.e.
* **Parameters of regularization:**
  + **alpha**: elastic net parameter [0, 1]
  + **Lambda**: regularization coefficient (preferred to do ***lambda\_search***)
* **Model Validation:**
  + Precision
  + Recall
  + Accuracy
  + AIC =
    - where,
      * k = no. of parameters
      * = fitted maximum likelihood
      * It is used to compute qualities of models
      * Lower the AIC value, optimal the model
      * It increases penalty with increase in number of parameters, helps in preventing over fitting
* **Regularization:**
  + Penalties of coefficients are introduced to prevent over fitting.
  + Reduces the variance of prediction error.
  + Handles correlated predictor variables.
  + **Examples**:
    - **Lasso Regression:**
      * Uses L1 Regularization
      * if lambda value is high, all coefficients are set to 0
      * Use it if the no. of features are high/correlated (Selects only one among correlated variable and sets other to 0)
      * Helps in reducing sparsity
    - **Ridge Regression:**
      * Uses L2 Regularization
      * Helps in achieving numerical stability
      * Shrinks all parameters proportionally
      * Reduces coefficients as the penalty increases without setting any value to 0
      * Reduces coefficients if the features are correlated.
    - **Elastic Net:**
      * Uses L1, L2 Regularization
      * has alpha, lambda
      * where:
        + alpha [0, 1] - elastic penalty distribution
        + lambda (> 0) - Penalty Strength

|  |  |  |
| --- | --- | --- |
| **lambda** | **alpha** | **results** |
| 0 | any value | no regularization |
| > 0 | 0 | ridge regression |
| > 0 | 1 | lasso regression |
| > 0 | (0, 1) | elastic net |

* + - * Handles sparsity and achieves stability
* **Model Families:**
  + **Gaussian (Regression)**
    - h2o.glm(x, y, training\_frame = df, family = **'gaussian'**)
  + **Binomial (Logistic regression)**
    - h2o.glm(x, y, training\_frame = df, family = **'binomial'**)
  + **Multinomial (multi-class classification)**
    - h2o.glm(x, y, training\_frame = df, family = **'multinomial'**)
  + **Poisson (if Y >= 0 and errors have possion distribution)**
    - h2o.glm(x, y, training\_frame = df, family = **'poisson'**)
  + **Gamma (if Y > 0)**
    - h2o.glm(x, y, training\_frame = df, family = **'gamma'**)
  + **Tweedie** (includes gamma, normal, poisson, and their combination, and for Y >= 0)
* **Paramerters in h2o.glm:**
  + training\_frame
  + validation\_frame
  + lambda
    - = **0** as default
  + lambda\_search
    - use **TRUE**, for searching optimal lambda value
  + max\_active\_predictors
    - uses when lambda\_search = TRUE
    - if = **10**, then stops after having 10 predictor values
  + remove\_collinear\_columns
    - use **TRUE** to remove collinear columns
  + standardize
    - use **TRUE** to scale the data to 0 mean and unit variance
  + compute\_p\_values
    - use **TRUE** to compute p values
    - used for hypothesis testing
    - higher the p values, unreliable the features are
    - lower the significant
  + nfolds
    - = **5** (does 5 fold cross validation)
    - to **print** use,
    - fit@model$training\_metrics@metrics$AUC
    - fit@model$cross\_validation\_metrics@metrics$AUC
  + **grid search over alpha:**

alphas = list(list(0), list(0.25), list(0.5), list(0.75), list(1))

hyper\_params = list(alpha = alphas)

grid = h2o.grid(model = 'glm', **hyper\_params = hyper\_params**, x, y, training\_frame = df, family = 'binomial')

models = lapply(grid@model\_ids, function(each\_model) {

h2o.getModel(each\_model)

})

print(models[[1]]@model$model\_summary$regularization)

print(h2o.auc(models[[1]]))

* **Model Statistics:**
  + h2o.mse
  + h2o.rmse
  + h2o.r2
  + h2o.auc
  + h2o.aic
  + h2o.logloss
  + h2o.null\_deviance
    - deviance from NULL model
  + h2o.residual\_deviance
    - deviance of built model
  + print(fit@model$coefficients)
    - prints model coefficients
  + print(fit@model$scoring\_history)
    - prints likelihood, iteration, objective scores at different timestamps and iterations
  + h2o.confusionMatrix(fit, valid = **T**)
    - valid = for validation data flag (TRUE or FALSE)
  + h2o.predict(fit, newdata = test)
    - used for prediction on new data set
  + h2o.performance(fit, newdata = test)
    - used to calculate different performance measures on the new data

1. **Random Forest**

* Distributed in case of H2O
* Multiple Trees (multiple weak learners)

h2o.randomForest(model\_id ='rf-model', training\_frame, validation\_frame, nfolds, x, y

, ntrees # grid search (10 -500)

, max\_depth # grid search seq(1, 29, 2)

, min\_rows # default = 10, 500 means it needs 500 TRUE and 500 FALSE to make split

# grid search seq(1, 20, 1)

, min\_split\_improvement # default = 10^-5

# grid search (10^-10 .... 10^-3)

, nbins # bin numeric value, grid search (8, 16, 32, 64, 128, 256, 512)

, nbin\_cats # bins for categorical, grid search (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)

, nbins\_top\_level # bins to use at top level of trees

# nbins, nbin\_cats, nbins\_top\_level less values less the overfitting

, stopping\_rounds # for specific no.of rounds if there is no improvement, then stop

, stopping\_metric # logloss, rmse, auc, mse

, stopping\_tolerance # 1e-3

, seed # to reproduce

, histogram\_type

# AUTO

# UniformAdaptive

# Random

# QuartilesGlobal

# Round Robin

, sample\_rate # (0, 1)

# - improves generalization

# - for large datasets use 0.7, 0.8, 1

# - for imbalanaced

, sample\_rate\_per\_class

# - c(1, 0.5) = down samples class 2 by 50%

, col\_sample\_rate # (0, 1)

# - improves generalization

# - reduces validation error

# - no. of cols to consider for each split

, col\_sample\_rate\_per\_tree

# if no. of cols (n) = 100

# - if col\_sample\_rate\_per\_tree = 0.75

# - and if col\_sample\_rate = 0.8

# - then for each tree = 0.75 \* 100 = 75 cols are considered

# - and for each split = 75 \* 0.8 = 60 cols are considered

, mtries # (-1 or >= 1)

# - no. of cols to select randomly at each level

# - grid search for best value)

1. **K Means**

h2o.kmeans(x, k = 3, training\_frame, validation, seed

, init

# Random - chooses k points randomly

# Furthest - Choose m1, calculate distance from all N-1 Points, choose far one, repeat till K

# PlusPlus - Choose m1, calculate distance from all N-1 Points, score by giving weights as distances,

# now pick m2 randomly out of weighted probability, repeat till K

# User - User defined centers

, estimate\_k # flag that helps and builds till <= K

, max\_iterations # 1 - 10^6

, standardize = TRUE)

* Determine K
  + use estimate\_k parameter
  + internatlly it calculates:
    - PRE = (SSW[after split] - SSW[before split])/SSW[before split]
  + if PRE < threshold, stop
  + where, threshold = min(0.8, 0.02 + (10/# of rows) + (2.5/ (# of features)^2))

1. **PCA**

* transforms correlated features to uncorrelated features (principle components)
* maximizes variances, reduces covariance
* used for dimension reduction
* makes all dimensions as orthogonal
* used before KMeans for best results
* Categorical - internally it converts to one hot encoding

h2o.prcomp(model\_id, training\_frame

, transform

# none

# standardize

# normalize

# Demean

# Descale

, seed)

1. **GLRM**

* Generalized Low Rank Models
* Dimensional Reduction
* Useful to reconstruct missing values
* Find important features
* Parallel and optimized
* same use as SVD, Matrix factorization
  + A [m x n] = X [m x k] %\*% Y [k x n]
* Choose k = lower the better compressed

h2o.glrm(training\_frame, cols = 1:ncol(data), k = 10)

1. **Load and Save Models**

h2o.loadModel(path)

h2o.saveModel(model, dir, name)

1. **GBM**

* Gradient Boosting Model
* Gradually improved estimations
* Models Non-Linear relationships
* Weak classification algorithms are sequentially applied to the incrementally changed data to create a series of decision trees, producing an ensemble
* In H2O
  + Distributed and Parallelized
  + Fast and memory efficient
  + Uses stochastic gradient descent with column and row sampling
* Boosting is optimized by Gradient descent to minimize a model loss
* **Steps**
  + - 1. fit a model to the data F1(x) = Y
      2. fit a model to the residuals (Y - Y')

h1(x) = Y - F1(x)

* + - 1. create a model F2(x) = F1(x) + h1(x)

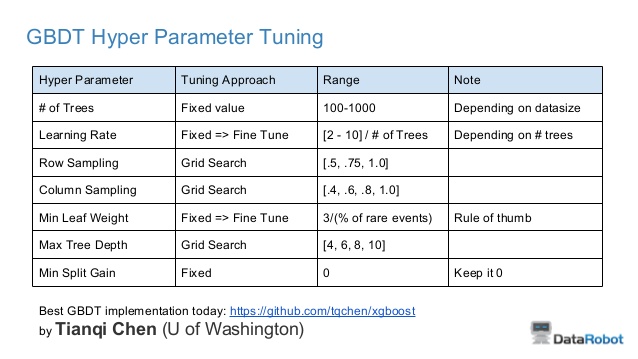
i.e.

F(x) = F1(x) -> F2(x) - - - - - -> FM(x)

where FM(x) = FM-1(x) + hM-1(x)

and h1(x) weak learner (a decision stump)

* **Main Parameters**
  + eta = 0.3 or 0.1
  + max\_depth = 3
  + n\_estimators = 100
  + subsample = 1
* **Tips by Owen Zhang (Kaggle #2) and Tianqi Chen**
  + Tune learning rate by using fixed values of trees
  + Use random search over grid search if search space is too large



* **Codes**

fit = h2o.gbm(x, y, training\_frame, validation\_frame, nfolds

, ntrees # default = 50

, max\_depth # default = 5

, min\_rows # defualt = 10

, min\_split\_improvement

, nbins

, nbin\_cats

, seed

, learn\_rate # (0, 1)

# - lower the better but more time to reach minima

# - use 0.05, with 0.99 learn\_rate\_annealing

, learn\_rate\_annealing

# - reduces learning rate by this factor for every tree

# - start with learn\_rate

# - ends with learn\_rate \* (learn\_rate\_annealing) ^ N

# - instead of LR = 0.01, use LR = 0.05, LRA = 0.99 - executes faster

# - use = 1 to disable, but we should use low learn\_rate

, distribution # loss function

# - AUTO - if selected and if y == numeric, it selects gaussian, else bernoulli

# - bernoulli - 2 class

# - multinomial - multi class

# - gaussian - numeric

# - poisson

# - gamma

# - laplace

# - quantile

# - huber

# - tweedie

, sample\_rate

, sample\_rate\_per\_class

, col\_sample\_rate

, col\_sample\_rate\_per\_tree

, score\_each\_iterator # Flag

, score\_tree\_interval # if = 5, scores every 5 trees

, weights\_column # no. of times each row to be repeated

, balance\_classes # flag

, class\_sampling\_factors # c(0.5, 1) = reduces class - 1 by 50%

, max\_after\_balance\_size # if 0.85 reduces the total data size to 85%

# if 1.7 then it increases the total data size to 117%

, stopping\_rounds

, stopping\_metric

, stopping\_tolerance

, max\_runtime\_secs

, checkpoint

# used to continue model building

)

h2o.varimp(fit)

1. **Deep Learning**

* Neural Networks that are having more hidden layers
* Fast, memory efficient
* Multi-threaded, parallel and distributed
* Implemented in Java
* **Adaptive Learning rate**
  + if errorN > errorN-1
  + alpha = alpha/2
* Regularization - L1 (lambda1 \* Sigma(W)), L2 (lambda2 \* Sigma(W^2)), dropout, model averaging, HOGWILD!
* Can have learning rate, momentum
* Auto encoders for unsupervised
* **Activation Functions**
  + tanh: (exp(x) - exp(-x))/(exp(x) + exp(-x))
  + makes the value of x in range of [-1, 1]
  + relu: max(0, x)
  + rectified linear unit
  + maxout: max(x1, x2)
* **Loss functions:**
  + for gaussian - use mse
  + for laplace - use absolute error
  + for bernoulli - use cross entropy
* **Sample:**

h2o.deeplearning(x, y, training\_frame, validation\_frame, distribution = 'bernoulli')

* **Loss minimization:**
  + Uses SGD to parallelize via back propagation
  + It also uses HOGWILD! to parallelize as SGD can’t be parallelized completely
  + HOGWILD! is lock free parallelization scheme
  + HOGWILD! uses shared memory when multiple cores handle multiple subsets of data
* **Other Optimizations:**
  + L1
  + L2
  + dropout
    - input\_dropout\_ratios
    - hidden\_dropout\_ratios
  + momentum (0, 1)
    - helps in avoiding local minima
    - too much momentum leads instability
    - momentum\_start
    - momentum\_ramp
    - momentum\_stable
    - **nesterov\_accelerated\_gradient** # recommended
    - Large momentum + low learning rate is better
    - damps oscillation in the direction of high curvature
  + Rate Annealing
    - 10^-6
    - i.e. it takes 10^-6 samples to change learning rate to half
  + Adaptive Learning
    - avoids slow convergence
    - adds benefits of momentum + rate\_annealing
    - rho -> momentum (0.9 or 0.9999)
    - epsilon -> rate annealing (10^-10 or 10^-4)
* **Code**:

h2o.deeplearning(x, y, training\_frame, validation\_frame, seed, nfolds

, activation

# Tanh

# Tanhwithdropout

# Rectifier

# Rectifierwithdropout

# maxout

# maxoutwithdropout

, hidden # c(100, 100) - 2 hidden layerss with 100 neurons each

, epochs # no. of times to iterate through the data

, variable\_importance # flag

, weight\_columns

, balance\_classes

, class\_sampling\_factors

, max\_after\_balance\_size

, standardize

, checkpoint

, adaptive\_rate # flag

, input\_dropout\_ratio # (0.1 or 0.2 suggested)

, hidden\_dropout\_ratios # [0, 1) - default = 0.5

, l1

, l2

, loss

# - Automatic

# - CrossEntropy

# - Quadratic

# - Huber

# - Absolute

, distribution

, score\_interval

, stopping\_rounds

, stopping\_metric

, stopping\_tolerance

, autoencoder # flag

, rho # adaptive rate = T, then use this - adaptive rate time decay factor

, epsilon # adaptive rate time smoothing factor

, shuffle\_training\_data

, missing\_values\_handling # Skip/MeanImputation

, rate

, rate\_annealing

, rate\_decay

, nesterov\_accelerated\_gradient # flag

, momentum\_start # 0.5 is preferred

, momentum\_ramp

, momentum\_stable)

* **Tuning:**
  + Use hidden:
    - [200, 200]
    - [512]
    - [64, 64, 64]
    - [32, 32, 32, 32, 32]
  + Use L1, L2, dropout, adaptive rate with rho, epsilon

1. **Stacked Ensembles**

* **Steps:**
  + Specify a list of L base algorithms with model parameters
  + Specify a meta learning algorithm
  + Train L models on training
  + Predict on validation set (N rows)
  + we get N x L matrix (level one data)
  + Use the level one data to train a super meta learner
  + Predict on test data
* **Code:**

fit = h2o.stackedEnsemble(x, y, training\_frame, model\_id, base\_models = list(fit1@model\_id, fit2@model\_id))

fit = h2o.stackedEnsemble(x, y, training\_frame, model\_id, base\_models = list(gbm\_grid\_fit@model\_ids))

* **Using h2oEnsemble**

library(h2oEnsemble)

learner = c('h2o.randomForest', 'h2o.deeplearning.1', 'h2o.deeplearning.2')

metalearner = 'h2o.glm.wrapper'

fit = h2o.ensemble(x, y, training\_frame, family = 'binomial', learner = learner,

metalearner = metalearner, cvControl = list(V = 5))

h2o.ensemble\_performance(fit, newdata)

predict(fit, newdata)

* **Default Wrappers:**
  + h2o.glm.wrapper
  + h2o.gbm.wrapper
  + h2o.randomForest.wrapper
  + h2o.deeplearning.wrapper

1. **H2O Parameter Tuning**

* hyper\_parameters = list(.....)
* **Grid Search**

grid = h2o.grid('gbm', x, y, training\_frame, hyper\_params = hyper\_parameters)

model = lapply(grid@model\_ids, function(x) {

h2o.getModel(x)

})

* **Random Search**

search\_criteria = list(strategy = '**RandomDiscrete**', max\_runtime\_sec = 600, max\_models = 100, stopping\_metric = 'AUTO', stopping\_tolerance = 0.0001, stopping\_rounds = 5, seed = 1234)

grid = h2o.grid(grid\_id = 'random\_search', search\_criteria = search\_criteria, x, y, training\_frame, 'gbm', hyper\_params = hyper\_parameters)

h2o.getGrid(grid\_id = 'random\_search', sort\_by = 'rmse')