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**

2. 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)
```

3. 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)

4. 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`

5. 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:**
  - $Y = X^T\beta + \beta_0 + \epsilon$
    - where,
    - Y = Target (independent variable) and $Y \sim N(X^T\beta + \beta_0, \sigma^2)$
    - X^T = Feature vector
    - β = Parameter vector
    - β_0 = Intercept
    - ϵ = noise, $\epsilon \sim N(0, \sigma^2)$ 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. $\max_{\beta}(\log likelihood)$
  - For Regression: To minimize MSE.
- GLM also uses regularization and introduces parametric regularization penalty to prevent over fitting.
  - i.e. $\max_{\beta}(loglikelihood - reg\ penalty)$

- **Parameters of regularization:**
  - **alpha**: elastic net parameter [0, 1]
  - **Lambda**: regularization coefficient (preferred to do ***lambda_search***)

- **Model Validation:**
  - Precision
  - Recall
  - Accuracy
  - AIC = $2k - 2\log(l(y;\ \mu))$
    - where,
      - k = no. of parameters
      - $l(y;\ \mu)$ = 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
  - $Deviance = (max.loglikelihood.of.fitted - max.loglikelihood.of.saturated.models)$

- **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
      - $||B||_1 = \sum_{k=1}^{n} |B_k|$
      - 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
    - $||B||_2^2 = \sum_{k=1}^{n} B_k^2$
    - 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

```
o  h2o.performance(fit, newdata = test)
```

- used to calculate different performance measures on the new data

6. 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)
```

7. 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))

8. 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)
```

9. 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)
```

10. Load and Save Models

```
h2o.loadModel(path)

h2o.saveModel(model, dir, name)
```

11. 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**

    a) fit a model to the data F1(x) = Y

    b) fit a model to the residuals (Y - Y')

    ```
    h1(x) = Y - F1(x)
    ```

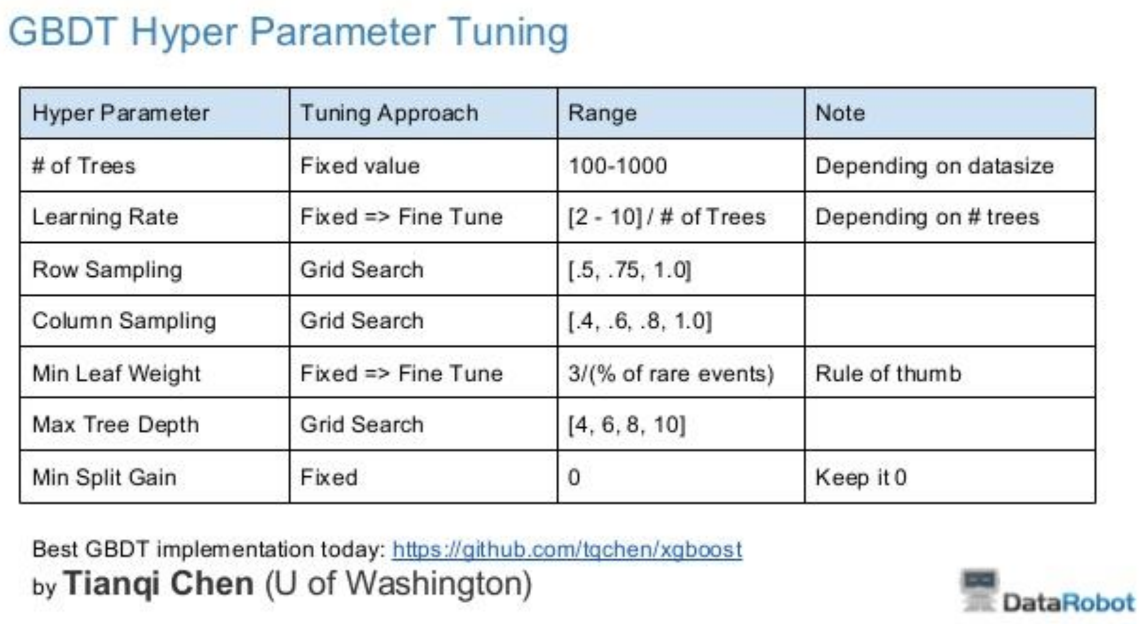
    c) create a model F2(x) = F1(x) + h1(x)

    ```
    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**

    ```
    o  eta = 0.3 or 0.1
    o  max_depth = 3
    o  n_estimators = 100
    o  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



GBDT Hyper Parameter Tuning

Hyper Parameter	Tuning Approach	Range	Note
# of Trees	Fixed value	100-1000	Depending on datasize
Learning Rate	Fixed => Fine Tune	[2 - 10] / # of Trees	Depending on # trees
Row Sampling	Grid Search	[.5, .75, 1.0]	
Column Sampling	Grid Search	[.4, .6, .8, 1.0]	
Min Leaf Weight	Fixed => Fine Tune	3/(% of rare events)	Rule of thumb
Max Tree Depth	Grid Search	[4, 6, 8, 10]	
Min Split Gain	Fixed	0	Keep it 0

Best GBDT implementation today: https://github.com/tqchen/xgboost
by **Tianqi Chen** (U of Washington)

- **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)
```

12. 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 $error_N > error_{N-1}$
    - 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

13. 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

14. 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')
```