# CIS 8398
# Advanced AI Topics in Business

## #High Performance Machine Learning

**Yu-Kai Lin**

# Diving into H2O

H2O requires Java SE; if you do not already have Java installed, install the latest version from https://jdk.java.net/archive/ before installing H2O.

Detailed instructions to set up H2O on ...

- Windows: https://github.com/h2oai/h2o-3#43-setup-on-windows
- OS X: https://github.com/h2oai/h2o-3#44-setup-on-os-x

Installing H2O for R:

```
# this installation will take a while to finish
install.packages("h2o")

library(h2o)
library(tidyverse)
```

## To start H2O in R on your local machine:

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

```
##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##     C:\Users\YU-KAI~1\AppData\Local\Temp\RtmpGE9DoF\file6cf0418be38/h2o_yklin_sta
##     C:\Users\YU-KAI~1\AppData\Local\Temp\RtmpGE9DoF\file6cf051066389/h2o_yklin_st
##
##
## Starting H2O JVM and connecting:  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         1 seconds 96 milliseconds
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.42.0.2
##     H2O cluster version age:    2 months and 9 days
##     H2O cluster name:           H2O_started_from_R_yklin_osn291
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   15.94 GB
##     H2O cluster total cores:    20
##     H2O cluster allowed cores:  20
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
```

# Important note about memory:

- The h2o cluster would not work properly if it has insufficient memory.
- In my experience, we need at least 3 to 4 GB memory for the h2o cluster to run smoothly.
- Otherwise, you will likely see errors when you put data into the h2o cluster and when you train ML models.
- If you saw from `h2o.init` that your h2o cluster has less than 3 GB memory, please use the VM assigned to you to run h2o for this lecture.

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:         1 seconds 332 milliseconds
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.42.0.2
##     H2O cluster version age:    2 months and 9 days
##     H2O cluster name:           H2O_started_from_R_yklin_osn291
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   15.94 GB
##     H2O cluster total cores:    20
##     H2O cluster allowed cores:  20
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.3.1 (2023-06-16 ucrt)
```

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 are currently running 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 WHILE!
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_b6c1_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 and tune model parameters/weights
- one for validation: to score model parameters and assess whether the model was overfit; will not update model parameters/weights but could be used to adjust hyperparameters
- one for testing: to find the true predictive performance

```
# Partition the data into training, validation and test sets
splits <- h2o.splitFrame(data = data_h2o, seed = 1234,
                         ratios = c(0.7, 0.15)) # 70/15/15 split
train_h2o <- splits[[1]] # from training data
valid_h2o <- splits[[2]] # from training data
test_h2o  <- splits[[3]] # from training data
```

---

- YouTube: Train, Test, & Validation Sets explained
- Stackoverflow: What is the difference between model hyperparameters and model parameters?

# *Your turn*

Follow the previous slides to ...

- Install, load, and initialize h2o
- Push data into h2o
- Split the training data to three frames: `train_h2o`, `valid_h2o`, and `test_h2o`

# 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 at 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 distribution, CrossEntrop
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   260     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2   200 Rectifier  0.00 % 0.000000 0.000000  0.098265 0.269837 0.000000
## 3     3   200 Rectifier  0.00 % 0.000000 0.000000  0.022251 0.033869 0.000000
## 4     4     2   Softmax      NA 0.000000 0.000000  0.002844 0.002445 0.000000
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.000828   0.071994  0.378884 0.034416
## 3   -0.011573   0.072498  0.646392 0.256789
## 4   -0.011137   0.339686  0.000252 0.129822
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 9993 samples **
##
## MSE:  0.06964922
## RMSE:  0.2639114
## LogLoss:  0.269996
## Mean Per-Class Error:  0.3673017
## AUC:  0.7105323
## AUCPR:  0.1867587
## Gini:  0.4210647
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##           0    1    Error        Rate
## 0      8096 1106 0.120191  =1106/9202
## 1       486  305 0.614412    =486/791
## Totals 8582 1411 0.159312  =1592/9993
```

# 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
```

# *Your turn*

Follow the previous slides to ...

- create `m1`
- save `m1` to a local file on your hard drive
- load `m1` from the local 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 distribution, CrossEntrop
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   260     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2    32 Rectifier  0.00 % 0.000000 0.000000  0.092242 0.250755 0.000000
## 3     3    32 Rectifier  0.00 % 0.000000 0.000000  0.006811 0.007385 0.000000
## 4     4    32 Rectifier  0.00 % 0.000000 0.000000  0.008691 0.026243 0.000000
## 5     5     2   Softmax      NA 0.000000 0.000000  0.003058 0.002917 0.000000
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.014202   0.136138  0.362083 0.185740
## 3    0.022444   0.189169  1.037592 0.097450
## 4   -0.016264   0.199036  0.967808 0.028203
## 5   -0.099346   0.833129  0.002629 0.039463
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10000 samples **
##
## MSE:  0.06264087
## RMSE:  0.2502816
## LogLoss:  0.2381949
## Mean Per-Class Error:  0.3826524
## AUC:  0.6913137
## AUCPR:  0.1483374
## Gini:  0.3826274
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##           0    1    Error         Rate
## 0      8243 1060 0.113942   =1060/9303
```

# 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 distribution, CrossEntrop
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   260     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2   128 Rectifier  0.00 % 0.000010 0.000010  0.001853 0.000000 0.243960
## 3     3   128 Rectifier  0.00 % 0.000010 0.000010  0.001853 0.000000 0.243960
## 4     4   128 Rectifier  0.00 % 0.000010 0.000010  0.001853 0.000000 0.243960
## 5     5     2   Softmax      NA 0.000010 0.000010  0.001853 0.000000 0.243960
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.003321   0.070170  0.372466 0.023665
## 3   -0.015196   0.069509  0.832298 0.046251
## 4   -0.027047   0.091638  0.928582 0.050684
## 5   -0.001252   0.246862  0.003460 0.968291
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10001 samples **
##
## MSE:  0.06759048
## RMSE:  0.2599817
## LogLoss:  0.242505
## Mean Per-Class Error:  0.3679224
## AUC:  0.781004
## AUCPR:  0.280131
## Gini:  0.5620081
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##           0   1    Error         Rate
## 0      8572 609 0.066333    =609/9181
```

# *Your turn*

Follow the previous slides to create `m2` and `m3`

# 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     [64, 64]             0.00000 0.02000        0.00000  dl_grid_model_6
## 2     [64, 64]             0.05000 0.01000        0.00000  dl_grid_model_4
## 3 [32, 32, 32]             0.00000 0.02000        0.00000 dl_grid_model_13
## 4     [64, 64]             0.00000 0.01000        0.00000 dl_grid_model_10
## 5     [64, 64]             0.05000 0.01000        0.00000 dl_grid_model_12
##   logloss
## 1 0.24735
## 2 0.25094
## 3 0.25101
## 4 0.25296
## 5 0.25396
##
## ---
##          hidden input_dropout_ratio    rate rate_annealing        model_ids
## 19 [32, 32, 32]             0.05000 0.02000        0.00000  dl_grid_model_7
## 20     [64, 64]             0.00000 0.01000        0.00000 dl_grid_model_18
## 21 [32, 32, 32]             0.00000 0.01000        0.00000  dl_grid_model_1
## 22     [64, 64]             0.05000 0.01000        0.00000 dl_grid_model_20
## 23 [32, 32, 32]             0.00000 0.02000        0.00000  dl_grid_model_5
## 24 [32, 32, 32]             0.05000 0.02000        0.00000 dl_grid_model_15
##    logloss
## 19 0.26352
## 20 0.26358
## 21 0.26393
## 22 0.26583
## 23 0.26717
## 24 0.27737
```

# 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 data frames.

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_6
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli distribution, CrossEntro
##   layer units      type dropout       l1       l2 mean_rate rate_rms momentum
## 1     1   260     Input  0.00 %       NA       NA        NA       NA       NA
## 2     2    64 Rectifier  0.00 % 0.000010 0.000010  0.019608 0.000000 0.580024
## 3     3    64 Rectifier  0.00 % 0.000010 0.000010  0.019608 0.000000 0.580024
## 4     4     2   Softmax      NA 0.000010 0.000010  0.019608 0.000000 0.580024
##   mean_weight weight_rms mean_bias bias_rms
## 1          NA         NA        NA       NA
## 2   -0.031141   0.170362 -0.021337 0.129080
## 3   -0.121633   0.172044  0.141338 0.219300
## 4   -0.004173   0.393722  0.097226 0.673265
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10027 samples **
##
## MSE:  0.07065114
## RMSE:  0.2658028
## LogLoss:  0.2513873
## Mean Per-Class Error:  0.3245699
## AUC:  0.7737425
## AUCPR:  0.2242086
## Gini:  0.547485
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
```

## 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_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
## [1] FALSE
```

# 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 defined in the `hyper_params` list.

Often, hyper-parameter searches 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 there are?

```
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    MaxoutWithDropout     [50, 50]             0.00000 0.00009 0.00008
## 2    MaxoutWithDropout [30, 30, 30]             0.05000 0.00008 0.00009
## 3 RectifierWithDropout [30, 30, 30]             0.00000 0.00008 0.00010
## 4               Maxout     [50, 50]             0.05000 0.00009 0.00007
## 5 RectifierWithDropout     [20, 20]             0.05000 0.00000 0.00003
##                 model_ids logloss
## 1 dl_grid_random_model_24 0.25282
## 2 dl_grid_random_model_54 0.25356
## 3 dl_grid_random_model_60 0.25386
## 4 dl_grid_random_model_26 0.25473
## 5 dl_grid_random_model_55 0.25580
##
## ---
##           activation           hidden input_dropout_ratio      l1      l2
## 62            Maxout         [20, 20]             0.05000 0.00001 0.00000
## 63            Maxout [25, 25, 25, 25]             0.05000 0.00008 0.00001
## 64            Maxout     [30, 30, 30]             0.05000 0.00002 0.00007
## 65   TanhWithDropout [25, 25, 25, 25]             0.00000 0.00004 0.00009
## 66            Maxout         [20, 20]             0.05000 0.00001 0.00003
## 67 MaxoutWithDropout [25, 25, 25, 25]             0.00000 0.00006 0.00009
##                  model_ids logloss
## 62 dl_grid_random_model_59 0.27669
## 63 dl_grid_random_model_37 0.27807
## 64 dl_grid_random_model_14 0.27870
## 65  dl_grid_random_model_7 0.28028
## 66 dl_grid_random_model_35 0.28052
## 67 dl_grid_random_model_25 0.31217
```

## 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_24
## Status of Neuron Layers: predicting TARGET, 2-class classification, bernoulli distribution, CrossEntrop
##   layer units          type dropout       l1       l2 mean_rate rate_rms
## 1     1   260         Input  0.00 %       NA       NA        NA       NA
## 2     2    50 MaxoutDropout 50.00 % 0.000094 0.000077  0.134826 0.290061
## 3     3    50 MaxoutDropout 50.00 % 0.000094 0.000077  0.063054 0.115362
## 4     4     2       Softmax      NA 0.000094 0.000077  0.000473 0.000500
##   momentum mean_weight weight_rms mean_bias bias_rms
## 1       NA          NA         NA        NA       NA
## 2 0.000000    0.005328   0.100954  0.389213 0.353440
## 3 0.000000   -0.002785   0.097566  0.598326 0.236291
## 4 0.000000    0.036981   0.286167 -0.002230 0.481621
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 9994 samples **
##
## MSE:  0.06915422
## RMSE:  0.2629719
## LogLoss:  0.2546653
## Mean Per-Class Error:  0.3728313
## AUC:  0.723176
## AUCPR:  0.2121386
## Gini:  0.446352
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##          0    1    Error        Rate
```

## 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_24"
##
## $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
## [1] FALSE
```

# 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 0.9598650 0.04013503
## 2       1 0.7827352 0.21726476
## 3       0 0.9733020 0.02669799
## 4       0 0.9838361 0.01616395
## 5       1 0.8739061 0.12609387
## 6       0 0.9817398 0.01826025
##
## [48744 rows x 3 columns]
```

```
prediction_dl_tbl
```

```
## # A tibble: 48,744 × 2
##    SK_ID_CURR TARGET
##         <dbl>  <dbl>
##  1     100001 0.0401
##  2     100005 0.217
##  3     100013 0.0267
##  4     100028 0.0162
##  5     100038 0.126
##  6     100042 0.0183
##  7     100057 0.0319
##  8     100065 0.0776
##  9     100066 0.0263
```

# *Your turn*

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

GBM is a type of ensemble learning method and makes predictions 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 are tuned carefully.

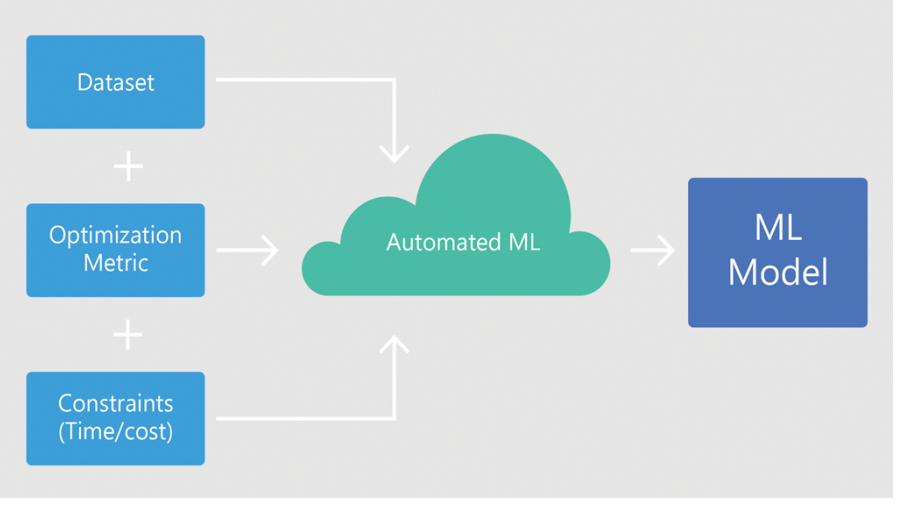
Try to apply the modeling strategies and procedure that you learned from the previous slides to train and tune GBM models. What is the best parameter? What is the best logloss?

- 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  = 300  # suppose we only have 5 minutes,
                           # which is too short for real-world projects
)
```

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 specs of the computer.

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

It is a good idea to first try a shorter run time, say 5 minutes, and then try a second time with a longer run time, say 1 hour.

# Explain H2O models

You can apply the XAI methods from our previous lecture to H2O models.

For demonstration purpose, let's fit a model using just 10 predictors.

We choose to use fewer predictors here because some XAI methods, such as Break Down plots, will become difficult to interpret with too many predictors.

```
x_train_processed_tbl = x_train_processed_tbl[, 1:10]
x_test_processed_tbl = x_test_processed_tbl[, 1:10]
# push data into h2o
data_h2o <- as.h2o(
  bind_cols(y_train_processed_tbl, x_train_processed_tbl),
  destination_frame= "train.hex"  #destination_frame is optional
)
# Partition the data into training, validation and test sets
splits <- h2o.splitFrame(
  data = data_h2o, ratios = c(0.7, 0.15), seed = 1234)

train_h2o <- splits[[1]]
valid_h2o <- splits[[2]]
test_h2o  <- splits[[3]]
```

```r
y <- "TARGET"
x <- setdiff(names(train_h2o), y)

m4 <- h2o.deeplearning(
  model_id = "dl_model_for_xai",

  x = x, y = y,

  training_frame = train_h2o,
  validation_frame = valid_h2o,

  epochs = 10,
  variable_importances = T
)
```

Now we can use `DALEX` to create an explainer for this H2O model:

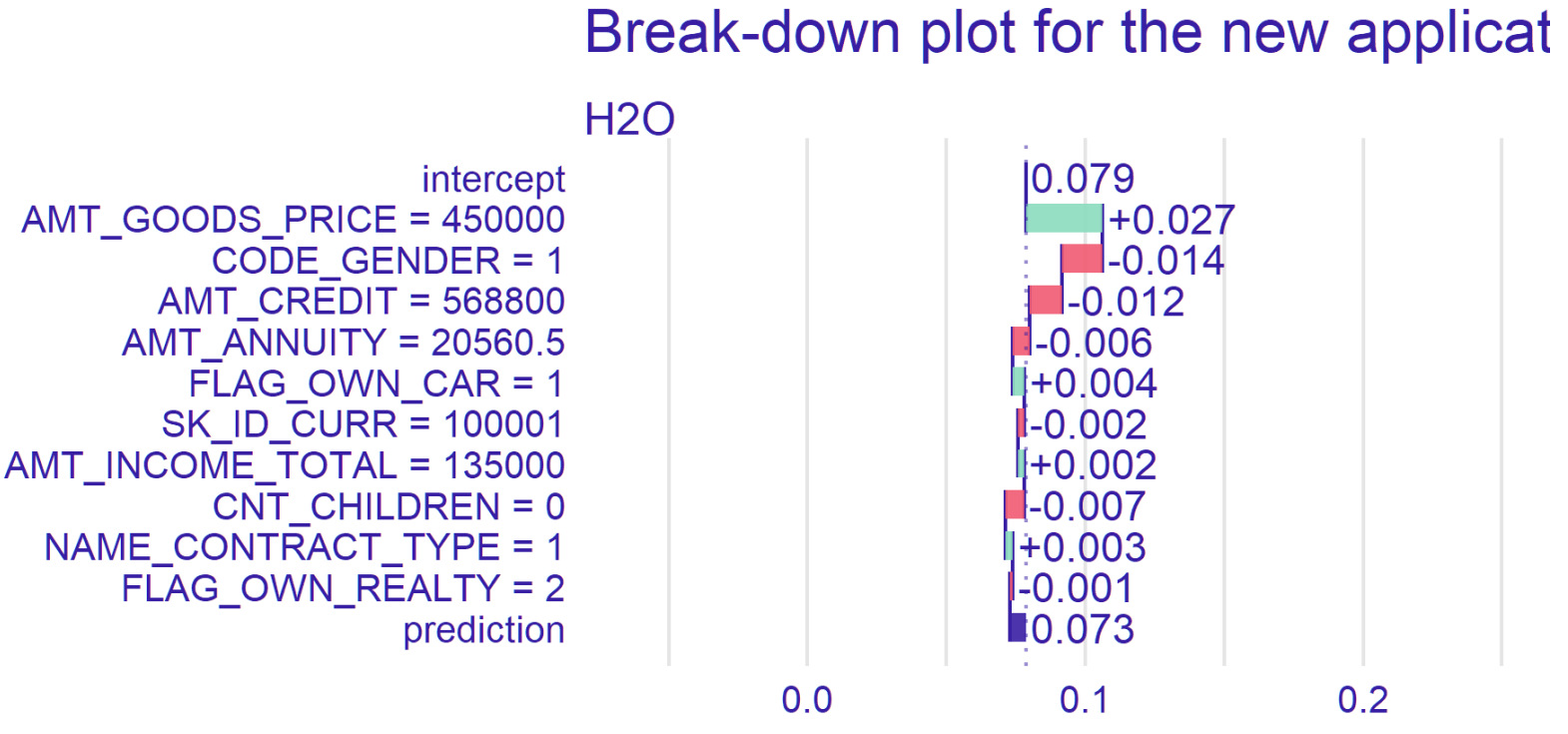
```
library(DALEXtra)

h2o_exp = explain_h2o(
  m4, data = x_train_processed_tbl,
  y = y_train_processed_tbl$TARGET == 1,
  label = "H2O", type = "classification")
```

Let's also consider a new application so that we can perform local/instance-level explanations:

```
library(DALEXtra)
new_application = x_test_processed_tbl[1, 1:10]
```

# Break-down Plot

```
h2o_exp_bd <- predict_parts(
  explainer = h2o_exp, new_observation = new_application,
  type = "break_down")

plot(h2o_exp_bd) + ggtitle("Break-down plot for the new application")
```



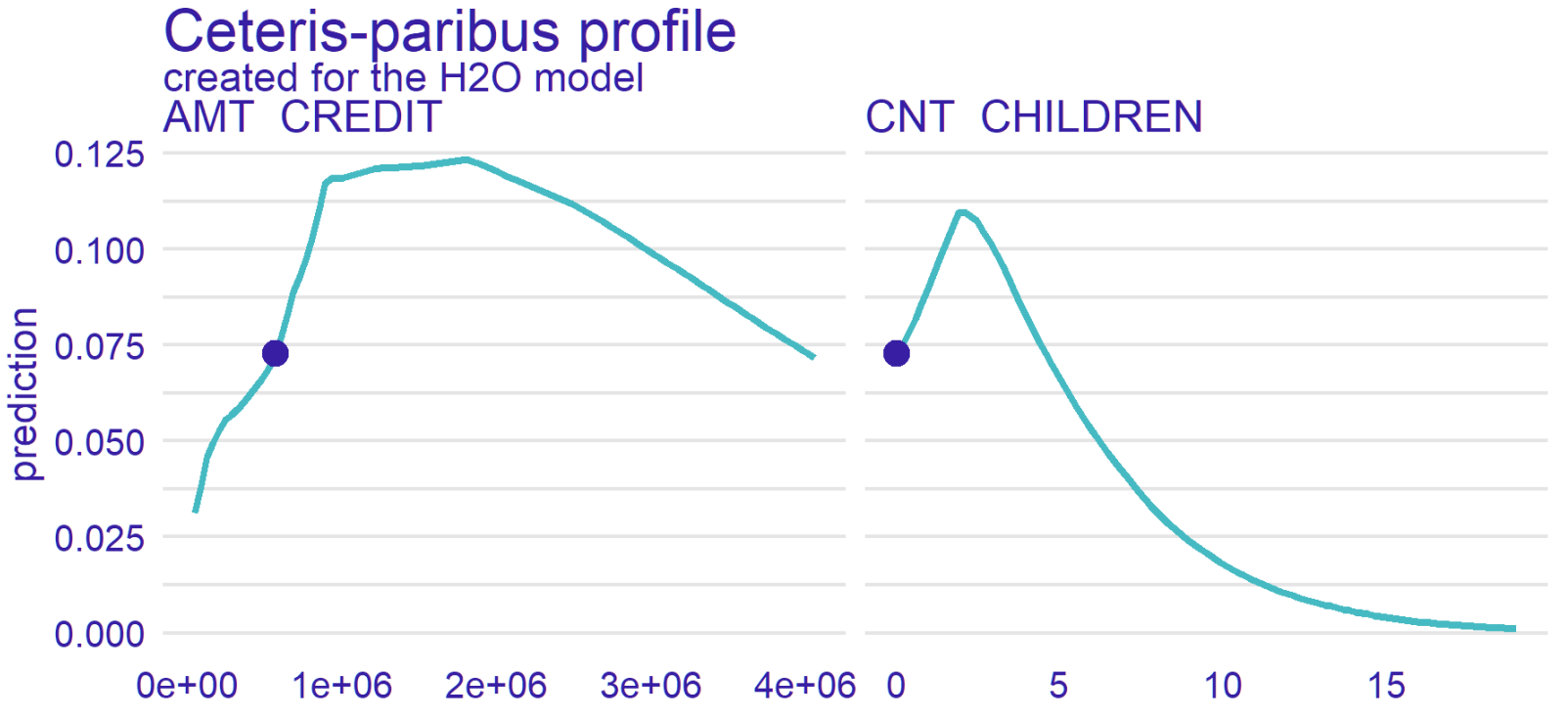
# SHAP

```
h2o_exp_shap <- predict_parts(
  explainer = h2o_exp, new_observation = new_application,
  type = "shap", B = 25)

h2o_exp_shap
```

```
##                                          min       q1  median          mean
## H2O: AMT_ANNUITY = 20560              -0.018 -0.01200 -0.0115 -0.0108461538
## H2O: AMT_CREDIT = 568800              -0.011 -0.00600  0.0030  0.0037307692
## H2O: AMT_GOODS_PRICE = 450000         -0.002  0.00550  0.0190  0.0158461538
## H2O: AMT_INCOME_TOTAL = 135000         0.001  0.00200  0.0030  0.0031538462
## H2O: CNT_CHILDREN = 0                 -0.007 -0.00575 -0.0040 -0.0045769231
## H2O: CODE_GENDER = F                  -0.021 -0.01900 -0.0160 -0.0164615385
## H2O: FLAG_OWN_CAR = N                  0.003  0.00500  0.0065  0.0059615385
## H2O: FLAG_OWN_REALTY = Y              -0.002 -0.00100 -0.0010 -0.0008846154
## H2O: NAME_CONTRACT_TYPE = Cash loans  0.000  0.00100  0.0020  0.0019615385
## H2O: SK_ID_CURR = 1e+05               -0.006 -0.00500 -0.0040 -0.0038076923
##                                            q3    max
## H2O: AMT_ANNUITY = 20560              -0.01000 -0.001
## H2O: AMT_CREDIT = 568800               0.01200  0.020
## H2O: AMT_GOODS_PRICE = 450000          0.02700  0.033
## H2O: AMT_INCOME_TOTAL = 135000         0.00300  0.007
## H2O: CNT_CHILDREN = 0                 -0.00300 -0.002
## H2O: CODE_GENDER = F                  -0.01400 -0.012
## H2O: FLAG_OWN_CAR = N                  0.00700  0.009
## H2O: FLAG_OWN_REALTY = Y              -0.00100  0.000
## H2O: NAME_CONTRACT_TYPE = Cash loans  0.00300  0.004
## H2O: SK_ID_CURR = 1e+05               -0.00325  0.000
```

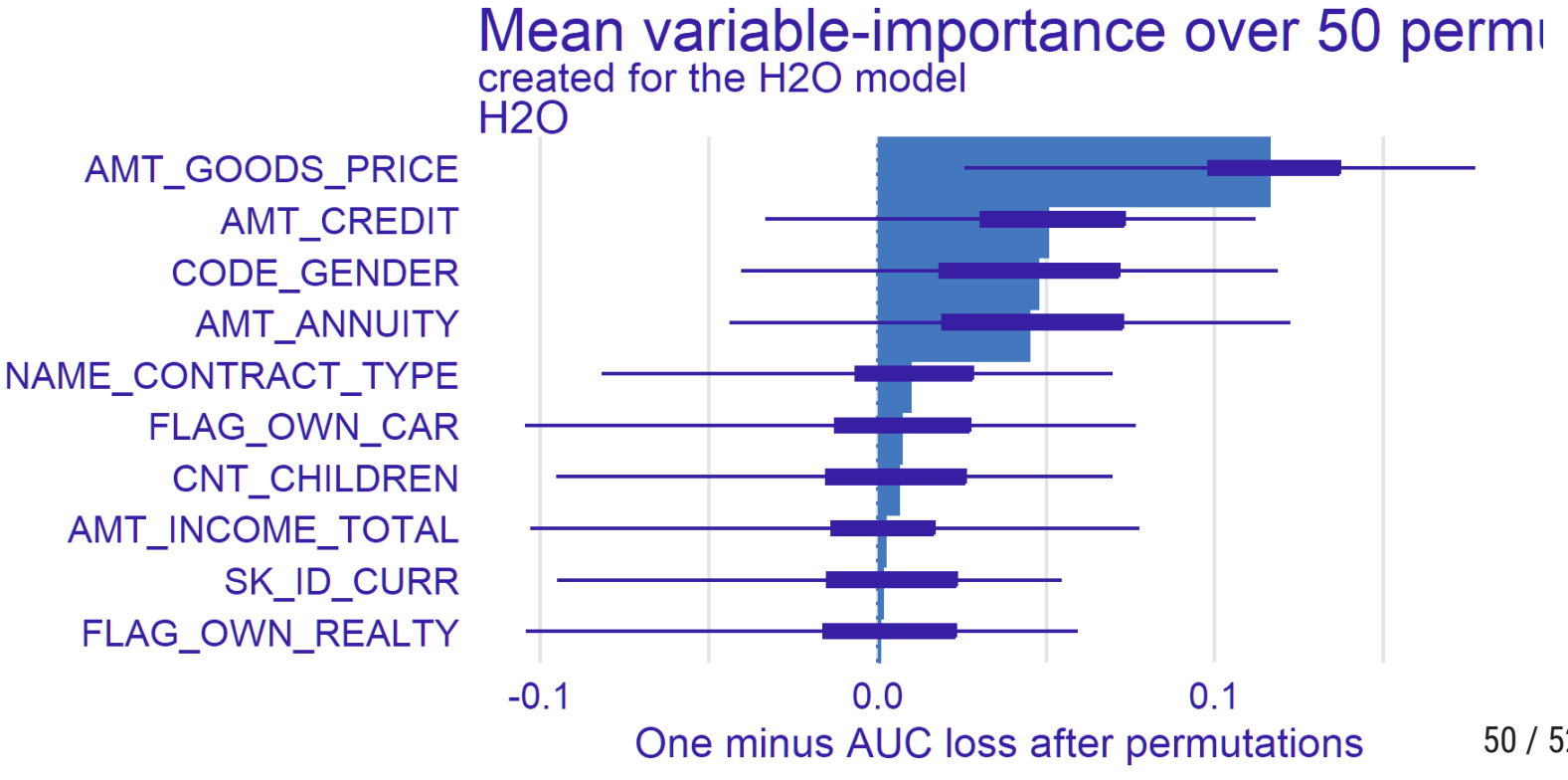
# Ceteris-paribus Profiles

```
h2o_exp_cp <- predict_profile(
  explainer = h2o_exp, new_observation = new_application)

plot(h2o_exp_cp, variables = c("AMT_CREDIT", "CNT_CHILDREN")) +
  ggtitle("Ceteris-paribus profile")
```



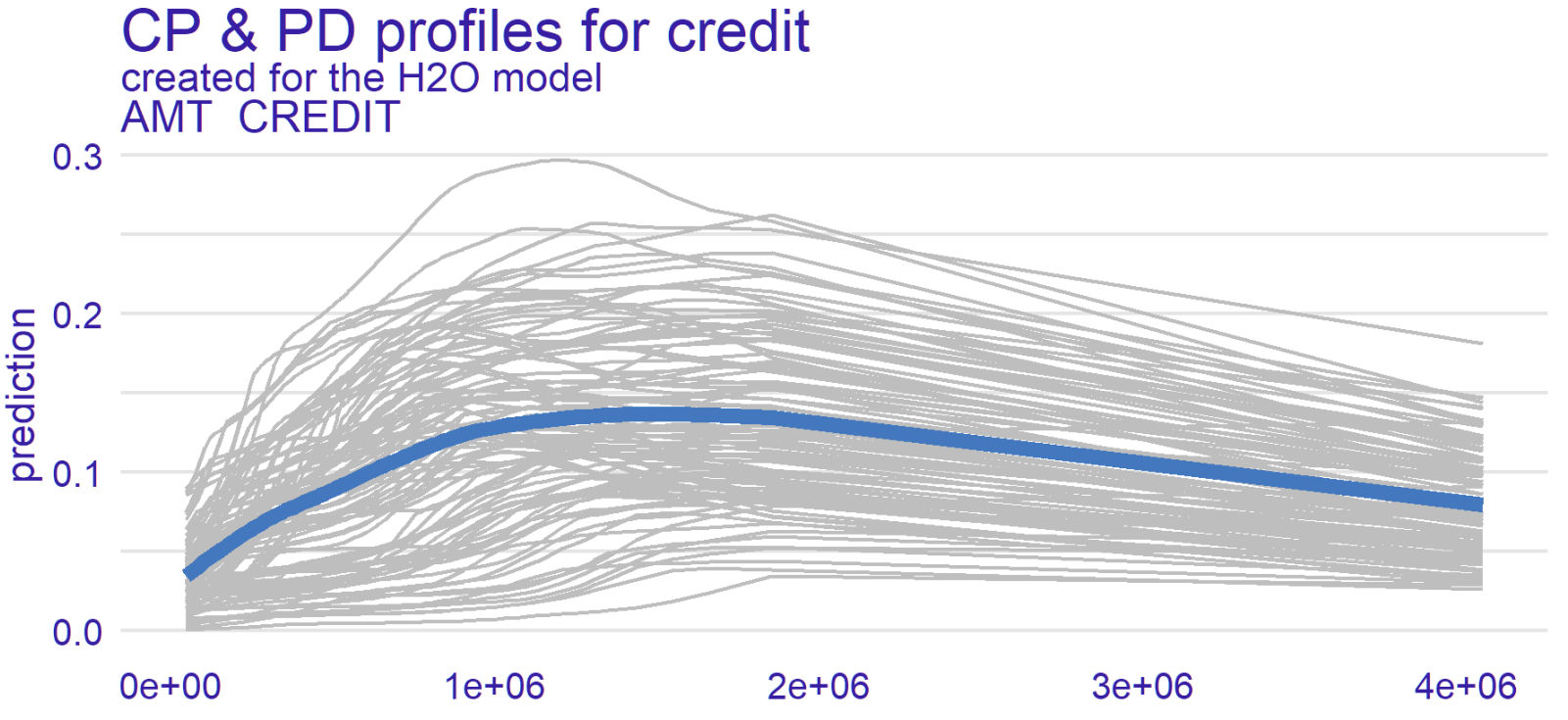
# Variable-importance Measures

```
h2o_exp_vip <- model_parts(
  explainer = h2o_exp,
  B = 50, type = "difference")

plot(h2o_exp_vip) +
  ggtitle("Mean variable-importance over 50 permutations")
```



# Partial-dependence Profiles

```
h2o_exp_pdp <- model_profile(
  explainer = h2o_exp, variables = "AMT_CREDIT")

plot(h2o_exp_pdp, geom="profiles") +
  ggtitle("CP & PD profiles for credit")
```



# Finally...

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

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

- After you shut down the h2o cluster, the data in the cluster will be gone.
- If you restart the cluster (i.e., `h2o.init()`), be sure to push data into h2o again!