DZone

Machine Learning With H2O — Hands-On Guide for Data Scientists

by Pavel Pscheidl · Jun. 27, 18 · AI Zone · Analysis

H_2O is the world's number one machine learning platform. It is an open-source software, and the H2O-3 GitHub repository is available for anyone to start hacking. This hands-on guide aims to explain the basic principles behind H_2O and get you as a data scientist started as quickly as possible in the most simple way. The rest is just machine learning.

After reading this guide, you'll be able to:

- understand which basic problems H_2O solves and why,
- play with H_2O — explore data and create and tune models,
- see beyond the horizon. Understand where H_2O can take you.

As a data scientist, you're most likely to use R and/or Python. H_2O integrates with both. Interestingly, H_2O makes it easy to seamlessly switch Python, R, and other data science tools while still working on the same project. This allows data scientists to interact more easily, as well as use the best tool for the job, but the possibilities do not stop there. H_2O also offers its own web-based interface named Flow. By means of Flow, data scientists are able to import, explore, and modify datasets, play with models, verify models performances, and much more. Flow is beautiful and a quick way to do machine learning. Flows can be saved and given to other data scientists, making cooperation easy.

H_2O respects habits of data scientists and does not get into their way. Using Python, data scientists are familiar with Pandas, Scikit, NumPy and others. H_2O's syntax is very similar to those. H_2O is able to work directly with Pandas' data structures, as well as being compatible with NumPy's arrays and primitive Python lists and collections. H_2O follows the same pattern with R, respecting the naming and syntax R developers are used to.

Getting Started

The preparations before take-off are short. H_2O is extremely easy to start with. All it takes is a common laptop to get started. Once H_2O is installed, it is very easy and convenient to import a dataset and create a model out of it. In the examples below, the famous Airlines Delay dataset is used. There is no need to download it, as H_2O is going to take care of downloading the dataset for you. The dataset is very intuitive to work with, however if you're unfamiliar with it or simply want to know more, visit Kaggle website for a brief description of each column.

Once fluent in R, working with H_2O in Python's environment is easy. And it always works the other way around, as the APIs are very similar, while respecting the target platforms.

Getting Started in R

Open up R CLI (by typing `R` in terminal on most systems) or start R-Studio. It takes just a few lines to install H_2O in R.

Before installing H_2O itself, H_2O requires two packages: `RCurl` and `jsonlite`. Install those by entering the following command into R console.

Installation

```
install.packages("RCurl","jsonlite")
```

After `RCurl` and `jsonlite` are installed, one last step is to install H_2O itself. Installation of latest stable release is done as demonstrated in the following snippet. During the installation, a one-time download of the H_2O backend containing all the algorithms and computing know-how will occur.

```
install.packages("h2o", type="source", repos=(c("http://h2o-release.s3.amazonaws.co
```

That's it. H_2O is now installed and ready to be used. As a first step, it is required to tell R to import the H_2O library with `library(h2o)` command. Once the library is imported, instruct H_2O to start itself by calling `h2o.init()`. Both commands are placed in the following code snippet for clarity.

```
library(h2o)
h2o.init()
```

The `h2o.init()` command is pretty smart and does a lot of things. First, an attempt is made to search for an existing H_2O instance being started already before starting a new one. When none is found automatically or specified manually with argument available, a new instance of H_2O is started. As this is a fresh installation and it is highly unlikely there is an instance of H_2O already running in your environment, a new instance is started right away. During startup, H_2O is going to print some useful information. The R version it is running on, H_2O's version, and how to connect to H_2O's Flow interface or where error logs reside, just to name a few. As usual, an example of H_2O's output during startup is to be found below this text in the snippet.

```
> h2o.init()

H2O is not running yet, starting it now...

Note:  In case of errors look at the following log files:
    /tmp/RtmpYs7uDC/h2o_pavel_started_from_r.out
    /tmp/RtmpYs7uDC/h2o_pavel_started_from_r.err

java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)

Starting H2O JVM and connecting: . Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         3 seconds 44 milliseconds
    H2O cluster timezone:       Europe/Prague
    H2O data parsing timezone:  UTC
```

```
  H2O data parsing timezone:  UTC
  H2O cluster version:        3.20.0.2
  H2O cluster version age:    8 days
  H2O cluster name:           H2O_started_from_R_pavel_yuw261
  H2O cluster total nodes:    1
  H2O cluster total memory:   5.21 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:         XGBoost, Algos, AutoML, Core V3, Core V4
  R Version:                  R version 3.4.4 (2018-03-15)
```

Data Import

Let's import a dataset and train a model on it very quickly!

```
airlinesTrainData <- h2o.importFile("https://s3.amazonaws.com/h2o-airlines-unpacke
```

H_2O will automatically download the dataset and parse it. It will also try to guess the datatype of each column automatically. H_2O does a great job at datatype recognition, however, each decision can be overridden manually by the user, if required. The imported dataset can also be given a name using `destination_frame` argument. For example, `h2o.importFile("https://s3.amazonaws.com/h2o-airlines-unpacked/allyears2k.csv", destination_frame='airlines_train')` imports the very same dataset with airplane delays that can be further addressed by name `airlines_train` , even from other interfaces like Python, another R console, Flow, Java, or direct API calls. If no name is provided, H_2O will generate an artificial name. Simply put, an imported dataset is called `Frame` in H_2O. List of frames can be shown by using the `h2o.ls()` function. An example output of calling `h2o.ls()` function can be found in the following code snippet. The very first record in the example is a named frame, and the second one is a frame name generated automatically by H_2O.

```
> h2o.ls()
                      key
1           airlines_train
2 allyears2k.hex_sid_ae6f_1
```

A preview of the data imported can be displayed with by typing the variable pointing to the `H2OFrame` , in this case `airlinesTrainData` .

```
> airlinesTrainData
  Year Month DayofMonth DayOfWeek DepTime CRSDepTime ArrTime CRSArrTime UniqueCarr
1 1987    10         14         3     741        730     912        849
2 1987    10         15         4     729        730     903        849
3 1987    10         17         6     741        730     918        849
4 1987    10         18         7     729        730     847
5 1987    10         19         1     749        730     922        849
6 1987    10         21         3     728        730     848        849
  Origin Dest Distance TaxiIn TaxiOut Cancelled CancellationCode Diverted CarrierDe
```

```
1    SAN  SFO      447     NaN      NaN          0                NA         0
2    SAN  SFO      447     NaN      NaN          0                NA         0
3    SAN  SFO      447     NaN      NaN          0                NA         0
4    SAN  SFO      447     NaN      NaN          0                NA         0
5    SAN  SFO      447     NaN      NaN          0                NA         0
6    SAN  SFO      447     NaN      NaN          0                NA         0
  IsDepDelayed
1          YES
2           NO
3          YES
4           NO
5          YES
6           NO
```

Model Training

On top of the data imported, a model can be built quickly. There are many algorithms available in H_2O. For the purpose of this tutorial, a widely known Gradient Boosting Machines method will be used. Let's train a model that is able to predict if the plane arrives late based on month, day of week, and distance the plane has to travel before reaching its destination. By invoking `h2o.gbm(...)` , H2O will run a gradient boosting algorithm on the data. There are many variables to play with that each and every data scientist can explore on his/her own. Overriding the default hyperparameters would only make this tutorial more complicated. H_2O only needs to know three things:

- predictor columns,
- response variable column,
- training frame — a dataset to train the model on.

Nothing more. It is even able to guess the distribution of the response variable, even though as stated before, everything can be overridden manually by the data scientist, if required. After a model is trained, basic information about the model can be shown just by typing the name of the variable pointing to the trained model, in this case `gbmModel` .

```
> gbmModel <- h2o.gbm(x=c("Month", "DayOfWeek", "Distance"), y="IsArrDelayed", tra
  |=================================================================================
> gbmModel
Model Details:
==============

H2OBinomialModel: gbm
Model ID:  GBM_model_R_1529834562303_470
Model Summary:
  number_of_trees number_of_internal_trees model_size_in_bytes min_depth max_depth
1              50                       50               20591         5         5


H2OBinomialMetrics: gbm
** Reported on training data. **
```

```
MSE:  0.2349663
RMSE:  0.4847332
LogLoss:  0.6609351
Mean Per-Class Error:  0.4892883
AUC:  0.6237765
Gini:  0.2475531

Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
        NO   YES    Error          Rate
NO     604 18933 0.969084  =18933/19537
YES    232 24209 0.009492    =232/24441
Totals 836 43142 0.435786  =19165/43978

Maximum Metrics: Maximum metrics at their respective thresholds
                        metric threshold    value idx
1                       max f1  0.431588 0.716423 368
2                       max f2  0.355217 0.862241 395
3                 max f0point5  0.513454 0.633681 278
4                 max accuracy  0.511993 0.594661 279
5                max precision  0.972534 1.000000   0
6                   max recall  0.347469 1.000000 397
7              max specificity  0.972534 1.000000   0
8             max absolute_mcc  0.605839 0.175948 151
9   max min_per_class_accuracy  0.539888 0.582464 234
10 max mean_per_class_accuracy  0.547177 0.584595 225

Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(
```

Overall, this model is not expected to perform very well, given the huge error rate to be observed in the confusion matrix. By playing with different GBM hyperparameters and including different predictors into the model, much better results can be achieved. As a data scientist, the task of making the model perform better is easy for you, and that's certain. In the `h2o.` package, there are many additional functions to work with the model and help a data scientist understand what happened during the training phase and much more. As an example, the `h2o.varimp` function shows importances of variables (relative, percentage) taken into account in the model. As you begin exploring H_2O, the reference guide will guide you through all the H_2O's functionality.

```
> h2o.varimp(gbmModel)
Variable Importances:
   variable relative_importance scaled_importance percentage
1  Distance         1379.994019          1.000000   0.501313
2     Month          970.739441          0.703437   0.352643
3 DayOfWeek          402.02[illegible]          0.291323   0.146044
```


It looks like distance is much more important than month or day of week when it com[illegible] delayed. Of course, according to this very basic model given the default parameters. A[illegible] supports XGBoost. It is trivial to swap GBM for XGBoost in this phase and see how the model changes with default hyperparameters:

```
xgBoostModel <- h2o.xgboost(x=c("Month", "DayOfWeek", "Distance"), y="IsArrDelayed"
```

Prediction

Prediction is very simple as well. Calling `h2o.predict(model, data)` , where `model` is the variable pointing to the model trained and `data` is the `H2OFrame` with data to do the prediction on. To test the prediction is functional in a very simple way, let's use the `gbmModel` and let it predict the original training dataset.

```
> h2o.predict(gbmModel, airlinesTrainData)
  |===============================================================================
  predict        NO       YES
1     YES 0.1419937 0.8580063
2     YES 0.1015739 0.8984261
3     YES 0.2036055 0.7963945
4     YES 0.1239904 0.8760096
5     YES 0.1384360 0.8615640
6     YES 0.1419937 0.8580063

[43978 rows x 3 columns]
```

The prediction is not very accurate in case there was no delay. This is expected, as the model is very basic. Of course, the confusion matrix seen earlier in this tutorial gave out the information about such "bad" performance beforehand.

Getting Started in Python

In order to get started in Python, only few lines of code are required. A common way of installing dependencies in python is `pip` or `anaconda` . In this tutorial, `pip` is preferred due to most users being familiar with it. If you'd like to use Conda, please follow the tutorial in H2O documentation. Python 2.7 up to Python 3.x are supported. The differences are minimal and this guide should work on both versions.

Installation

Before installing H_2O itself, a few dependencies are required. Please install them using `pip install` . On some systems, super-user privileges may be required. If so, adding `sudo` before `pip install` will solve the problem.

```
pip install requests
pip install tabulate
pip install scikit-learn
pip install colorama
pip install future
```

Once the dependencies required are installed, one last step is to install H_2O itself. Installation of the latest stable release is done as demonstrated in the following snippet. During the installation, a one-time download of the H_2O backend containing all the algorithms and computing know-how will occur.


```
pip install -f http://h2o-release.s3.amazonaws.com/h2o/latest_sta
```

That's it. H_2O is now installed and ready to be used. As a first step, it is required to tell Python to import the H_2O module with `import h2o` command. Once the module is imported, instruct H_2O to start itself by calling `h2o.init()` . Both commands are placed in the following code snippet for clarity. The process of setup is very similar to R.

```
import h2o
h2o.init()
```

The `h2o.init()` command is pretty smart and does a lot of things. First, an attempt is made to search for an existing H_2O instance being started already, before starting a new one. When none is found automatically or specified manually with argument available, a new instance of H_2O is started. As this is a fresh installation and it is highly unlikely there is an instance of H_2O already running in your environment, a new instance is started right away. During startup, H2O is going to print some useful information. Version of the Python it is running on, H_2O's version, how to connect to H2O's Flow interface or where error logs reside, just to name a few. As usual, an example of H_2O's output during startup is to be found below this text in the snippet.

```
>>> h2o.init()
Checking whether there is an H2O instance running at http://localhost:54321..... n
Attempting to start a local H2O server...
  Java Version: java version "1.8.0_171"; Java(TM) SE Runtime Environment (build 1
  Starting server from /usr/local/lib/python2.7/dist-packages/h2o/backend/bin/h2o.
  Ice root: /tmp/tmp7R8OvB
  JVM stdout: /tmp/tmp7R8OvB/h2o_pavel_started_from_python.out
  JVM stderr: /tmp/tmp7R8OvB/h2o_pavel_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321... successful.
versionFromGradle='3.19.0',projectVersion='3.19.0.99999',branch='pavel_pubdev-5336
--------------------------  ----------------------------------------
H2O cluster uptime:         01 secs
H2O cluster timezone:       Europe/Prague
H2O data parsing timezone:  UTC
H2O cluster version:        3.19.0.99999
H2O cluster version age:    3 months and 5 days
H2O cluster name:           H2O_from_python_pavel_oy9g50
H2O cluster total nodes:    1
H2O cluster free memory:    5.207 Gb
H2O cluster total cores:    8
H2O cluster allowed cores:  8
H2O cluster status:         accepting new members, healthy
H2O connection url:         http://127.0.0.1:54321
H2O connection proxy:
H2O internal security:      False
H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
Python version:             2.7.15 candidate
--------------------------  ----------------------------------------
```


Data Import

Let's import a dataset and train a model on it very quickly.

```
airlines_train_data = h2o.import_file("https://s3.amazonaws.com/h2o-airlines-unpac
```

H_2O will automatically download the dataset and parse it. It will also try to guess the datatype of each column automatically. H_2O does a great job at datatype recognition, however, each decision can be overridden manually by the user, if required. The imported dataset can also be given a name using `destination_frame` argument. For example, `h2o.import_file("https://s3.amazonaws.com/h2o-airlines-unpacked/allyears2k.csv", destination_frame='airlines_train')` imports the very same dataset with airplane delays that can be further addressed by name `airlines_train` , even from other interfaces like Python, another R console, Flow, Java, or direct API calls. If no name is provided, H_2O will generate an artificial name. Simply put, an imported dataset is called `Frame` in H_2O. List of frames can be shown by using the `h2o.ls()` function. An example output of calling `h2o.ls()` function can be found in the following code snippet. The very first record in the example is a named frame, the second one is a frame name generated automatically by H_2O.

```
>>> h2o.ls()
              key
0  airlines_train
1  allyears2k.hex
```

A preview of the data imported can be displayed with by typing the variable pointing to the H_2O Frame, in this case `airlines_train_data` .

```
>>> airlines_train_data
  Year Month DayofMonth DayOfWeek DepTime CRSDepTime ArrTime CRSArrTime UniqueCarr
1 1987    10         14         3     741        730     912        849
2 1987    10         15         4     729        730     903        849
3 1987    10         17         6     741        730     918        849
4 1987    10         18         7     729        730     847        849
5 1987    10         19         1     749        730     922        849
6 1987    10         21         3     728        730     848        849
  Origin Dest Distance TaxiIn TaxiOut Cancelled CancellationCode Diverted CarrierDe
1    SAN  SFO      447    NaN     NaN         0               NA        0
2    SAN  SFO      447    NaN     NaN         0               NA        0
3    SAN  SFO      447    NaN     NaN         0               NA        0
4    SAN  SFO      447    NaN     NaN         0               NA        0
5    SAN  SFO      447    NaN     NaN         0               NA        0
6    SAN  SFO      447    NaN     NaN         0               NA        0
  IsDepDelayed
1          YES
2           NO
3          YES
4           NO
5          YES
6           NO
```

Model Training

On top of the data imported, a model can be built quickly. There are many algorithms available in H2O. For the purpose of this tutorial, a widely known Gradient Boosting Machines method will be that is able to predict if the plane arrives late based on month, day of week, and distan before reaching its destination. GBM resides in `h2o.estimators.gbm` package. The first step is to import `H2OGradientBoostingEstimator` to avoid the need for typing a fully qualified name in future.

```
from h2o.estimators.gbm import H2OGradientBoostingEstimator
```

First, it is required to construct a new GBM estimator instance by calling `gbm_model = H2OGradientBoostingEstimator()` constructor. By invoking `gbm_model.train(...)` , H_2O will run a gradient boosting algorithm on the data. There are many variables to play with, and each and every data scientist can explore on his/her own. Overriding the default hyperparameters would only make this tutorial more complicated. H2O only needs to know three things:

- predictor columns,
- response variable column,
- training frame — a dataset to train the model on.

```
gbm_model = H2OGradientBoostingEstimator()
gbm_model.train(x = ["Month", "DayOfWeek", "Distance"], y = "IsArrDelayed", traini
```

Nothing more. It is even able to guess the distribution of the response variable, event hough as stated before, everything can be overridden manually by the data scientist, if required. After a model is trained, basic information about the model can be shown just by typing the name of the variable pointing to the trained model, in this case `gbm_model` . In the next figure, there is a demonstration of previous steps regarding model training, with H_2O's output included.

```
>>> from h2o.estimators.gbm import H2OGradientBoostingEstimator
>>> gbm_model = H2OGradientBoostingEstimator();
>>> gbm_model.train(x = ["Month", "DayOfWeek", "Distance"], y = "IsArrDelayed", tr
gbm Model Build progress: |██████████████████████████████████████████████████████
>>> gbm_model
Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1529844691141_1


ModelMetricsBinomial: gbm
** Reported on train data. **

MSE: 0.234966307798
RMSE: 0.484733233643
LogLoss: 0.660935092712
Mean Per-Class Error: 0.41540494848
AUC: 0.623776525749
Gini: 0.247553051497
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.43158830279:
       NO    YES    Error    Rate
-----  ----  -----  -------  -----------------
NO     604   18933  0.9691   (18933.0/19537.0)
YES    232   24209  0.0095   (232.0/24441.0)
Total  836   43142  0.4358   (19165.0/43978.0)
```

```
Maximum Metrics: Maximum metrics at their respective thresholds

metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                       0.431588     0.716423  368
max f2                       0.355217     0.862241  395
max f0point5                 0.513454     0.633681  278
max accuracy                 0.511993     0.594661  279
max precision                0.972534     1         0
max recall                   0.347469     1         397
max specificity              0.972534     1         0
max absolute_mcc             0.605839     0.175948  151
max min_per_class_accuracy   0.539888     0.582464  234
max mean_per_class_accuracy  0.547177     0.584595  225
Gains/Lift Table: Avg response rate: 55,58 %

    group    cumulative_data_fraction    lower_threshold    lift      cumulative_l
--  -------  --------------------------  -----------------  --------  ------------
    1        0.0100732                   0.897096           1.70593   1.70593
    2        0.0221247                   0.864273           1.64658   1.6736
    3        0.0302651                   0.83021            1.54302   1.63848
    4        0.040111                    0.753167           1.40873   1.58208
    5        0.0514803                   0.696079           1.34952   1.53072
    6        0.103688                    0.649372           1.30093   1.41502
    7        0.150348                    0.618087           1.225     1.35605
    8        0.20101                     0.600111           1.17103   1.30942
    9        0.3004                      0.578542           1.07854   1.23303
    10       0.400632                    0.557964           1.03234   1.18282
    11       0.500568                    0.540445           1.00633   1.14758
    12       0.606894                    0.525153           0.973175  1.11703
    13       0.701874                    0.508853           0.921     1.0905
    14       0.800673                    0.492089           0.869653  1.06325
    15       0.90261                     0.465819           0.80997   1.03465
    16       1                           0.279442           0.678906  1

Scoring History:
     timestamp            duration    number_of_trees    training_rmse    training_
---  -------------------  ----------  -----------------  ---------------  ---------
     2018-06-24 15:03:17  0.087 sec   0.0                0.49688163904    0.686916
     2018-06-24 15:03:17  0.368 sec   1.0                0.495487418342   0.684101
     2018-06-24 15:03:18  0.487 sec   2.0                0.494360221437   0.681803
     2018-06-24 15:03:18  0.5[illegible] sec   3.0                0.493435134409   0.679892
     2018-06-24 15:03:18  0.659 sec   4.0                0.492679667171   0.678305
---  ---                  ---         ---                ---
     2018-06-24 15:03:19  1.667 sec   46.0               0.48505[illegible]
     2018-06-24 15:03:19  1.688 sec   47.0               0.484945381131   0.661377
     2018-06-24 15:03:19  1.710 sec   48.0               0.484866365041   0.661216
```

```
     2018-06-24 15:03:19  1.710 sec   48.0             0.484866365041   0.661216
     2018-06-24 15:03:19  1.734 sec   49.0             0.484810585256   0.661098
     2018-06-24 15:03:19  1.760 sec   50.0             0.484733233643   0.660935

See the whole table with table.as_data_frame()
Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
Distance    1379.99                1                    0.501313
Month       970.739                0.703437             0.352643
DayOfWeek   402.024                0.291323             0.146044
```

Overall, this model is not expected to perform very well, given the huge error rate to be observed in the confusion matrix. By playing with different GBM hyperparameters and including different predictors into the model, much better results can be achieved. As a data scientist, the task of making the model perform better is easy for you, that's certain. To get detailed information about model and its scoring history, invoke the `print(gbm_model)` command. The output contains table with importances of variables (relative, percentage, scaled) taken into account in the model. Also, a detailed scoring history is available, as well as basic measures like mean squared error (MSE). As you begin exploring H_2O, the reference guide will guide you through all the H_2O's functionality. A shortened example of a detailed view on GBM model is to be found in the next figure. Some of the text was omitted.

```
print(gbm_model)
Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1529844691141_1



ModelMetricsBinomial: gbm
** Reported on train data. **

Scoring History:
     timestamp            duration    number_of_trees    training_rmse    training_
---  -------------------  ----------  -----------------  ---------------  ---------
     2018-06-24 15:03:17  0.087 sec   0.0                0.49688163904    0.686916
     2018-06-24 15:03:17  0.368 sec   1.0                0.495487418342   0.684101
     2018-06-24 15:03:18  0.487 sec   2.0                0.494360221437   0.681803
     2018-06-24 15:03:18  0.571 sec   3.0                0.493435134409   0.679892
     2018-06-24 15:03:18  0.659 sec   4.0                0.492679667171   0.678305
---  ---                  ---         ---                ---              ---
     2018-06-24 15:03:19  1.667 sec   46.0               0.485057037114   0.661619
     2018-06-24 15:03:19  1.688 sec   47.0               0.484945381131   0.661377
     2018-06-24 15:03:19  1.710 sec   48.0               0.484866365041   0.661216
     2018-06-24 15:03:19  1.734 sec   49.0               0.484810
     2018-06-24 15:03:19  1.760 sec   50.0               0.484733

See the whole table with table.as_data_frame()
```

```
Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
Distance    1379.99                1                    0.501313
Month       970.739                0.703437             0.352643
DayOfWeek   402.024                0.291323             0.146044
```

Looks like distance is much more important than month or day of week when it comes to the plane being delayed. Of course, according to this very basic model given the default parameters. A pro-tip at the end: H_2O supports XGBoost. It is trivial to swap GBM for XGBoost in this phase and see how the model changes with default hyperparameters:

```
from h2o.estimators.xgboost import H2OXGBoostEstimator
xgb_model.train(x = ["Month", "DayOfWeek", "Distance"], y = "IsArrDelayed", traini
```

Prediction

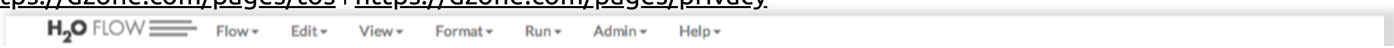
Once a model is created, predictions are simply done by calling `predict(data)` method on a model, where the `data` argument is the variable pointing to an `H2OFrame` with data to do the prediction on. To test the prediction is functional in a very simple way, let's use the `gbm_model` and let it predict the original training dataset by issuing `gbm_model.predict(airlines_train_data)` command.

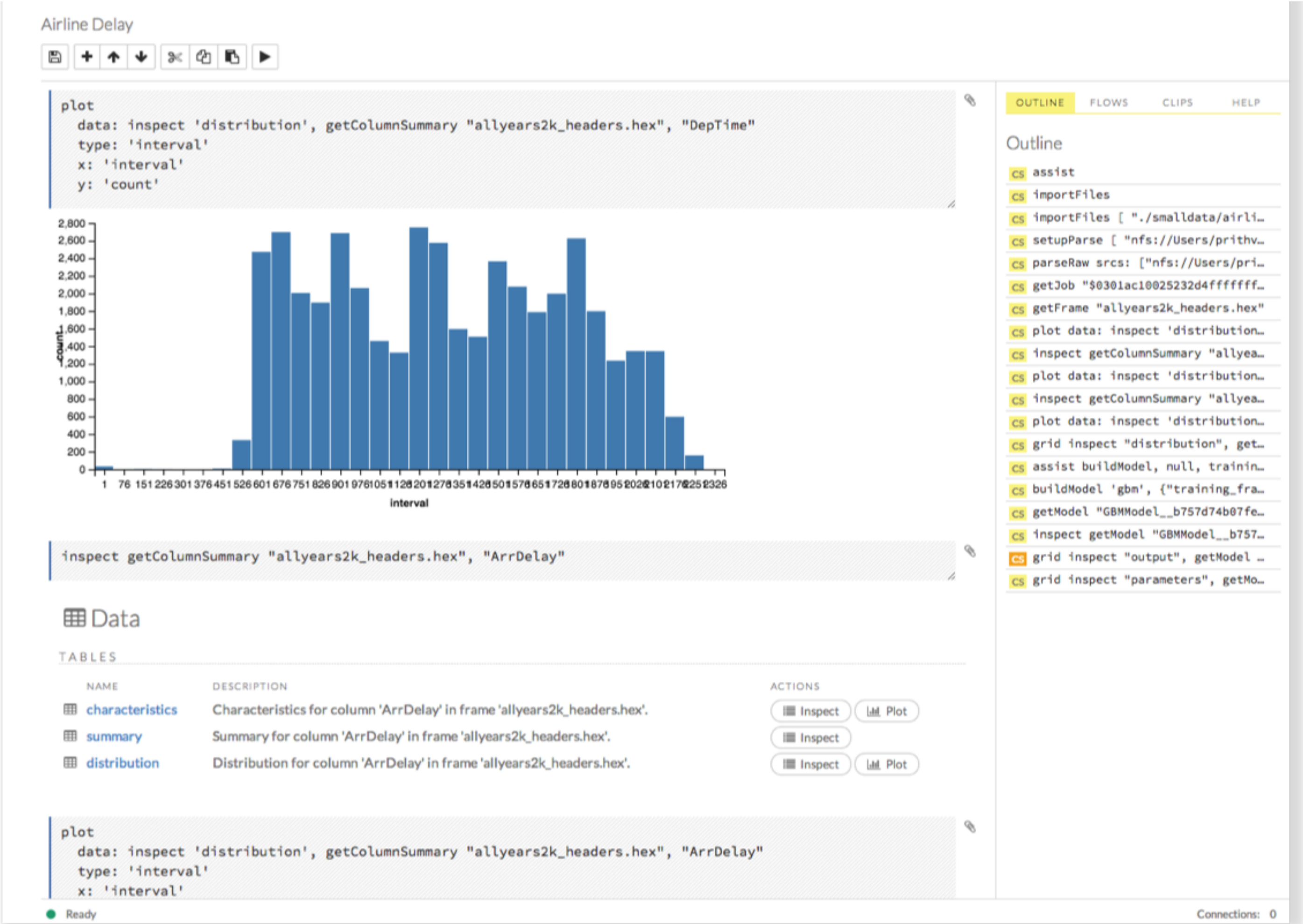
```
>>> gbm_model.predict(airlines_train_data)
gbm prediction progress: |████████████████████████████████████████
predict          NO       YES
---------  --------  --------
YES        0.141994  0.858006
YES        0.101574  0.898426
YES        0.203606  0.796394
YES        0.12399   0.87601
YES        0.138436  0.861564
YES        0.141994  0.858006
YES        0.101574  0.898426
YES        0.103421  0.896579
YES        0.203606  0.796394
YES        0.12399   0.87601

[43978 rows x 3 columns]
```

A result of the prediction is a `H2OFrame`. Pointer to it can be saved into a variable as well, e.g. `prediction = gbm_model.predict(airlines_train_data)`. The table printed is only a preview of the first few predictions made. As the above example demonstrates, the prediction is not very accurate in case there was no delay. This is expected, as the model is very basic. Of course, the confusion matrix seen earlier in this tutorial gave out the information about such "bad" performance beforehand.

Getting Started With Flow

Flow is H_2O's web interface. It is very powerful and oriented on visuals. It has it all. I
new models, visualizing, and refining existing achievements to scoring.





Flow is active whenever H_2O is started. If you've followed previous Python or R tutorials, during the active Python or R session, Flow can be reached from your browser. Whenever `h2o.init()` is called, Flow is started with H_2O. In the output after `h2o.init()`, the URL and are printed. When ran locally, the Flow is usually bound to `localhost`, with the default port being `54321`. Address or port may be changed by `h2o.init()` arguments, e.g. `h2o.init(ip='127.0.0.1', port='10001')`.

```
>>> h2o.init()
Checking whether there is an H2O instance running at http://localhost:54321..... n
Attempting to start a local H2O server...
//Some output omitted
--------------------------  ----------------------------------------
H2O cluster uptime:         01 secs
H2O cluster timezone:       Europe/Prague
H2O data parsing timezone:  UTC
H2O cluster version:        3.19.0.99999
H2O cluster version age:    3 months and 5 days
H2O cluster name:           H2O_from_python_pavel_oy9g50
H2O cluster total nodes:    1
H2O cluster free memory:    5.207 Gb
H2O cluster total cores:    8
H2O cluster allowed cores:  8
H2O cluster status:         accepting new members, healthy
H2O connection url:         http://127.0.0.1:54321          <<<<--- URL to FLOW
H2O connection proxy:
```

```
H2O internal security:      False
H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
Python version:             2.7.15 candidate
--------------------------  ----------------------------------------
```

From the shortened output, the actual URL can be observed: `H2O connection url: http://127.0.0.1:54321`. As `127.0.0.1` resolves to localhost, copying the given URL or simply typing `localhost:54321` into a web browser opens H2O Flow.

Starting Flow Without Python/R

There is no need to install the Python module or R library in order to run H_2O. Just download the latest stable H2O release and run it. Java is required to run H_2O.

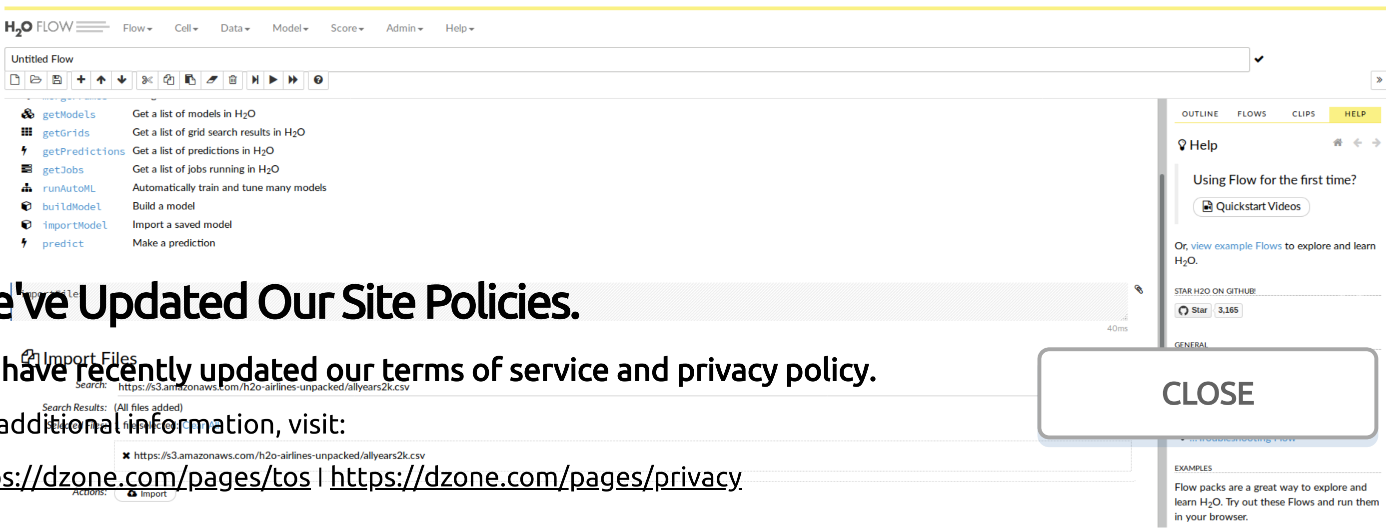
1. Download package with latest stable release
2. Unpack it
3. Run java -jar h2o.jar

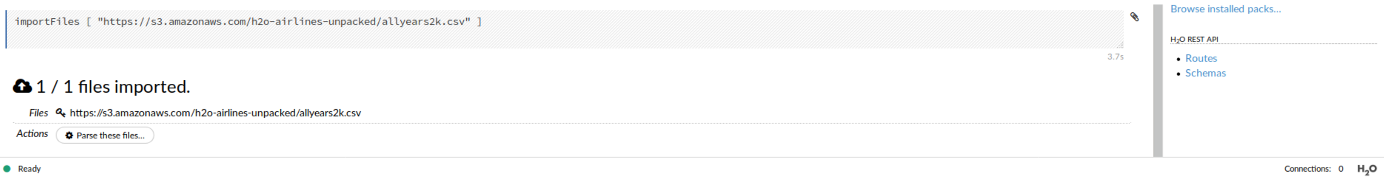
The `h2o.jar` file is to be found in the extracted directory. If you're on Windows, double-clicking the h2o.jar file should be sufficient to run it if there is Java installed. At the time this tutorial is written, Java version 7 is the minimum required version. Java 8 is recommended.

Importing Data With Flow

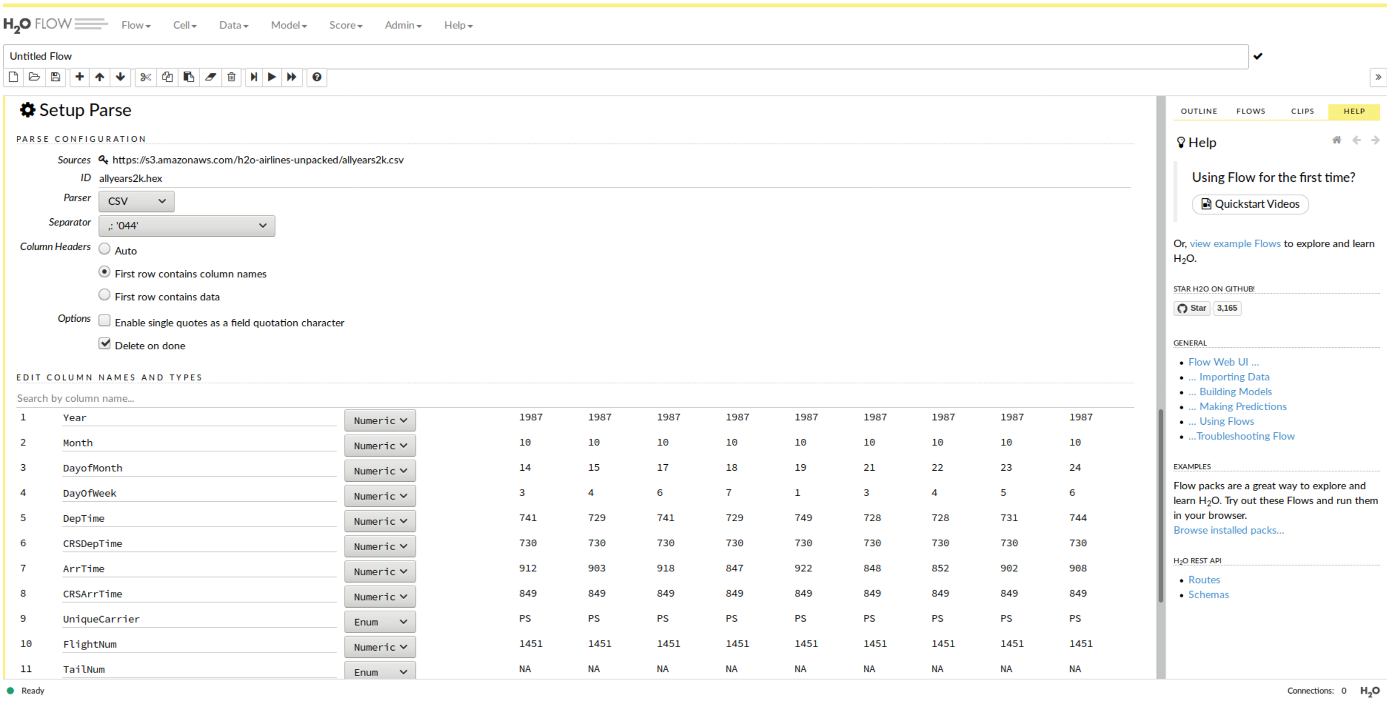
On top of the page, click on `Data`. As the menu appears, choose `Import files`. Do not interchange with "Upload file" option, which is only useful to upload files from your very own machine. After clicking on the `Import files` option, a form appears at the bottom of the page. Please fill the `Search` input box with `https://s3.amazonaws.com/h2o-airlines-unpacked/allyears2k.csv` to download the Airlines dataset described at the beginning of this chapter. By clicking on the `magnifying glass` next to the input box, H2O is going to verify that it can reach the file. This way, even local files or files from remote filesystems (including Hadoop filesystem) can be imported. By inserting whole folders, H2O will pop-up all the files available, thus enabling multi-file import with ease. In this simple case, after clicking on the `magnifying glass`, a single file named exactly as the URL inserted into the input box will appear a little bit lower. By clicking on the `plus icon`, the file is marked for import. By clicking `Import button`, H2O will automatically download the file.

After the file is imported, another form will appear. This time, H_2O confirms that the file has been imported and asks to user to starting parsing it. By clicking on `Parse these files`, H2O will begin parsing.

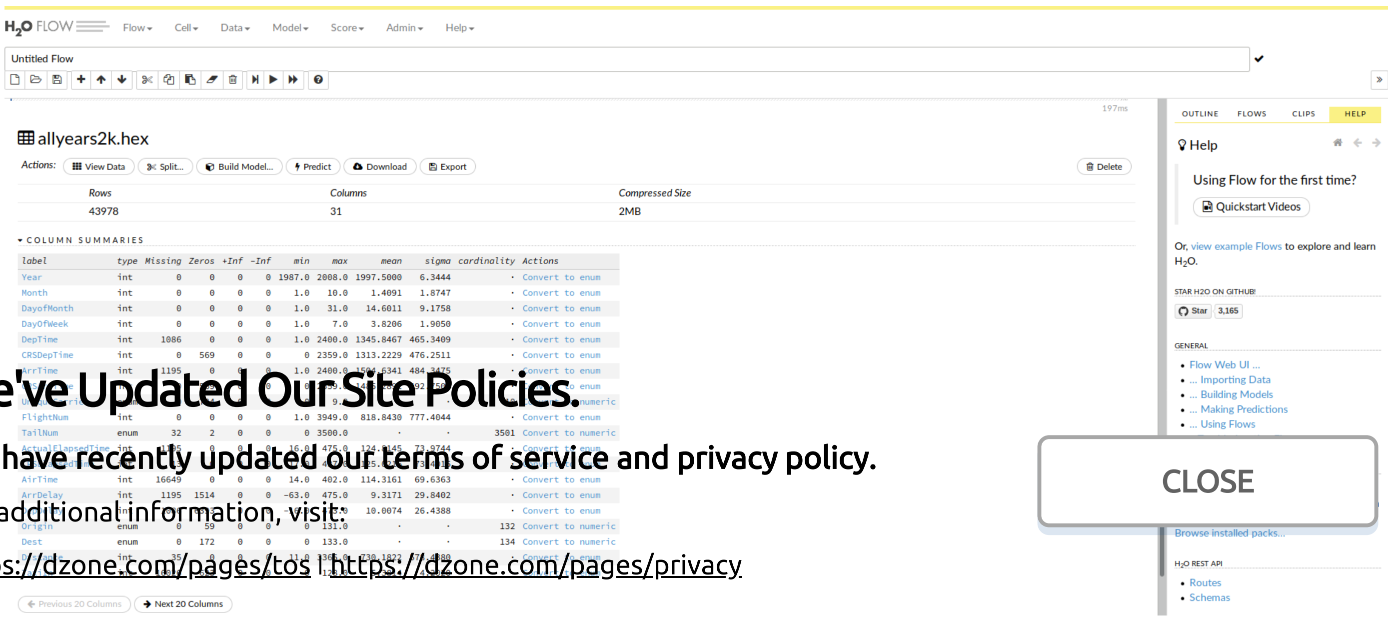




However, H2O is not going to parse the whole file right away. The could be potentially very big (or there may be multiple files). Instead, it will automatically detect the format and suggests best parsing settings based on internal heuristics. In case of CSV imported in this tutorial, it will correctly detect the format, set separator to be a comma, detect column headers on first row. However, there is more — columns datatypes are detected. Yet the data scientist has the power to override every decision, e.g. change column type. By clicking on `Parse` button at the very bottom, the file is finally parsed by H_2O.



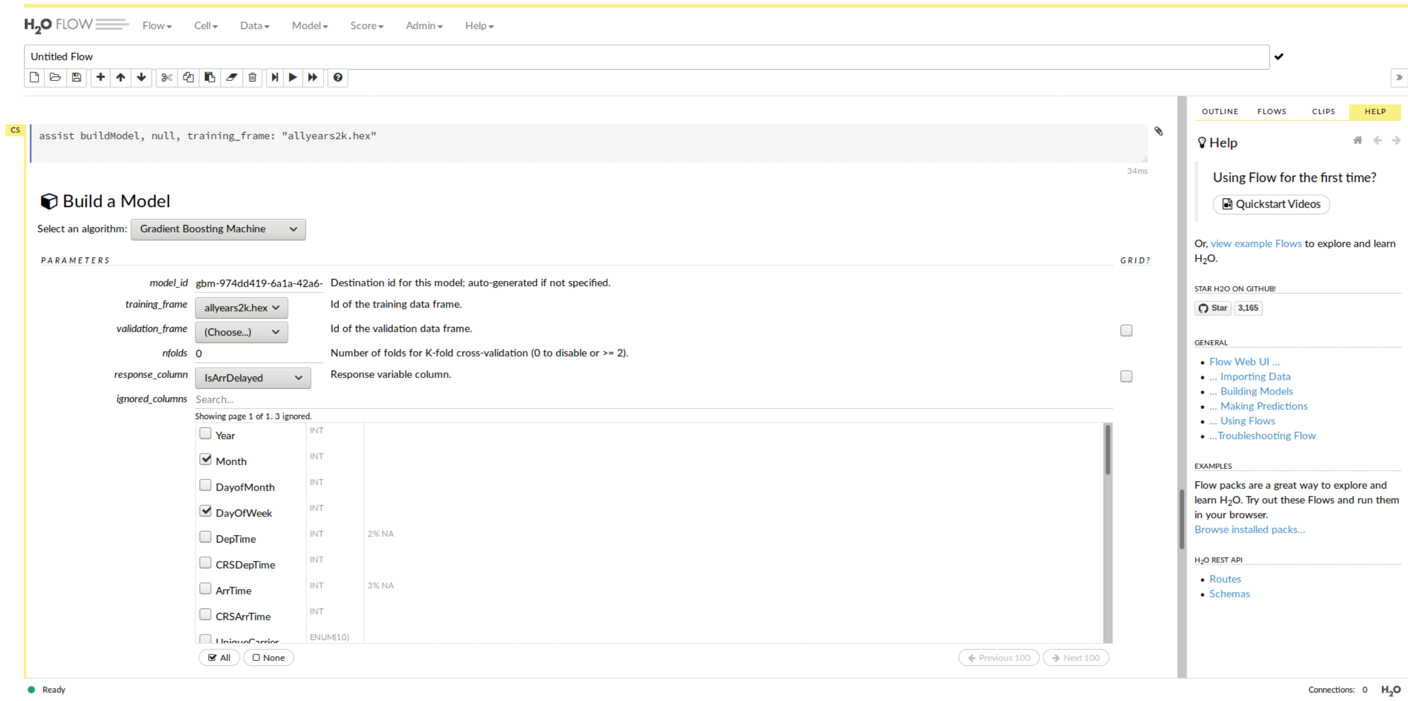
The sample dataset is very small (4,4 MB) and H_2O's CSV parser is very fast, so the parsing should be quick. After the parsing phase, H_2O will inform you about the dataset being ready for an inspection. By clicking on the `View` button, a table view preview of the parsed data appears. Here, the user has several choices, including viewing all the data, exporting it, splitting it into several smaller chunks (e.g.for testing), or most importantly to `Build Model`.



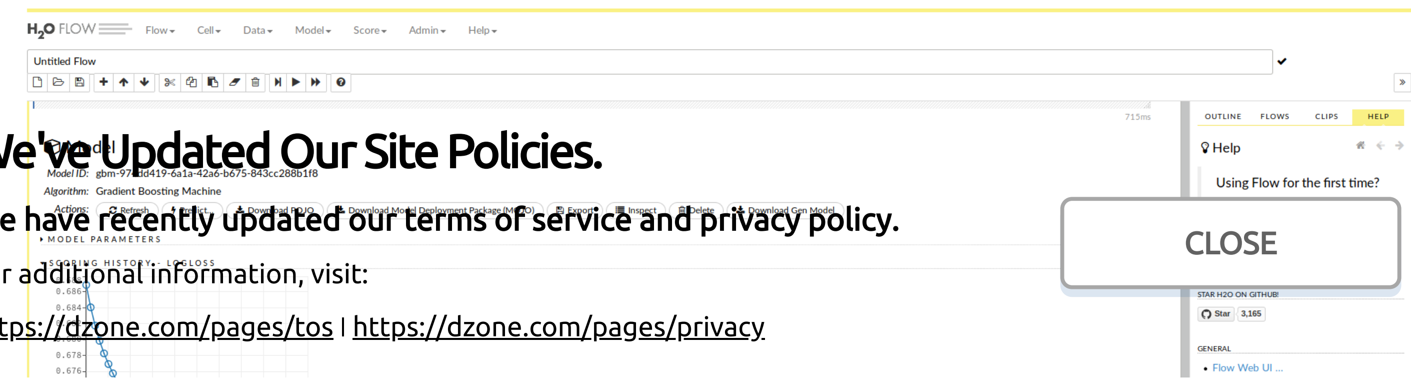
By clicking on the `Build model` button above the data overview, a dialogue asking the user to select an algorithm appears. In this tutorial, GBM is used as a simple example, therefore, select `Gradient Boosting Machine` . As with Python and R, let's train a model that is able to predict if the plane arrives late based on month, day of week, and distance the plane has to travel before reaching its destination. From top to bottom, only few of the choices need your attention right now. Most of these are hyperparameters, which are of course important in real world, yet for a getting started guide, the default values will suffice.

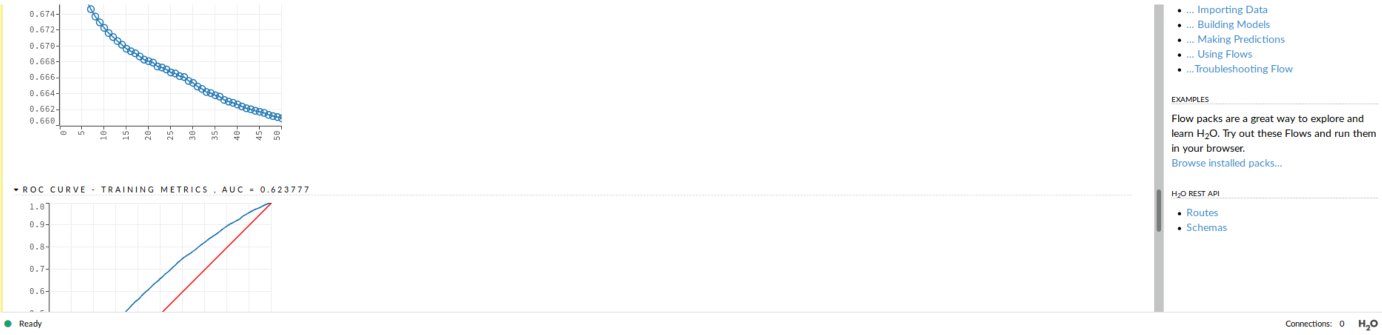
The `training_frame` option is already pre-set by flow to point to the data imported. Creating validation frames from training is easy using the following option, which we'll leave intact for now. Since this model is trying to predict if the plane will be delayed on arrival ot its destination, the response column should be set to `IsArrDelayed` . The prediction is based on month, day of week, and traveling distance (chosen arbitrarily). Therefore, in the ignored column section, first check all the columns as ignored and leave only `Month` , `DayOfWeek` , `Distance` , and `IsArrDelayed` (response variable) unchecked.

After the parameters are set, click on `Build model` button at the bottom to start the model training.



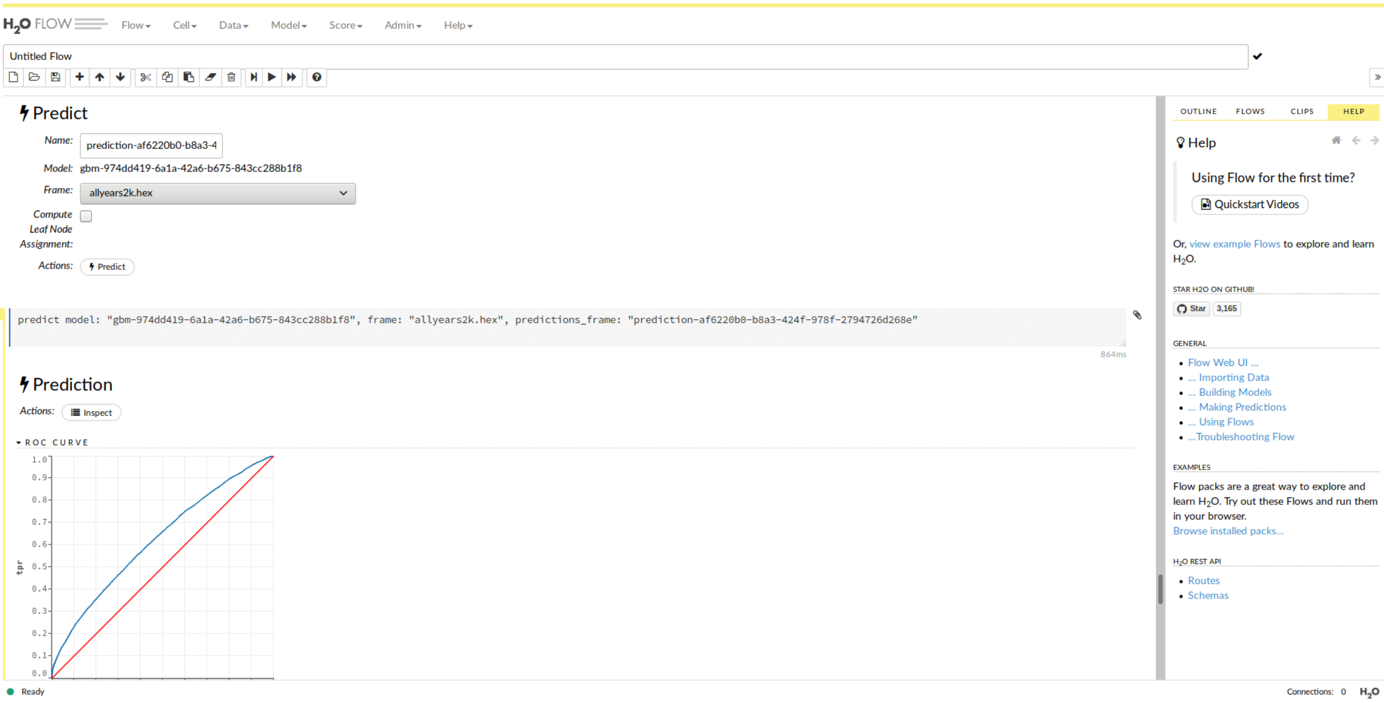
Model training in this small case should be done in an instant. H_2O will again display a message about training progress and after the training is finished, the model can be displayed by clicking on the `View` button. A complete model description appears. Variable importance, logloss, complete history per each round of training (including durations), and much more can be observed.





Predicting With Flow

After the model is built, click the `Predict` button (with a lightning icon) on top of model summary reached in previous steps. To test the prediction is functional in a very simple way, let's use the current model and let it predict the original training dataset. After clicking on the `Predict` button, select the training frame in the dropdown menu's `Frame` option and click on `Predict` . The resulting predictions appear, including various metrics. The model is not very accurate, especially the error rate regarding prediction of a plane not arriving late are very inaccurate. Creating a better model by tuning the hyperparameters or playing with various predictors is up to each and every data scientist to try now.



Where to Go Next?

In this hands-on guide, very basic ways to interact with H_2O were shown. However, there is so much more to H_2O. You can take H_2O from your laptop to large clusters, creating models and predicting on extremely large datasets. Cross-validation, stacked ensembled, tuning parameters with grid search and many, many other expected functionality. After data scientists tune the model, deploying it into production is easy with H_2O's POJO/MOJO functionality. This way, the model is simply exported into a self-containing package, making it easy for the engineers to plug it into any Java-based production environment. Model's performance can be observed even in production at real-time, providing useful reports.

Data scientists and developers are best to visit H2O documentation. It provides step-b... users as well as guides for users interested specifically in R language or Python langu... overview of what H2O is capable of. Besides a list of available algorithms, methods of cross-validation, or the ability to build models on top of existing ones, ways of productionizing created models quickly are described.

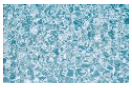
In the documentation, everything from data import, exploration, and filtering to deployment into production is described. Running on Hadoop? No problem. Using Apache Spark? Sparkling water is at your service. H2O also integrates well with various cloud services. H2O also offers countless tutorials on GitHub.

Video tutorials, speeches, and expert advices are available on H2O's YouTube channel. There is so much to explore.

H_2O also holds conferences named H2O World. Come and visit us, we'll be glad to talk to you.

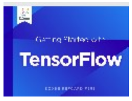
Like This Article? Read More From DZone

Inspecting Decision Trees in H2O



What's New in H2O Machine Learning: Christmas 2018

How We Built Tagger News: Machine Learning on a Tight Schedule



Free DZone Refcard
Introduction to TensorFlow

Topics: ARTIFICAL INTELLIGENCE , MACHINE LEARNING , H2O.AI , PYTHON , R , XGBOOST , GBM , STATISTICS

Published at DZone with permission of Pavel Pscheidl . See the original article here.
Opinions expressed by DZone contributors are their own.

IN PROGRESS