H2Overview

Doug Raffle

Overview

- Distributed Computing Background
- Hadoop vs. Spark vs. H2O
- H2O Workflow
- Using H2O
  - H2O Prereqs
  - Initializing a Session & Loading Data
  - Modeling
  - Grid Searches
  - Working with Results

Distributed Computing Background

R is slow. The R-core team controls the basic functionality and structure of the language ("base R").

R-core tends to be very conservative about accepting new code. It can be frustrating to see R-core reject proposals that would improve performance. However, ***the overriding concern for R-core is not to make R fast, but to build a stable platform for data analysis and statistics*** *.* – Hadley Wickham (Advanced R)

How can R code be made faster without changing the language?

- Write critical pieces in C/C++/Java
- Use parallelization

What is Parallel Computing?

Parallel computing is the process of taking a problem or task and breaking it up into smaller pieces.

Each piece gets assigned to a **node** or **worker** which computes part of the solution. When all workers are finished, the parts are combined into an overall solution.

Distributed Computing Background

Parallelization in R can be done locally or on a remote cluster

Local Parallelization

- Packages: `parallel`, `foreach`, etc.
- Good for "embarassingly parallel problems"
  - Grid searching, cross validation, group/batch processing, etc
- Works natively on Linux (harder in Windows)
- Creates a new R process on separate processor cores
- Extremely memory intensive (data gets replicated for each core)
- Not good for iterative processes (IRLS, gradient descent, etc.)

When the problem is too big for local parallelization, R can connect to remote clusters (Hadoop, Spark, etc.).

Hadoop vs. Spark vs. H2O

A **Hadoop** cluster consists of four main pieces:

- Hadoop Common: libraries/utilities used in the system
- Hadoop Distributed File System (HDFS): file system which stores data in chunks across a cluster of computers
- YARN: scheduler/orchestrator for the cluster
- MapReduce: actual data processing framework

There is also a wider ecosystem of programs/utilites that run on top of the main framework (e.g., Hive, PIG)

Hadoop vs. Spark vs. H2O

The central component of **Hadoop** is MapReduce:

- Data is mapped into key-value pairs and processed in two stages
  - KV Pairs are processed individually & some computation is performed on them
  - After all data is initially processed, results are aggregated by key
- All steps are written to disk throughout processing, adding robustness but slowing IO times
- Well-suited for single-pass, "embarassingly parallel" problems
- For data science, MapReduce is generally best suited for data processing, not model fitting

Hadoop vs. Spark vs. H2O

Spark was an answer to the limitations of Hadoop

- Can sit on top of Hadoop, Using HDFS and YARN to manage data
- Spark has "Distributed Data Frame" archetype. The whole data set is viewed as a single entity, and distributed nature is invisible to the user
- Data is loaded in-memory on compute nodes, up to a 100x speed increase over Hadoop
- Ideal for iterative, multi-pass problems
- Focus was placed on providing a tool for data science, with implementations of common models and solvers (IRLS, Distributed Gradient Descent) written for massively parallel implementations
- Is integrated with R through `SparkR` and `SparklyR`
- Additional features:
- APIs in Java, Scala, Python, R, & Spark SQL
- Has interactive mode for data exploration

Hadoop vs. Spark vs. H2O

H2O is an app that sits on top of Hadoop and/or Spark and has an advanced modeling toolkit

- Data processing & variable transformation (feature engineering)
- Aggregation & summary statistics (exploratory data analysis)
- Statistical models & machine learning algorithms
- Includes capability to create stratified samples for k-Fold Cross Validation within model specification
- Can perform optimized grid and random hyperparameter searches for model tuning
- Full suite of built-in performance metrics & validation tools
- **Sparkling Water** is a Spark/H2O pipeline which integrates the tools for more flexible data processing in Spark and model fitting in H2O

H2O Workflow

A standard H2O workflow can be orchestrated (almost) entirely within R or Python

- Initialize H2O
- Import data from HDFS/local files
- Process/transform data with H2O and/or SparklyR/SparkR/Sparkling Water
- Grid search for model parameters
- Fit full model using cross validation or testing/training set
- Validate model, grab performance metrics
- Save/export model for scoring

Initializing a Local H2O Session

H2O sessions are initialized with the `h2o.init()` function.

Note that, by default, H2O will use all available cores.

It is extremely important to set the following parameters when starting H2O on ACE01 or ACE02

```
h2o.init(
    "localhost",
    nthreads = 20,                  # Number of cores to use.  Sharing is caring.
    min_mem_size = "1G",            # Set min mem used for safety.
    max_mem_size = "10G",           # Max mem size. sharing is still caring.
                                    #      H2O will generally be smart about only
                                    #      using what it needs.
    enable_assertions = FALSE,      # Sometimes Java is weird, this helps.
    port = 11111                    # Port to use
)
```

Initializing H2O on ACE is appropriate when you have small- to medium-sized data sets.

H2O on Hadoop

When the size of your data or the search space for your models too large to comfortable run H2O on ACE, it can be run on the Hadoop Cluster.

First, check the cluster's current workload using the scheduler: http://tdhfd6n3.thehartford.com:8088/cluster/scheduler (http://tdhfd6n3.thehartford.com:8088/cluster/scheduler)

By default H2O sessions are spun up in the `analytics.h2o` queue.

After ensuring that there is capacity on the cluster for your job, running H2O requires:

- Creating a Kerberos ticket
- Initializing a session
- Uploading/loading your data
- Executing your workflow
- Shutting down your session

H2O Flow

H2O includes a UI for interacting with your session. While it's possible to orchestrate your workflow entirely from Flow, it can also be used to track the status of jobs started in R (or Python).

For local sessions

ace02:portnumber (ace02:portnumber)

For Hadoop Sessions

The URL will be included towards the middle of the output messages:

```
H2O cluster (7 nodes) is up
Open H2O Flow in your web browser: http://10.218.2.22:54325
Disowning cluster and exiting.
```

Importing Data

Data in H2O is stored in a "data cloud" in the cluster's memory, and any data set is referenced by a `hex` key in H2O.

Creating H2O objects ties the R objects to these `hex` keys, but be aware names in Flow/behind the scenes may not match the names of the R objects.

Data can be uploaded directly from the R workspace:

```
library(data.table)
data.url <- "https://raw.githubusercontent.com/h2oai/app-consumer-loan/master/data/loan.csv"
loan.dt <- fread(data.url, stringsAsFactors = TRUE)
loan.hex <- as.h2o(loan.dt)
```

Note that importing large objects to/from the R workspace can be time consuming.

Also note that H2O converts R data types to H2O data types – this is generally OK, but you may need to manually coerce data ahead of time for unsupported types.

Importing Data from HDFS

Alternatively, you can directly import CSVs and other text-based data files from HDFS. If your data is not already in HDFS, you can use helper functions from the `ace` package to upload your data.

```
fwrite(loan.dt, "loan.csv")
hdfs.put("loan.csv", path = "hdfs://TDHFD6/user/dr88850e/loan.csv")
```

The ability to load data directly from HIVE is still in development, but you can see some posts on the ACE forum for details.

Once data is in HDFS, it can be loaded with `h2o.importFile`.

```
loan.hex <- h2o.importFile(path = "hdfs://TDHFD6/user/dr88850/loan.csv")
```

Importing from HDFS will be faster for large data sets you use multiple times.

Available Models/Algorithms

Supervised	Unsupervised
Cox Proportional Hazards (CoxPH)	Aggregator
Deep Learning (Neural Networks)	Generalized Low Rank Models (GLRM)
Distributed Random Forest (DRF)	Isolation Forest
Generalized Linear Model (GLM)	K-Means Clustering
Gradient Boosting Machine (GBM)	Principal Component Analysis (PCA)
Naive Bayes Classifier	
Stacked Ensembles	
XGBoost	

H2O has great documentation, including brief theoretical overviews of the algorithms, at the docs page http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science.html (http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science.html)

Specifying a Model

One of the advantages of H2O is that all models are specified with the same structure:

```
model.obj <- h2o.algorithmName(
    model_id = "",
    training_frame = tframe,
    validation_frame = vframe, ## if train/test/validate
    x = c("Predictor", "Column", "Names"),
    y = "Response_Column_Name",
    nfolds = nfolds, ## if cross-validating
    algo_specific_paramters
)
```

H2O will automatically generate a random `model_id`, but you will want to specify it yourself if you need to download your model later.

GLMs

GLMs in H2O are regularized by default using the Elastic Net penalty, to disable it set $\lambda = 0$.

Note that H2O will also standardize your columns by default and does not compute p-values because it regularizes by default.

```
x <- c(
    "term", "int_rate", "emp_length", "home_ownership", "annual_inc", "purpose",
    "addr_state", "dti", "delinq_2yrs", "revol_util", "total_acc",
    "longest_credit_length", "verification_status", "log_loan_amnt"
)
y <- "bad_loan"
loan.glm <- h2o.glm(
    model_id = "loan_glm",
    training_frame = train.hex,
    validation_frame = validate.hex,
    y = y,
    x = x,
    lambda = 0,
    family = "binomial",
    standardize = FALSE,
    compute_p_values = TRUE,
    remove_collinear_columns = TRUE
)
```

GLMs

Performance metrics and model summaries can seen using accessor functions

```
loan.glm
h2o.performance(loan.glm)
h2o.confusionMatrix(loan.glm)
h2o.coef(loan.glm)
```

More complex analyses of fit require scoring your test set manually and pulling the predictions to R

Regularized GLMs

H2O's GLM function has built-in support for Elastic Net regularization with automated λ searching. By default, the Elastic net is fit with $\alpha = 0.5$.

```
loan.enet <- h2o.glm(
    model_id = "loan_glm",
    training_frame = train.hex,
    validation_frame = validate.hex,
    y = y,
    x = x,
    family = "binomial",
    lambda_search = TRUE
)
```

Line searching is necessary for tuning α.

To obtain the full regularization path (similar to R's `glmnet`), use:

```
reg.path <- as.list(h2o.getGLMFullRegularizationPath(loan.enet))
```

GBMs

GBMs in H2O can be fit with `h2o.gbm` (or `h2o.xgboost`) with the following distributions:

- Bernoulli
- Quasibinomial
- Multinomial
- Gaussian
- Poisson
- Gamma
- Tweedie
- Laplace
- Quantile
- Huber

One main advantage of `h2o.gbm` is the handling of categorical variables by binning/grouping instead of needing to one-hot encode them.

Random Forests

Distributed Random Forests can be fit with the `h2o.randomForest` function.

```
loan.untuned.rf <- h2o.randomForest(
    model_id = "loan_untuned_rf",
    x = x,
    y = y,
    training_frame = train.hex,
    validation_frame = validate.hex,
    ## hyper parameters
    sample_rate = 0.6,
    col_sample_rate_per_tree = 0.3,
    min_rows = 20,
    max_depth = 10,
    ## early stopping
    stopping_metric = "logloss",
    stopping_tolerance = 0.01,
    stopping_rounds = 3,
    score_tree_interval = 5
)
```

Grid Searching

Grid searching in H2O is done using the function `h2o.grid`, which offers a convenient and consistent interface for all algorithms.

```
grid.search <- h2o.grid(
    grid_id = "grid_id",
    algorithm = "algorithm_name",
    training_frame = tframe,
    validation_frame = vframe, ## if train/test/validate
    nfolds = nfolds,           ## if crossvalidating
    x = c("Predictor", "Column", "Names"),
    y = "Response_Column_Name",
    hyper_params = list(),
    search_criteria = list(strategy = "Cartesian")
    ## any parameters that apply to all models
)
```

GBM Grid Search

To start, we specify a parameter space to search:

```
gbm.hyper.list <- list(
    sample_rate = c(0.5, 0.8),
    col_sample_rate_per_tree = c(0.5, 0.8),
    max_depth = c(5, 10),
    learn_rate = c(0.1, 0.05)
)
```

The search space is deliberately kept small for this training, but all parameters can be tuned.

For large data sets it may be necessary to search in stages, starting from a very coarse grain and honing in on your best parameters.

GBM Grid Search

```
gbm.grid <- h2o.grid(
    grid_id = "gbm_grid",
    algorithm = "gbm",
    training_frame = train.hex,
    validation_frame = validate.hex,
    ## nfolds = 3,
    x = x,
    y = y,
    hyper_params = gbm.hyper.list,
    ## global model parameters
    learn_rate_annealing = 0.99,
    ## early stopping
    stopping_metric = "logloss",
    stopping_tolerance = 0.001,
    stopping_rounds = 3,
    score_tree_interval = 5
)
```

Note that in this case we are tuning with a validation frame instead of cross-validation for speed during the training.

Because our data set is relatively small, this may lead to unstable results. In practice for a dataset this size, k-fold cross-validation would be more appropriate.

Viewing Grid Results

The default print for grid search objects gives an overview of the grid details and best results. The actual model specifications can be extracted from using `h2o.getGrid`, sorted by an appropriate loss metric.

```
sorted.grid <- h2o.getGrid(
    "gbm_grid",
    sort_by = "logloss",
    decreasing = TRUE
)
```

The actual results table can be extracted from the grid object and pulled to R as a `data.frame`

```
gbm.grid.df <- as.data.frame(sorted.grid@summary_table)
head(gbm.grid.df)
```

```
##   col_sample_rate_per_tree learn_rate max_depth sample_rate         model_ids             logloss
## 1                      0.5       0.05         5         0.5  gbm_grid_model_2  0.4381096363865256
## 2                      0.5       0.05        10         0.5  gbm_grid_model_6 0.43726558218108563
## 3                      0.8        0.1        10         0.5  gbm_grid_model_5 0.43718636044787995
## 4                      0.5       0.05         5         0.8 gbm_grid_model_10   0.436999507873676
## 5                      0.5        0.1        10         0.5  gbm_grid_model_4  0.4365085498294973
## 6                      0.8        0.1        10         0.8 gbm_grid_model_13  0.4362512846652138
```

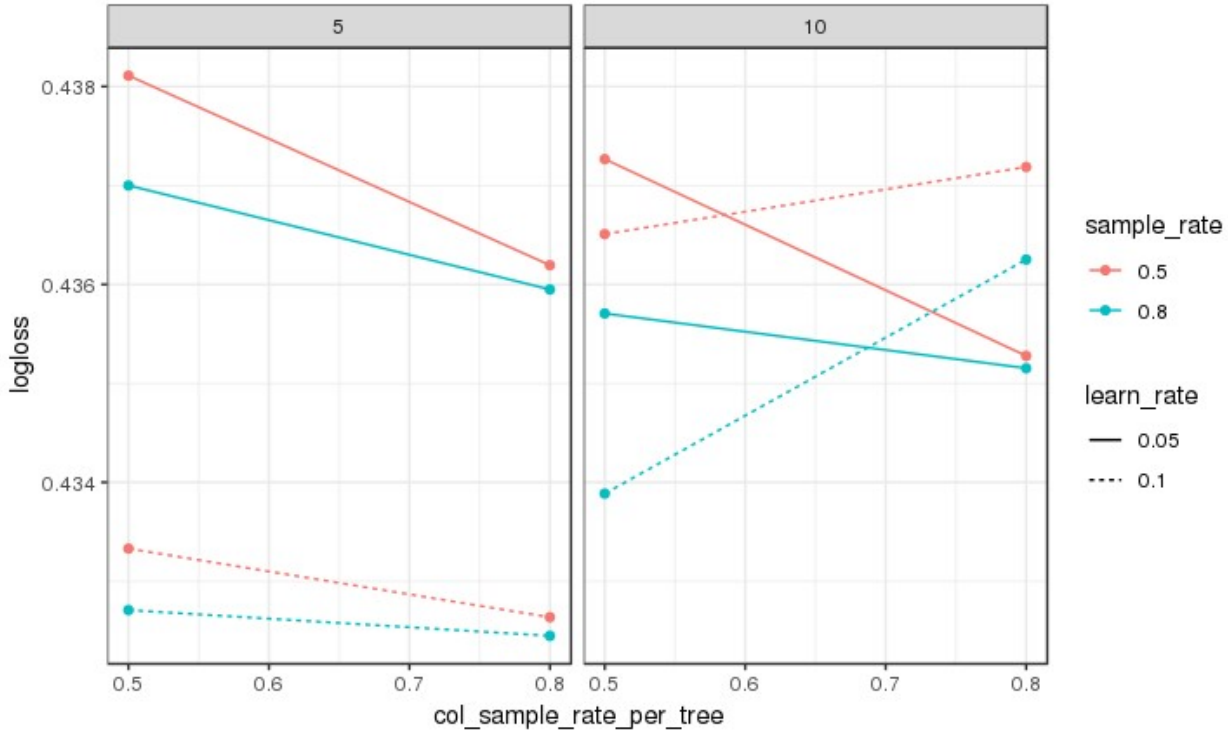
Plotting Grid Results

It can often be helpful to view the results of a grid search as a plot. Colors, linetypes, shapes, and faceting can be used to represent different parameters.

```
grid.plot <- gbm.grid.df %>%
    ggplot(aes(
        x = as.numeric(col_sample_rate_per_tree),
        y = as.numeric(logloss),
        color = sample_rate
    )) +
    geom_line(aes(linetype = learn_rate)) +
    geom_point() +
    facet_grid(~as.numeric(max_depth)) +
    theme_bw() +
    ylab("logloss") +
    xlab("col_sample_rate_per_tree")
```

Plotting Grid Results

```
grid.plot
```



Performance Metrics

Appropriate performance summaries for a model can be pulled using `h2o.performance()`.

```
h2o.performance(loan.enet, valid = TRUE)
```

The output is quite verbose, but more specific functions for specific performance metrics are also available (e.g., AUC, RMSE, R^2).

Note that all of these function have parameters `valid = T/F` for whether to use metrics evaluated on the validation frame if it exists, and `xval = T/F` for cross-validation metrics.

```
h2o.auc(loan.enet, valid = TRUE)
```

```
## [1] 0.7003024
```

Scoring Predictions

Prediction from H2O models are extracted with `h2o.predict`. Similar to its base R counterpart, the `newdata` argument allows you to score out-of-sample data.

```
h2o.predict(loan.enet, newdata = test.hex)
```

```
##   predict        p0         p1
## 1       1 0.8004469 0.19955312
## 2       0 0.9438414 0.05615864
## 3       0 0.8287273 0.17127270
## 4       0 0.9080053 0.09199468
## 5       0 0.9132598 0.08674022
## 6       0 0.9467658 0.05323422
##
## [32711 rows x 3 columns]
```

```
test.hex$enet.phat <- h2o.predict(loan.enet, newdata = test.hex)$p1
```

Extracting Predictions

Once predictions are made, you can use the `as.data.frame` function to load the data in your local R session and operate on them as you normally would.

```
test.df <- as.data.frame(test.hex)
```

For large datasets it may be best to extract just the predictions and a primary key, then join the predictions to your local data separately.

Alternatively, you can save your scored dataset directly to HDFS:

```
h2o.exportFile(test.hex, path = "hdfs://TDHFD6/user/dr88850e/loan_test_scored.csv")
```

Clean-Up

When finished, kill the H2O session.

```
h2o.shutdown(prompt = FALSE)
```

```
## [1] TRUE
```