CIS 8392
Topics in Big Data Analytics

#High Performance Machine Learning

Yu-Kai Lin

Diving into H2O

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

To use H2O with R, start H2O outside of R and connect to it, or launch H2O from R. However, if you launch H2O from R and close the R session, the H2O session closes as well.

Installing H2O for R

```
if(!"h2o" %in% rownames(installed.packages()) |
   packageVersion("h2o") != "3.32.1.3") {

  install.packages("remotes")
  remotes::install_version("h2o", "3.32.1.3", upgrade=F)
}

library(h2o)
library(tidyverse)
```

To start H2O in R on your local machine:

```
h2o.init(nthreads = -1) #-1 to use all cores
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         3 hours 19 minutes
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.32.1.3
##     H2O cluster version age:    5 months and 14 days !!!
##     H2O cluster name:           H2O_started_from_R_yklin_gux597
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   7.45 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3, TargetEncoder,
##     R Version:                  R version 4.1.0 (2021-05-18)
```

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:

```
# Don't run!
h2o.init(ip = "123.45.67.89", port = 54321,
         username = "admin", password = "someP@ssword")
```

You can perform other configuration for the H2O instance. Type `?h2o.init` to learn more about the parameters when initiating an H2O instance.

Alternatively, you can also connect H2O to a Spark cluster:

- https://spark.rstudio.com/guides/h2o/
- https://h2o-release.s3.amazonaws.com/sparkling-water/rel-2.4/1/doc/rsparkling.html

Checking Cluster Status:

```
h2o.clusterInfo()
```

```
## R is connected to the H2O cluster:
##     H2O cluster uptime:         3 hours 19 minutes
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.32.1.3
##     H2O cluster version age:    5 months and 14 days !!!
##     H2O cluster name:           H2O_started_from_R_yklin_gux597
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   7.45 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3, TargetEncoder,
##     R Version:                  R version 4.1.0 (2021-05-18)
```

Read the processed data (assuming that you saved them to your working directory):

```
y_train_processed_tbl <- read_rds("data/loan/y_train_processed_tbl.rds")
x_train_processed_tbl <- read_rds("data/loan/x_train_processed_tbl.rds")
x_test_processed_tbl <- read_rds("data/loan/x_test_processed_tbl.rds")
```

We created these RDS files earlier today before the break. If you do not have them, you can download them (~260 MB in total) directly from links below:

- https://www.dropbox.com/s/k5p2t1yn57rquoz/x_train_processed_tbl.rds?dl=0
- https://www.dropbox.com/s/i646kslbr6emmtt/x_test_processed_tbl.rds?dl=0
- https://www.dropbox.com/s/j3xtlvz0rrh69bc/y_train_processed_tbl.rds?dl=0

Push data into H2O

We currently run H2O on our local machine. But it can also run on distributed nodes. To ensure that data and models can be used in a distributed environment, an H2O instance essentially creates a layer of virtual container to host data and models so that you don't need to worry about whether you are dealing with one computer or a cluster of Spark nodes.

```
# push data into h2o; NOTE: THIS MAY TAKE A FEW MINUTES!
data_h2o <- as.h2o(
  bind_cols(y_train_processed_tbl, x_train_processed_tbl),
  destination_frame= "train.hex"  #destination_frame is optional
)
new_data_h2o <- as.h2o(
  x_test_processed_tbl,
  destination_frame= "test.hex"  #destination_frame is optional
)

# what if you do not assign destination_frame
data_h2o_no_destination <- as.h2o(
  bind_cols(y_train_processed_tbl, x_train_processed_tbl)
)
```

You can also list and remove data from an H2O instance:

```
h2o.ls()
```

```
##                    key
## 1 data.frame_sid_a65e_3
## 2              test.hex
## 3             train.hex
```

```
h2o_keys = as.character(h2o.ls()$key)
h2o.rm(h2o_keys[str_detect(h2o_keys, "^data")])
h2o.ls()
```

```
##         key
## 1  test.hex
## 2 train.hex
```

Splitting the training data

Most ML models need to be tuned. A simple way to allow H2O to tune a model is to split **training** data into 3 subsets:

- one for training: to fit a model
- one for validation: to scoring the parameters
- one for testing: to find the performance

```
# Partition the data into training, validation and test sets
splits <- h2o.splitFrame(data = data_h2o,
                         ratios = c(0.7, 0.15), # 70/15/15 split
                         seed = 1234)

train_h2o <- splits[[1]] # from training data
valid_h2o <- splits[[2]] # from training data
test_h2o  <- splits[[3]] # from training data
```

- Q: Why don't we just use `x_test_processed_tbl` for testing?
- A: In `x_test_processed_tbl`, we do not have the TARGET column. Without that, we do not know the true answer and we cannot evaluate the performance of our model.

Modeling

Template to build a supervised ML model in H2O:

```
# do not run; pseudo code
m1 <- h2o.<ALGORITHM_NAME>(
  model_id = <A_UNIQUE_ID_IN_THE_H2O_CONTAINER>,

  x = <COLUMN_NAMES_FOR_PREDICTORS>,
  y = <COLUMN_NAME_FOR_OUTCOME>,

  training_frame = <THE_NAME_OF_TRAINING_DATA_SPLIT>,
  validation_frame = <THE_NAME_OF_VALIDATION_DATA_SPLIT>,

  <OTHER_ALGORITHM_SPECIFIC_PARAMETERS>,

  <OTHER_MODELING_SPECIFIC_PARAMETERS>
)
```

Deep learning

Deep learning is a very promising algorithm, and we will take a close look on deep learning next week. Right now, you can just consider it as a black box, focusing on how to use it.

```
y <- "TARGET" # column name for outcome
x <- setdiff(names(train_h2o), y) # column names for predictors

m1 <- h2o.deeplearning(
  model_id = "dl_model_first",

  x = x,
  y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o, ## validation dataset: used for scoring and
                                ## early stopping

  #activation="Rectifier",      ## default
  #hidden=c(200,200),           ## default: 2 hidden layers, 200 neurons each

  epochs = 1                    ## one pass over the training data
)
```

```
summary(m1)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: deeplearning
## Model Key:  dl_model_first
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli dis
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   333     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2   200 Rectifier  0.00 % 0.000000 0.000000  0.200198 0.393466 0.000000
## 3     3   200 Rectifier  0.00 % 0.000000 0.000000  0.021804 0.046993 0.000000
## 4     4     2   Softmax      NA 0.000000 0.000000  0.004772 0.004855 0.000000
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.005826   0.062313  0.445917 0.021327
## 3   -0.004740   0.070422  0.382601 0.423107
## 4   -0.009300   0.365442  0.000025 0.065130
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10026 samples **
##
## MSE:  0.07135004
## RMSE:  0.2671143
## LogLoss:  0.2619753
## Mean Per-Class Error:  0.3732423
## AUC:  0.7174102
```

Save an H2O model

If you start H2O in R, the H2O instance will disappear once you run `h2o.shutdown()` or when you close RStudio. If it takes you a long time to train an ML model, you might want to save the model so that you can reuse it later.

```
h2o.saveModel(object = m1,     # the model you want to save
              path = getwd(),  # the folder to save
              force = TRUE)    # whether to overwrite an existing file
model_filepath = str_c(getwd(), "/dl_model_first") #dl_model_first is model_id
m1 <- h2o.loadModel(model_filepath) # load a model from file
```

Config algorithm/modeling parameters

You can manually configure many algorithm/modeling parameters. See here for a list of parameters in `h2o.deeplearning`: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/deep-learning.html

```
m2 <- h2o.deeplearning(
  model_id = "dl_model_faster",

  x = x,
  y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o,

  hidden = c(32,32,32),                  ## small network, runs faster
  epochs = 1000000,                      ## hopefully converges earlier...

  score_validation_samples = 10000,      ## sample the validation dataset (faster)
  stopping_metric = "misclassification", ## could also be "MSE","logloss","r2"
  stopping_rounds = 2,                   ## for 2 consecutive scoring events
  stopping_tolerance = 0.01              ## stop if the improvement is less than 1%
)
```

```
summary(m2)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: deeplearning
## Model Key:  dl_model_faster
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli dis
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   333     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2    32 Rectifier  0.00 % 0.000000 0.000000  0.185478 0.386405 0.000000
## 3     3    32 Rectifier  0.00 % 0.000000 0.000000  0.002942 0.011932 0.000000
## 4     4    32 Rectifier  0.00 % 0.000000 0.000000  0.019870 0.081021 0.000000
## 5     5     2   Softmax      NA 0.000000 0.000000  0.002712 0.002728 0.000000
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.002194   0.076385  0.475433 0.050793
## 3   -0.018157   0.178958  0.946162 0.036034
## 4   -0.010495   0.180952  0.966461 0.050450
## 5    0.048695   0.976581  0.000466 0.025538
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10014 samples **
##
## MSE:  0.0738655
## RMSE:  0.2717821
## LogLoss:  0.2742218
```

More serious tuning

```
m3 <- h2o.deeplearning(
  model_id="dl_model_tuned",

  x = x,
  y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o,
  overwrite_with_best_model = F,    ## Return the final model after 10 epochs,
                                    ## even if not the best

  hidden = c(128,128,128),          ## more hidden layers -> more complex interactions
  epochs = 10,                      ## to keep it short enough

  score_validation_samples = 10000, ## downsample validation set for faster scoring
  score_duty_cycle = 0.025,         ## don't score more than 2.5% of the wall time

  adaptive_rate = F,                ## manually tuned learning rate
  rate = 0.01,
  rate_annealing = 2e-6,
  momentum_start = 0.2,             ## manually tuned momentum
  momentum_stable = 0.4,
  momentum_ramp = 1e7,
  l1 = 1e-5,                        ## add some L1/L2 regularization
  l2 = 1e-5,
  max_w2 = 10                       ## helps stability for Rectifier
)
```

```
summary(m3)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: deeplearning
## Model Key:  dl_model_tuned
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli dis
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   333     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2   128 Rectifier  0.00 % 0.000010 0.000010  0.001883 0.000000 0.243094
## 3     3   128 Rectifier  0.00 % 0.000010 0.000010  0.001883 0.000000 0.243094
## 4     4   128 Rectifier  0.00 % 0.000010 0.000010  0.001883 0.000000 0.243094
## 5     5     2   Softmax      NA 0.000010 0.000010  0.001883 0.000000 0.243094
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.003417   0.067049  0.421413 0.024896
## 3   -0.015076   0.072690  0.823584 0.032859
## 4   -0.022123   0.088362  0.924006 0.050105
## 5    0.014168   0.176364  0.019453 0.914846
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10060 samples **
##
## MSE:  0.06901361
## RMSE:  0.2627044
## LogLoss:  0.2461145
```

Hyper-parameter tuning w/ grid search

What if you want to try different values for a parameter?

```
hyper_params <- list(
  hidden = list( c(32,32,32), c(64,64) ),
  input_dropout_ratio = c(0, 0.05),
  rate = c(0.01, 0.02),
  rate_annealing = c(1e-8, 1e-7, 1e-6)
)
```

```
grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id="dl_grid",

  x = x,
  y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o,

  epochs = 10,
  stopping_metric = "misclassification",
  stopping_tolerance = 1e-2,        ## stop when misclassification does not
                                    ## improve by >=1% for 2 scoring events
  stopping_rounds = 2,
  score_validation_samples = 10000, ## downsample validation set for faster scoring
  score_duty_cycle = 0.025,         ## don't score more than 2.5% of the wall time
  adaptive_rate = F,                #manually tuned learning rate
  momentum_start = 0.5,             #manually tuned momentum
  momentum_stable = 0.9,
  momentum_ramp = 1e7,
  l1 = 1e-5,
  l2 = 1e-5,
  activation = c("Rectifier"),
  max_w2 = 10,                      #can help improve stability for Rectifier
  hyper_params = hyper_params
)
```

```
grid <- h2o.getGrid("dl_grid", sort_by="logloss", decreasing=FALSE)
dl_grid_summary_table <- grid@summary_table
dl_grid_summary_table
```

```
## Hyper-Parameter Search Summary: ordered by increasing logloss
##         hidden input_dropout_ratio rate rate_annealing        model_ids
## 1 [32, 32, 32]                0.05 0.01         1.0E-7 dl_grid_model_11
## 2 [32, 32, 32]                 0.0 0.01         1.0E-7  dl_grid_model_9
## 3     [64, 64]                 0.0 0.02         1.0E-8  dl_grid_model_6
## 4     [64, 64]                0.05 0.01         1.0E-6 dl_grid_model_20
## 5     [64, 64]                0.05 0.01         1.0E-8  dl_grid_model_4
##               logloss
## 1 0.24699585414428377
## 2   0.248182862018067
## 3 0.24865745027434474
## 4 0.24907703785686172
## 5  0.2507739953908935
##
## ---
##          hidden input_dropout_ratio rate rate_annealing        model_ids
## 19 [32, 32, 32]                0.05 0.02         1.0E-8  dl_grid_model_7
## 20 [32, 32, 32]                 0.0 0.02         1.0E-7 dl_grid_model_13
## 21     [64, 64]                0.05 0.02         1.0E-7 dl_grid_model_16
## 22     [64, 64]                0.05 0.02         1.0E-8  dl_grid_model_8
## 23 [32, 32, 32]                0.05 0.01         1.0E-6 dl_grid_model_19
## 24 [32, 32, 32]                 0.0 0.01         1.0E-8  dl_grid_model_1
##                logloss
## 19  0.2700563431070738
## 20 0.27016505506241933
## 21  0.2721386109382501
## 22  0.2727716331920314
## 23   0.274676041154203
```

What's that @ sign?

R has three object oriented (OO) systems: S3, S4 and Reference Classes.

Central to any object-oriented system are the concepts of class and method. A class defines a type of object, describing what properties it possesses, how it behaves, and how it relates to other types of objects. Every object must be an instance of some class. A method is a function associated with a particular type of object.

We typically deal with S3 objects, and we use `$` to access values/attributes in an S3 object. Recall our lists and dataframes.

Compared to S3, the S4 object system is much stricter, and much closer to other OO systems. To access attributes of an S4 object you use `@`, not `$`.

So, `grid` in the previous slide is an S4 object. To access the summary table in `grid`, we run `grid@summary_table`.

To find the best model in the grid:

```
dl_grid_best_model <- h2o.getModel(dl_grid_summary_table$model_ids[1])
summary(dl_grid_best_model)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: deeplearning
## Model Key:  dl_grid_model_11
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli distribution,
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   333     Input  5.00 %       NA       NA        NA       NA       NA
## 2     2    32 Rectifier  0.00 % 0.000010 0.000010  0.008210 0.000000 0.587232
## 3     3    32 Rectifier  0.00 % 0.000010 0.000010  0.008210 0.000000 0.587232
## 4     4    32 Rectifier  0.00 % 0.000010 0.000010  0.008210 0.000000 0.587232
## 5     5     2   Softmax      NA 0.000010 0.000010  0.008210 0.000000 0.587232
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.007336   0.114471  0.375350 0.050625
## 3   -0.066901   0.193682  0.623189 0.178568
## 4   -0.088325   0.185394  0.567111 0.150150
## 5    0.096548   0.420455  0.055042 0.875516
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10111 samples **
##
## MSE:  0.07013385
## RMSE:  0.264828
## LogLoss:  0.2530402
```

To find the parameters used in the best model:

```
dl_grid_best_model_params <- dl_grid_best_model@allparameters
dl_grid_best_model_params # too long to show on one slide
```

```
## $model_id
## [1] "dl_grid_model_11"
##
## $nfolds
## [1] 0
##
## $keep_cross_validation_models
## [1] TRUE
##
## $keep_cross_validation_predictions
## [1] FALSE
##
## $keep_cross_validation_fold_assignment
## [1] FALSE
##
## $ignore_const_cols
## [1] TRUE
##
## $score_each_iteration
## [1] FALSE
##
## $balance_classes
```

Random Hyper-Parameter Search

We see the benefits of hyper-parameter search. But what if you have many parameter combinations that you want to try? How many combinations did we have in the previous `hyper_params`?

```
hyper_params <- list(
  hidden = list( c(32,32,32), c(64,64) ),
  input_dropout_ratio = c(0, 0.05),
  rate = c(0.01, 0.02),
  rate_annealing = c(1e-8, 1e-7, 1e-6)
)
```

We essentially construct a grid to store and try each combination define in the `hyper_params` list.

Often, hyper-parameter search for more than 4 parameters can be done more efficiently with **random parameter search** than with **grid search**.

Basically, chances are good to find one of many good models in less time than performing an exhaustive grid search.

```
hyper_params2 <- list(
  activation = c("Rectifier", "Tanh", "Maxout", "RectifierWithDropout",
                 "TanhWithDropout", "MaxoutWithDropout"),
  hidden = list( c(20,20), c(50,50), c(30,30,30), c(25,25,25,25)),
  input_dropout_ratio = c(0, 0.05),
  l1 = seq(from=0, to=1e-4, by=1e-6),
  l2 = seq(from=0, to=1e-4, by=1e-6)
)
```

Can you see how many possible combinations are there?

```
hyper_params2 # too long to show its entirety
```

```
## $activation
## [1] "Rectifier"             "Tanh"                  "Maxout"
## [4] "RectifierWithDropout" "TanhWithDropout"      "MaxoutWithDropout"
##
## $hidden
## $hidden[[1]]
## [1] 20 20
##
## $hidden[[2]]
## [1] 50 50
##
## $hidden[[3]]
## [1] 30 30 30
##
## $hidden[[4]]
## [1] 25 25 25 25
##
##
## $input_dropout_ratio
## [1] 0.00 0.05
##
## $l1
##   [1] 0.0e+00 1.0e-06 2.0e-06 3.0e-06 4.0e-06 5.0e-06 6.0e-06 7.0e-06 8.0e-06
##  [10] 9.0e-06 1.0e-05 1.1e-05 1.2e-05 1.3e-05 1.4e-05 1.5e-05 1.6e-05 1.7e-05
##  [19] 1.8e-05 1.9e-05 2.0e-05 2.1e-05 2.2e-05 2.3e-05 2.4e-05 2.5e-05 2.6e-05
##  [28] 2.7e-05 2.8e-05 2.9e-05 3.0e-05 3.1e-05 3.2e-05 3.3e-05 3.4e-05 3.5e-05
##  [37] 3.6e-05 3.7e-05 3.8e-05 3.9e-05 4.0e-05 4.1e-05 4.2e-05 4.3e-05 4.4e-05
##  [46] 4.5e-05 4.6e-05 4.7e-05 4.8e-05 4.9e-05 5.0e-05 5.1e-05 5.2e-05 5.3e-05
##  [55] 5.4e-05 5.5e-05 5.6e-05 5.7e-05 5.8e-05 5.9e-05 6.0e-05 6.1e-05 6.2e-05
##  [64] 6.3e-05 6.4e-05 6.5e-05 6.6e-05 6.7e-05 6.8e-05 6.9e-05 7.0e-05 7.1e-05
##  [73] 7.2e-05 7.3e-05 7.4e-05 7.5e-05 7.6e-05 7.7e-05 7.8e-05 7.9e-05 8.0e-05
```

```
length(unique(hyper_params2$activation)) *
  length(unique(hyper_params2$hidden)) *
  length(unique(hyper_params2$input_dropout_ratio)) *
  length(unique(hyper_params2$l1)) *
  length(unique(hyper_params2$l2))
```

```
## [1] 489648
```

Suppose each combination takes 60 seconds to train, how long will it take to finish them all?

```
lubridate::duration(60) * 489648
```

```
## [1] "29378880s (~48.58 weeks)"
```

So we don't want to try all combinations. Instead, we should *randomly* search these combinations and define when to stop searching.

```
# there could be multiple stopping criteria
# so we use a list to put all of them together
search_criteria = list(
  strategy = "RandomDiscrete",
  seed=1234567,

  stopping_metric = "auto", # logloss for classification
                            # deviance for regression
  stopping_rounds=5,        # stop when the last 5 models
  stopping_tolerance=0.01,  # improve less than 1%

  max_runtime_secs = 360,   # stop when the search took more than 360 seconds
  max_models = 100          # stop when the search tried over 100 models
)
```

```
grid2 <- h2o.grid(
  algorithm = "deeplearning",
  grid_id = "dl_grid_random",

  x = x,
  y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o,

  epochs = 1,
  stopping_metric = "logloss",
  stopping_tolerance = 0.01,        #stop when logloss improvement <1%
  stopping_rounds = 2,              #for 2 scoring events
  score_validation_samples = 10000,
  score_duty_cycle = 0.025,
  max_w2 = 10,                      #can help improve stability for Rectifier
  hyper_params = hyper_params2,
  search_criteria = search_criteria
)
```

```
ordered_grid2 <- h2o.getGrid("dl_grid_random",sort_by="logloss",decreasing=F)
dl_grid_random_summary_table <- ordered_grid2@summary_table
dl_grid_random_summary_table
```

```
## Hyper-Parameter Search Summary: ordered by increasing logloss
##             activation           hidden input_dropout_ratio     l1     l2
## 1 RectifierWithDropout     [30, 30, 30]                0.05 9.9E-5 9.9E-5
## 2            Rectifier         [50, 50]                 0.0 5.6E-5 5.0E-6
## 3                 Tanh     [30, 30, 30]                0.05 5.6E-5 7.4E-5
## 4      TanhWithDropout         [20, 20]                 0.0 3.2E-5 1.8E-5
## 5            Rectifier [25, 25, 25, 25]                 0.0 4.5E-5 3.6E-5
##                model_ids             logloss
## 1  dl_grid_random_model_6 0.25173783050545573
## 2 dl_grid_random_model_14  0.2518834669438815
## 3 dl_grid_random_model_10 0.25329866484261987
## 4 dl_grid_random_model_19  0.2586022208567894
## 5 dl_grid_random_model_16 0.25956554509619895
##
## ---
##           activation           hidden input_dropout_ratio     l1     l2
## 24            Maxout         [20, 20]                0.05 1.5E-5 9.0E-5
## 25   TanhWithDropout [25, 25, 25, 25]                 0.0 2.4E-5 4.9E-5
## 26   TanhWithDropout [25, 25, 25, 25]                 0.0 6.9E-5 1.6E-5
## 27         Rectifier [25, 25, 25, 25]                0.05 1.6E-5 8.3E-5
## 28 MaxoutWithDropout         [20, 20]                 0.0 1.2E-5 1.0E-4
## 29 MaxoutWithDropout         [50, 50]                 0.0 5.0E-6 8.5E-5
##                 model_ids             logloss
## 24 dl_grid_random_model_23  0.2749449290226682
## 25 dl_grid_random_model_18  0.2763380634719027
## 26  dl_grid_random_model_7  0.2771241112347529
## 27  dl_grid_random_model_9  0.2802006216251652
## 28 dl_grid_random_model_13  0.2806165819351336
```

To get the best model:

```
dl_grid_random_best_model <- h2o.getModel(dl_grid_random_summary_table$model_ids[1])
summary(dl_grid_random_best_model)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: deeplearning
## Model Key:  dl_grid_random_model_6
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli distribution,
##   layer units             type dropout       l1       l2 mean_rate rate_rms
## 1     1   333            Input  5.00 %       NA       NA        NA       NA
## 2     2    30 RectifierDropout 50.00 % 0.000099 0.000099  0.184219 0.381378
## 3     3    30 RectifierDropout 50.00 % 0.000099 0.000099  0.007448 0.005994
## 4     4    30 RectifierDropout 50.00 % 0.000099 0.000099  0.007135 0.007076
## 5     5     2          Softmax      NA 0.000099 0.000099  0.002009 0.000866
##   momentum mean_weight weight_rms mean_bias bias_rms
## 1       NA          NA         NA        NA       NA
## 2 0.000000   -0.004629   0.074622  0.408290 0.036740
## 3 0.000000   -0.014322   0.183705  0.800541 0.152622
## 4 0.000000   -0.032287   0.168356  0.674619 0.137307
## 5 0.000000   -0.063026   0.363297 -0.002349 0.689923
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 9995 samples **
##
## MSE:  0.06854883
## RMSE:  0.2618183
## LogLoss:  0.2541015
## Mean Per-Class Error:  0.3594403
```

To find the parameters used in the best model:

```
dl_grid_random_best_model_params <- dl_grid_random_best_model@allparameters
dl_grid_random_best_model_params # too long to show on one slide
```

```
## $model_id
## [1] "dl_grid_random_model_6"
##
## $nfolds
## [1] 0
##
## $keep_cross_validation_models
## [1] TRUE
##
## $keep_cross_validation_predictions
## [1] FALSE
##
## $keep_cross_validation_fold_assignment
## [1] FALSE
##
## $ignore_const_cols
## [1] TRUE
##
## $score_each_iteration
## [1] FALSE
##
## $balance_classes
```

Make prediction on unseen testing data

```
prediction_h2o_dl <- h2o.predict(dl_grid_random_best_model,
                                 newdata = new_data_h2o)


prediction_dl_tbl <- tibble(
  SK_ID_CURR = x_test_processed_tbl$SK_ID_CURR,
  TARGET = as.vector(prediction_h2o_dl$p1)
)
```

```
prediction_h2o_dl
```

```
##   predict        p0         p1
## 1       0 0.9424808 0.05751919
## 2       1 0.8152257 0.18477426
## 3       0 0.9660169 0.03398311
## 4       0 0.9743299 0.02567009
## 5       1 0.8797558 0.12024421
## 6       0 0.9688162 0.03118377
##
## [48744 rows x 3 columns]
```

```
prediction_dl_tbl
```

```
## # A tibble: 48,744 x 2
##    SK_ID_CURR TARGET
##         <dbl>  <dbl>
##  1     100001 0.0575
##  2     100005 0.185
##  3     100013 0.0340
##  4     100028 0.0257
##  5     100038 0.120
##  6     100042 0.0312
##  7     100057 0.0268
##  8     100065 0.122
##  9     100066 0.0296
```

Your turn

H2O has many other ML algorithms. **Gradient Boosting Machine (GBM)** is a popular choices in practice and frequently used by leading teams in Kaggle competitions.

GBM is a type of ensemble learning method and predict by combining the outputs from individual trees. GBM is similar to Random Forests in that both utilize trees to make predictions. However, it has been shown that GBM performs better than RF if parameters tuned carefully.

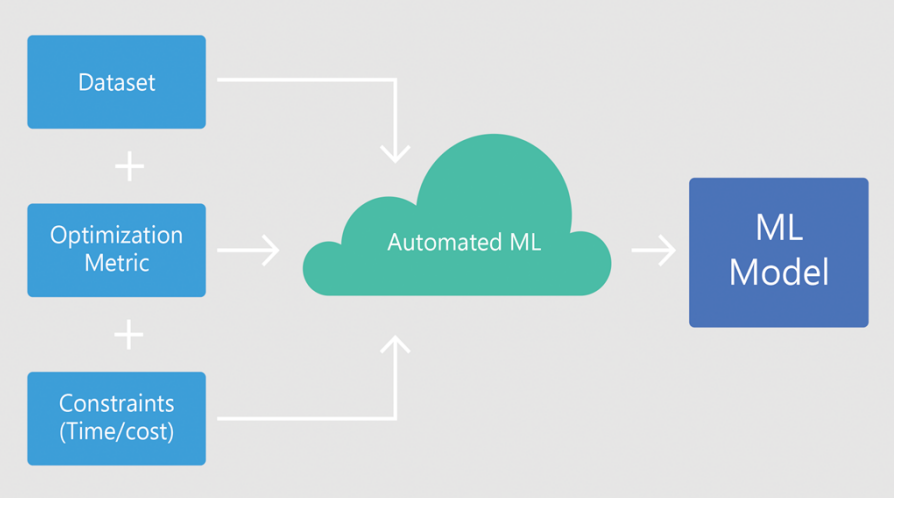
Try to apply the modeling strategies and procedure that you learned from the previous slides to train and tune GBM models. What is the best parameter? What is the best logloss?

- Here, you can find the H2O documentation for GBM: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html
- Here is an example on how to train a GBM model in H2O: https://github.com/h2oai/h2o-tutorials/blob/master/tutorials/gbm-randomforest/GBM_RandomForest_Example.R

AutoML

Automated machine learning (AutoML) is the process of automating the end-to-end process of applying machine learning to real-world problems.

AutoML enables developers with limited machine learning expertise to train high-quality models specific to their business needs in minutes.



Some interesting articles on AutoML:

- The Future Of Work Now: AutoML At 84.51° And Kroger
- AutoML 2.0: Is The Data Scientist Obsolete?
- The Risks of AutoML and How to Avoid Them

AutoML is hot

In view of the surging industrial needs in ML, many AutoML solutions have been proposed in recent years:

- Auto-sklearn
- Auto-Keras
- Google's AutoML
- Microsoft's AutoML
- Amazon's AutoML
- IBM's AutoML
- H2O's AutoML

H2O's AutoML

H2O's AutoML can be used for automating the machine learning workflow, which includes automatic training and tuning of many models within a user-specified time-limit.

In other words, instead of specifying what models and parameters you want, you specify how much time you have. H2O's AutoML will try different models and tune their parameters in the allocated time.

Obviously, if you do not allocate sufficient time, AutoML won't be able to reach the best model/parameters.

Run AutoML

```
automl_models_h2o <- h2o.automl(
  x = x,
  y = y,
  training_frame    = train_h2o,
  validation_frame  = valid_h2o,
  leaderboard_frame = test_h2o,
  max_runtime_secs  = 600  # suppose we only have 10 minutes,
                           # which is too short for most projects
)
```

```
##
## 16:31:59.911: User specified a validation frame with cross-validation still enabl
## 16:31:59.943: AutoML: XGBoost is not available; skipping it.
```

It is impossible to say how much time you would need to find a good model. It depends on many factors, including the complexity of the problem, the size of the data, the choice of algorithm, and the spec of the computer.

That being said, my experience is that I often need an hour to get a satisfactory model.

Inside AutoML

The current version of AutoML trains and cross-validates the following models:

- a default Random Forest,
- an Extremely-Randomized Forest,
- a random grid of Gradient Boosting Machines (GBMs),
- a random grid of Deep Neural Nets,
- a fixed grid of GLMs, and then
- two **Stacked Ensemble** models at the end
    - One ensemble contains all the models (optimized for model performance)
    - the second ensemble contains just the best performing model from each algorithm class/family (optimized for production use).

AutoML Leaderboard

```
automl_leaderboard <- automl_models_h2o@leaderboard
automl_leaderboard
```

```
##                                                  model_id       auc   logloss
## 1    StackedEnsemble_AllModels_AutoML_20211103_163159 0.7553451 0.2472808
## 2 StackedEnsemble_BestOfFamily_AutoML_20211103_163159 0.7541836 0.2476953
## 3          GBM_grid__1_AutoML_20211103_163159_model_2 0.7538553 0.2480176
## 4          GBM_grid__1_AutoML_20211103_163159_model_1 0.7528744 0.2483210
## 5          GBM_grid__1_AutoML_20211103_163159_model_3 0.7487122 0.2498449
## 6                        GBM_1_AutoML_20211103_163159 0.7425894 0.2527811
##       aucpr mean_per_class_error      rmse        mse
## 1 0.2329051            0.3352042 0.2610177 0.06813024
## 2 0.2298728            0.3455166 0.2612343 0.06824335
## 3 0.2299503            0.3498097 0.2612577 0.06825560
## 4 0.2285343            0.3457418 0.2614395 0.06835061
## 5 0.2203963            0.3511469 0.2622364 0.06876791
## 6 0.2163506            0.3458713 0.2629110 0.06912217
##
## [18 rows x 7 columns]
```

To get the best model:

```
automl_leader <- automl_models_h2o@leader
```

Look into its performance metrics:

```
performance_h2o <- h2o.performance(automl_leader, newdata = test_h2o)

performance_h2o %>%
  h2o.confusionMatrix()
```

```
## Confusion Matrix (vertical: actual; across: predicted)  for max f1 @ threshold = 
##            0    1    Error         Rate
## 0      36608 5779 0.136339  =5779/42387
## 1       1983 1730 0.534069   =1983/3713
## Totals 38591 7509 0.168373  =7762/46100
```

```
performance_h2o %>%
  h2o.auc()
```

```
## [1] 0.7553451
```

Make prediction on unseen testing data

```
prediction_h2o_automl <- h2o.predict(automl_leader,
                                     newdata = new_data_h2o)

prediction_automl_tbl <- tibble(
  SK_ID_CURR = x_test_processed_tbl$SK_ID_CURR,
  TARGET = as.vector(prediction_h2o_automl$p1)
)
prediction_automl_tbl
```

```
## # A tibble: 48,744 x 2
##    SK_ID_CURR TARGET
##         <dbl>  <dbl>
##  1     100001 0.0657
##  2     100005 0.134
##  3     100013 0.0371
##  4     100028 0.0329
##  5     100038 0.166
##  6     100042 0.0340
##  7     100057 0.0181
##  8     100065 0.0695
##  9     100066 0.0180
## 10     100067 0.0957
## # ... with 48,734 more rows
```

Finally...

Once we finish our h2o session, remember to shutdown the h2o cluster:

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