Machine Learning with Python and H2O

Pasha Stetsenko

Edited by: Angela Bartz

---

http://h2o.ai/resources/

June 2017: Fifth Edition

Machine Learning with Python and H2O
by Pasha Stetsenko
with assistance from Spencer Aiello,
Cliff Click, Hank Roark, & Ludi Rehak
Edited by: Angela Bartz

Published by H2O.ai, Inc.
2307 Leghorn St.
Mountain View, CA 94043

June 2017: Fifth Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

1 Introduction

This documentation describes how to use H2O from Python. More information on H2O's system and algorithms (as well as complete Python user documentation) is available at the H2O website at `http://docs.h2o.ai`.

H2O Python uses a REST API to connect to H2O. To use H2O in Python or launch H2O from Python, specify the IP address and port number of the H2O instance in the Python environment. Datasets are not directly transmitted through the REST API. Instead, commands (for example, importing a dataset at specified HDFS location) are sent either through the browser or the REST API to perform the specified task.

The dataset is then assigned an identifier that is used as a reference in commands to the web server. After one prepares the dataset for modeling by defining significant data and removing insignificant data, H2O is used to create a model representing the results of the data analysis. These models are assigned IDs that are used as references in commands.

Depending on the size of your data, H2O can run on your desktop or scale using multiple nodes with Hadoop, an EC2 cluster, or Spark. Hadoop is a scalable open-source file system that uses clusters for distributed storage and dataset processing. H2O nodes run as JVM invocations on Hadoop nodes. For performance reasons, we recommend that you do not run an H2O node on the same hardware as the Hadoop NameNode.

H2O helps Python users make the leap from single machine based processing to large-scale distributed environments. Hadoop lets H2O users scale their data processing capabilities based on their current needs. Using H2O, Python, and Hadoop, you can create a complete end-to-end data analysis solution.

This document describes the four steps of data analysis with H2O:

1. installing H2O
2. preparing your data for modeling
3. creating a model using simple but powerful machine learning algorithms
4. scoring your models

2 What is H2O?

H2O.ai is focused on bringing AI to businesses through software. Its flagship product is H2O, the leading open source platform that makes it easy for financial services, insurance companies, and healthcare companies to deploy AI and deep learning to solve complex problems. More than 9,000 organizations and 80,000+ data scientists depend on H2O for critical applications like predictive maintenance and operational intelligence. The company – which was recently named to the CB Insights AI 100 – is used by 169 Fortune 500 enterprises, including 8 of the world's 10 largest banks, 7 of the 10 largest insurance companies, and 4 of the top 10 healthcare companies. Notable customers include Capital One, Progressive Insurance, Transamerica, Comcast, Nielsen Catalina Solutions, Macy's, Walgreens, and Kaiser Permanente.

Using in-memory compression, H2O handles billions of data rows in-memory, even with a small cluster. To make it easier for non-engineers to create complete analytic workflows, H2O's platform includes interfaces for R, Python, Scala, Java, JSON, and CoffeeScript/JavaScript, as well as a built-in web interface, Flow. H2O is designed to run in standalone mode, on Hadoop, or within a Spark Cluster, and typically deploys within minutes.

H2O includes many common machine learning algorithms, such as generalized linear modeling (linear regression, logistic regression, etc.), Naïve Bayes, principal components analysis, k-means clustering, and word2vec. H2O implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. H2O also includes a Stacked Ensembles method, which finds the optimal combination of a collection of prediction algorithms using a process known as "stacking." With H2O, customers can build thousands of models and compare the results to get the best predictions.

H2O is nurturing a grassroots movement of physicists, mathematicians, and computer scientists to herald the new wave of discovery with data science by collaborating closely with academic researchers and industrial data scientists. Stanford university giants Stephen Boyd, Trevor Hastie, and Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. And with hundreds of meetups over the past several years, H2O continues to remain a word-of-mouth phenomenon.

Try it out

- Download H2O directly at `http://h2o.ai/download`.
- Install H2O's R package from CRAN at `https://cran.r-project.org/web/packages/h2o/`.

- Install the Python package from PyPI at `https://pypi.python.org/pypi/h2o/`.

Join the community

- To learn about our training sessions, hackathons, and product updates, visit `http://h2o.ai`.
- To learn about our meetups, visit `https://www.meetup.com/topics/h2o/all/`.
- Have questions? Post them on Stack Overflow using the **h2o** tag at `http://stackoverflow.com/questions/tagged/h2o`.
- Have a Google account (such as Gmail or Google+)? Join the open source community forum at `https://groups.google.com/d/forum/h2ostream`.
- Join the chat at `https://gitter.im/h2oai/h2o-3`.

2.1 Example Code

Python code for the examples in this document is located here:

`https://github.com/h2oai/h2o-3/tree/master/h2o-docs/src/booklets/v2_2015/source/Python_Vignette_code_examples`

2.2 Citation

To cite this booklet, use the following:

Aiello, S., Cliff, C., Roark, H., Rehak, L., Stetsenko, P., and Bartz, A. (Jun 2017). *Machine Learning with Python and H2O*. `http://h2o.ai/resources/`.

3 Installation

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

The easiest way to directly install H2O is via a Python package.

3.1 Installation in Python

To load a recent H2O package from PyPI, run:

```
pip install h2o
```

To download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in Python" tab.
4. Copy and paste the commands into your Python session.

After H2O is installed, verify the installation:

```
import h2o

# Start H2O on your local machine
h2o.init()

# Get help
help(h2o.estimators.glm.H2OGeneralizedLinearEstimator)
help(h2o.estimators.gbm.H2OGradientBoostingEstimator)

# Show a demo
h2o.demo("glm")
h2o.demo("gbm")
```

4 Data Preparation

The next sections of the booklet demonstrate the Python interface using examples, which include short snippets of code and the resulting output.

In H2O, these operations all occur distributed and in parallel and can be used on very large datasets. More information about the Python interface to H2O can be found at `docs.h2o.ai`.

Typically, we import and start H2O on the same machine as the running Python process:

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

To connect to an established H2O cluster (in a multi-node Hadoop environment, for example):

```
h2o.init(ip="123.45.67.89", port=54321)
```

To create an H2OFrame object from a Python tuple:

```
df = h2o.H2OFrame(zip(*((1, 2, 3), ('a', 'b', 'c'), (0.1, 0.2, 0.3))))

# View the H2OFrame
df

#   C1  C2      C3
# ----  ----  ----
#    1  a       0.1
#    2  b       0.2
#    3  c       0.3
#
# [3 rows x 3 columns]
```

To create an H2OFrame object from a Python list:

```
df = h2o.H2OFrame(zip(*[[1, 2, 3], ['a', 'b', 'c'], [0.1, 0.2, 0.3]]))

# View the H2OFrame
df

#   C1  C2      C3
# ----  ----  ----
#    1  a       0.1
#    2  b       0.2
#    3  c       0.3
#
# [3 rows x 3 columns]
```

To create an H2OFrame object from `collections.OrderedDict` or a Python dict:

```
df = h2o.H2OFrame({'A': [1, 2, 3],'B': ['a', 'b', 'c'],'C': [0.1, 0.2, 0.3]})

# View the H2OFrame
df

#   A    C  B
# ---  ---  ---
#   1  0.1  a
#   2  0.2  b
#   3  0.3  c
#
# [3 rows x 3 columns]
```

To create an H2OFrame object from a Python dict and specify the column types:

```
df2 = h2o.H2OFrame.from_python({'A': [1, 2, 3],
                                'B': ['a', 'a', 'b'],
                                'C': ['hello', 'all', 'world'],
                                'D': ['12MAR2015:11:00:00', '13MAR2015
                                     :12:00:00', '14MAR2015:13:00:00']},
                                 column_types=['numeric', 'enum', 'string', '
                                      time'])

# View the H2OFrame
```

```
df2

#   A  C      B              D
# ---  -----  ---  -----------
#   1  hello  a    1.42618e+12
#   2  all    a    1.42627e+12
#   3  world  b    1.42636e+12
#
# [3 rows x 4 columns]
```

To display the column types:

```
df2.types
# {u'A': u'numeric', u'B': u'string', u'C': u'enum', u'D': u'time'}
```

4.1 Viewing Data

To display the top and bottom of an H2OFrame:

```
import numpy as np
df = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
     list('ABCD'))

# View top 10 rows of the H2OFrame
df.head()

#          A           B           C          D
# ---------  ----------  ----------  ---------
# -0.613035  -0.425327   -1.92774    -2.1201
# -1.26552   -0.241526   -0.0445104   1.90628
#  0.763851   0.0391609  -0.500049    0.355561
# -1.24842    0.912686   -0.61146     1.94607
#  2.1058    -1.83995     0.453875   -1.69911
#  1.7635     0.573736   -0.309663   -1.51131
# -0.781973   0.051883   -0.403075    0.569406
#  1.40085    1.91999     0.514212   -1.47146
# -0.746025  -0.632182    1.27455    -1.35006
# -1.12065    0.374212    0.232229   -0.602646
#
# [10 rows x 4 columns]

# View bottom 5 rows of the H2OFrame
df.tail(5)

#          A           B          C          D
# ---------  ----------  ---------  ---------
#  1.00098   -1.43183    -0.322068   0.374401
#  1.16553   -1.23383    -1.71742    1.01035
# -1.62351   -1.13907     2.1242    -0.275453
# -0.479005  -0.0048988   0.224583   0.219037
# -0.74103    1.13485     0.732951   1.70306
#
# [5 rows x 4 columns]
```

To display the column names:

```
df.columns
# [u'A', u'B', u'C', u'D']
```

To display compression information, distribution (in multi-machine clusters), and summary statistics of your data:

```
df.describe()

# Rows: 100 Cols: 4
#
# Chunk compression summary:
# chunk_type     chunkname   count   count_%   size   size_%
# ------------   ---------   -----   -------   ----   ------
# 64-bit Reals    C8D        4       100      3.4 KB  100
#
# Frame distribution summary:
#                    size   #_rows   #_chunks_per_col  #_chunks
# ---------------  ------  ------   ----------------   --------
# 127.0.0.1:54321  3.4 KB  100      1                  4
# mean             3.4 KB  100      1                  4
# min              3.4 KB  100      1                  4
# max              3.4 KB  100      1                  4
# stddev           0  B    0        0                  0
# total            3.4 KB  100      1                  4
#
#            A          B          C          D
# --------   --------   --------   --------   --------
# type       real       real       real       real
# mins       -2.49822   -2.37446   -2.45977   -3.48247
# mean       -0.01062   -0.23159    0.11423   -0.16228
# maxs        2.59380    1.91998    3.13014    2.39057
# sigma       1.04354    0.90576    0.96133    1.02608
# zeros       0          0          0          0
# missing     0          0          0          0
```

4.2 Selection

To select a single column by name, resulting in an H2OFrame:

```
df['A']

#          A
# ---------
# -0.613035
# -1.265520
#  0.763851
# -1.248425
#  2.105805
#  1.763502
# -0.781973
#  1.400853
# -0.746025
# -1.120648
#
# [100 rows x 1 column]
```

To select a single column by index, resulting in an H2OFrame:

```
df[1]

#          B
# ---------
# -0.425327
# -0.241526
# 0.039161
#  0.912686
# -1.839950
#  0.573736
#  0.051883
#  1.919987
# -0.632182
#  0.374212
#
# [100 rows x 1 column]
```

To select multiple columns by name, resulting in an H2OFrame:

```
df[['B','C']]

#          B          C
# ---------  ---------
# -0.425327  -1.927737
# -0.241526  -0.044510
#  0.039161  -0.500049
#  0.912686  -0.611460
# -1.839950   0.453875
#  0.573736  -0.309663
#  0.051883  -0.403075
#  1.919987   0.514212
# -0.632182   1.274552
#  0.374212   0.232229
#
# [100 rows x 2 columns]
```

To select multiple columns by index, resulting in an H2OFrame:

```
df[0:2]

#          A          B
# ---------  ---------
# -0.613035  -0.425327
# -1.265520  -0.241526
#  0.763851   0.039161
# -1.248425   0.912686
#  2.105805  -1.839950
#  1.763502   0.573736
# -0.781973   0.051883
#  1.400853   1.919987
# -0.746025  -0.632182
# -1.120648   0.374212
#
# [100 rows x 2 columns]
```

To select multiple rows by slicing, resulting in an H2OFrame:

Note By default, H2OFrame selection is for columns, so to slice by rows and get all columns, be explicit about selecting all columns:

```
df[2:7, :]

#          A          B          C           D
# ---------  ---------  ---------  ----------
#  1.31828    0.316926   0.970535   0.218061
# -0.18547    0.207064   1.3229    -0.432614
# -0.424018  -1.72759    0.356871   0.206214
#  1.3377     1.10761   -0.280443   0.0964197
# -0.385682   0.190449   0.760816   1.92447
#
# [5 rows x 4 columns]
```

To select rows based on specific criteria, use Boolean masking:

```
df2[ df2["B"] == "a", :]

#   A  C      B              D
# ---  -----  ---  -----------
#   1  hello  a    1.42618e+12
#   2  all    a    1.42627e+12
#
# [2 rows x 4 columns]
```

4.3 Missing Data

The H2O parser can handle many different representations of missing data types, including '' (blank), 'NA', and None (Python). They are all displayed as `nan` in Python.

To create an H2OFrame from Python with missing elements:

```
df3 = h2o.H2OFrame.from_python(
    {'A': [1, 2, 3,None,''],
     'B': ['a', 'a', 'b', 'NA', 'NA'],
     'C': ['hello', 'all', 'world', None, None],
     'D': ['12MAR2015:11:00:00',None,
           '13MAR2015:12:00:00',None,
           '14MAR2015:13:00:00']},
    column_types=['numeric', 'enum', 'string', 'time'])
```

To determine which rows are missing data for a given column ('1' indicates missing):

```
df3["A"].isna()

#   C1
# ----
#    0
#    0
```

```
#    0
#    1
#    1
#
# [5 rows x 1 column]
```

To change all missing values in a column to a different value:

```
df3[ df3["A"].isna(), "A"] = 5
```

To determine the location of all missing data in an H2OFrame:

```
df3.isna()

#   C1    C2    C3    C4
# ----  ----  ----  ----
#    0     0     0     0
#    0     0     0     1
#    0     0     0     0
#    0     0     0     1
#    0     0     0     0
#
# [5 rows x 4 columns]
```

4.4 Operations

When performing a descriptive statistic on an entire H2OFrame, missing data is generally excluded and the operation is only performed on the columns of the appropriate data type:

```
df4 = h2o.H2OFrame.from_python(
    {'A': [1, 2, 3,None,''],
     'B': ['a', 'a', 'b', 'NA', 'NA'],
     'C': ['hello', 'all', 'world', None, None],
     'D': ['12MAR2015:11:00:00',None,
           '13MAR2015:12:00:00',None,
           '14MAR2015:13:00:00']},
    column_types=['numeric', 'enum', 'string', 'time'])

df4.mean(na_rm=True)
# [2.0, nan, nan, nan]
```

When performing a descriptive statistic on a single column of an H2OFrame, missing data is generally *not* excluded:

```
df4["A"].mean()
# [nan]

df4["A"].mean(na_rm=True)
# [2.0]
```

In both examples, a native Python object is returned (list and float respectively in these examples).

When applying functions to each column of the data, an H2OFrame containing the means of each column is returned:

```
df5 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
    list('ABCD'))

df5.apply(lambda x: x.mean(na_rm=True))

# H2OFrame:
#          A           B           C           D
# ----------  ----------  ----------  ----------
# 0.0304506   0.0334168  -0.0374976   0.0520486
#
# [1 row x 4 columns]
```

When applying functions to each row of the data, an H2OFrame containing the sum of all columns is returned:

```
df5.apply(lambda row: row.sum(), axis=1)

# H2OFrame:
#         C1
# ----------
# -0.388512
#  1.67669
# -2.56216
# -0.277616
#  1.13655
# -0.575992
# -3.49258
#  0.776883
# -0.778604
#  2.30617
#
# [100 rows x 1 column]
```

H2O provides many methods for histogramming and discretizing data. Here is an example using the `hist` method on a single data frame:

```
df6 = h2o.H2OFrame.from_python(np.random.randn(100,1).tolist())

df6.hist(plot=False)

# Parse Progress: [##############################] 100%
#    breaks    counts    mids_true          mids    density
# ---------  --------  -----------  -----------  ---------
# -1.51121         nan   nan          nan          0
# -0.868339          9    -1.07704    -1.18977    0.139997
# -0.225468         12    -0.73561    -0.546904   0.186663
#  0.417403         18    -0.413093    0.0959675  0.279994
#  1.06027          26    -0.10108     0.738839   0.404436
#  1.70315          22     0.214337    1.38171    0.342215
#  2.34602           7     0.607727    2.02458    0.108887
#  2.98889           6     0.860969    2.66745    0.0933313
#
# [8 rows x 5 columns]
```

H2O includes a set of string processing methods in the H2OFrame class that make it easy to operate on each element in an H2OFrame.

To determine the number of times a string is contained in each element:

```
df7 = h2o.H2OFrame.from_python(['Hello', 'World', 'Welcome', 'To', 'H2O', '
    World'])

# View the H2OFrame
df7

#  C1     C2     C3       C4    C5    C6
#  -----  -----  -------  ----  ----  -----
#  Hello  World  Welcome  To    H2O   World
#
# [1 row x 6 columns]

# Find how many times "l" appears in each string
df7.countmatches('l')

#   C1    C2    C3    C4    C5    C6
# ----  ----  ----  ----  ----  ----
#    2     1     1     0     0     1
#
# [1 row x 6 columns]
```

To replace the first occurrence of 'l' (lower case letter) with 'x' and return a new H2OFrame:

```
df7.sub('l','x')

#  C1     C2     C3       C4    C5    C6
#  -----  -----  -------  ----  ----  -----
#  Hexlo  Worxd  Wexcome  To    H2O   Worxd
```

For global substitution, use `gsub`. Both `sub` and `gsub` support regular expressions.

To split strings based on a regular expression:

```
df7.strsplit('(l)+')

# C1    C2    C3    C4    C5    C6    C7    C8    C9    C10
# ----  ----  ----  ----  ----  ----  ----  ----  ----  -----
# He    o     Wor   d     We    come  To    H2O   Wor   d
#
# [1 row x 10 columns]
```

4.5 Merging

To combine two H2OFrames together by appending one as rows and return a new H2OFrame:

```
# Create a frame of random numbers w/ 100 rows
df8 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
     list('ABCD'))

# Create a second frame of random numbers w/ 100 rows
df9 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
     list('ABCD'))

# Combine the two frames, adding the rows from df9 to df8
df8.rbind(df9)

#            A           B           C           D
# ------------  ----------  ----------  ----------
#   1.11442       1.31272     0.250418    1.73182
# -1.61876       0.428622   -1.16684    -0.032936
#   0.637249    -0.48904     1.55848     0.669266
#   0.00355574  -0.40736    -0.979222   -0.395017
#   0.218243    -0.154004   -0.219537   -0.750664
# -0.047789      0.306318    0.557441   -0.319108
# -1.45013      -0.614564    0.472257   -0.456181
# -0.594333     -0.435832   -0.0257311   0.548708
#   0.571215    -1.22759    -2.01855    -0.491638
# -0.697252     -0.864301   -0.542508   -0.152953
#
# [200 rows x 4 columns]
```

For successful row binding, the column names and column types between the two H2OFrames must match. To combine two H2O frames together by appending one as columns and return a new H2OFrame:

```
df8.cbind(df9)

#      A       B       C       D      A0     B0     C0     D0
# ------ ------- ------- ------- ------ ------ ------ ------
# -0.09   0.944   0.160   0.271 -0.351  1.66  -2.32  -0.86
# -0.95   0.669   0.664   1.535 -0.633 -1.78   0.32   1.27
#  0.17   0.657   0.970 -0.419 -1.413 -0.51   0.64  -1.25
#  0.58  -0.516  -1.598 -1.346  0.711  1.09   0.05   0.63
#  1.04  -0.281  -0.411  0.959 -0.009 -0.47   0.41  -0.52
#  0.49   0.170   0.124 -0.170 -0.722 -0.79  -0.91  -2.09
#  1.42  -0.409  -0.525  2.155 -0.841 -0.19   0.13   0.63
#  0.94   1.192  -1.075  0.017  0.167  0.54   0.52   1.42
# -0.53   0.777  -1.090 -2.237 -0.693  0.24  -0.56   1.45
#  0.34  -0.456  -1.220 -0.456 -0.315  1.10   1.38  -0.05
#
# [100 rows x 8 columns]
```

H2O also supports merging two frames together by matching column names:

```
df10 = h2o.H2OFrame.from_python( {
        'A': ['Hello', 'World', 'Welcome', 'To', 'H2O', 'World'],
        'n': [0,1,2,3,4,5]} )

# Create a single-column, 100-row frame
# Include random integers from 0-5
df11 = h2o.H2OFrame.from_python(np.random.randint(0,6,(100,1)), column_names=
    list('n'))

# Combine column "n" from both datasets
df11.merge(df10)

#    n  A
#  ---  -------
#    2  Welcome
#    5  World
#    4  H2O
#    2  Welcome
#    3  To
#    3  To
#    1  World
#    1  World
#    3  To
#    1  World
#
# [100 rows x 2 columns]
```

4.6 Grouping

"Grouping" refers to the following process:

- splitting the data into groups based on some criteria
- applying a function to each group independently
- combining the results into an H2OFrame

To group and then apply a function to the results:

```
df12 = h2o.H2OFrame(
    {'A' : ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
    'B' : ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
    'C' : np.random.randn(8).tolist(),
    'D' : np.random.randn(8).tolist()})

# View the H2OFrame
df12

#  A             C  B                D
#  ---  ----------  -----  -----------
#  foo  -0.710095   one     0.253189
#  bar  -0.165891   one    -0.433233
#  foo  -1.51996    two     1.12321
#  bar   2.25083    three   0.512449
#  foo  -0.618324   two     1.35158
#  bar   0.0817828  two     0.00830419
```

```
#  foo   0.634827   one     1.25897
#  foo   0.879319   three   1.48051
#
# [8 rows x 4 columns]

df12.group_by('A').sum().frame

#  A       sum_C    sum_B      sum_D
#  ---  --------  -------  ---------
#  bar   2.16672        3  0.0875206
#  foo  -1.33424        5  5.46746
#
# [2 rows x 4 columns]
```

To group by multiple columns and then apply a function:

```
df13 = df12.group_by(['A','B']).sum().frame

# View the H2OFrame
df13

#  A    B           sum_C        sum_D
#  ---  -----  ----------  -----------
#  bar  one    -0.165891   -0.433233
#  bar  three   2.25083     0.512449
#  bar  two     0.0817828   0.00830419
#  foo  one    -0.0752683   1.51216
#  foo  three   0.879319    1.48051
#  foo  two    -2.13829     2.47479
#
# [6 rows x 4 columns]
```

Use `merge` to join the results into the original H2OFrame:

```
df12.merge(df13)

#  A    B                C             D        sum_C        sum_D
#  ---  -----  ----------  -----------  ----------  -----------
#  foo  one    -0.710095    0.253189    -0.0752683   1.51216
#  bar  one    -0.165891   -0.433233    -0.165891   -0.433233
#  foo  two    -1.51996     1.12321     -2.13829     2.47479
#  bar  three   2.25083     0.512449     2.25083     0.512449
#  foo  two    -0.618324    1.35158     -2.13829     2.47479
#  bar  two     0.0817828   0.00830419   0.0817828   0.00830419
#  foo  one     0.634827    1.25897     -0.0752683   1.51216
#  foo  three   0.879319    1.48051      0.879319    1.48051
#
# [8 rows by 6 columns]
```

4.7 Using Date and Time Data

H2O has powerful features for ingesting and feature engineering using time data. Internally, H2O stores time information as an integer of the number of milliseconds since the epoch.

To ingest time data natively, use one of the supported time input formats:

```
df14 = h2o.H2OFrame.from_python(
       {'D': ['18OCT2015:11:00:00',
       '19OCT2015:12:00:00',
       '20OCT2015:13:00:00']},
       column_types=['time'])

df14.types
# {u'D': u'time'}
```

To display the day of the month:

```
df14['D'].day()

#   D
# ---
#  18
#  19
#  20
```

To display the day of the week:

```
df14['D'].dayOfWeek()

# D
# ---
# Sun
# Mon
# Tue
```

4.8 Categoricals

H2O handles categorical (also known as enumerated or factor) values in an H2OFrame. This is significant because categorical columns have specific treatments in each of the machine learning algorithms.

Using 'df12' from above, H2O imports columns A and B as categorical/enumerated/factor types:

```
df12.types
# {u'A': u'enum', u'C': u'real', u'B': u'enum', u'D': u'real'}
```

To determine if any column is a categorical/enumerated/factor type:

```
df12.anyfactor()
# True
```

To view the categorical levels in a single column:

```
df12["A"].levels()
# ['bar', 'foo']
```

To create categorical interaction features:

```
df12.interaction(['A','B'], pairwise=False, max_factors=3, min_occurrence=1)

#  A_B
#  -------
#  foo_one
#  bar_one
#  foo_two
#  other
#  foo_two
#  other
#  foo_one
#  other
#
# [8 rows x 1 column]
```

To retain the most common categories and set the remaining categories to a common 'Other' category and create an interaction of a categorical column with itself:

```
bb_df = df12.interaction(['B','B'], pairwise=False, max_factors=2,
     min_occurrence=1)

# View H2OFrame
bb_df

#  B_B
#  -----
#  one
#  one
#  two
#  other
#  two
#  two
#  one
#  other
#
# [8 rows x 1 column]
```

These can then be added as a new column on the original dataframe:

```
df15 = df12.cbind(bb_df)

# View H2OFrame
df15

# A             C  B               D  B_B
# ---  ----------  -----  ----------  -----
# foo  -0.809171   one     1.79059    one
# bar   0.216644   one     2.88524    one
# foo  -0.033664   two     0.61205    two
# bar   0.985545   three   0.357742   other
# foo  -2.15563    two     0.0456449  two
# bar  -0.0170454  two    -1.33625    two
# foo   1.32524    one     0.308092   one
# foo  -0.546305   three  -0.92675    other
#
# [8 rows x 5 columns]
```

4.9 Loading and Saving Data

In addition to loading data from Python objects, H2O can load data directly from:

- disk
- network file systems (NFS, S3)
- distributed file systems (HDFS)
- HTTP addresses

H2O currently supports the following file types:

- CSV (delimited) files
- ORC
- SVMLite
- ARFF
- XLS
- XLST

To load data from the same machine running H2O:

```
df = h2o.upload_file("/pathToFile/fileName")
```

To load data from the machine(s) running H2O to the machine running Python:

```
df = h2o.import_file("/pathToFile/fileName")
```

To save an H2OFrame on the machine running H2O:

```
h2o.export_file(df,"/pathToFile/fileName")
```

To save an H2OFrame on the machine running Python:

```
h2o.download_csv(df,"/pathToFile/fileName")
```

5 Machine Learning

The following sections describe some common model types and features.

5.1 Modeling

The following section describes the features and functions of some common models available in H2O. For more information about running these models in

Python using H2O, refer to the documentation on the H2O.ai website or to the booklets on specific models.

H2O supports the following models:

- Deep Learning
- Naïve Bayes
- Principal Components Analysis (PCA)
- K-means
- Stacked Ensembles
- Generalized Linear Models (GLM)
- Gradient Boosting Machine (GBM)
- Generalized Low Rank Model (GLRM)
- Distributed Random Forest (DRF)
- Word2vec

The list continues to grow, so check `www.h2o.ai` to see the latest additions.

5.1.1 Supervised Learning

Generalized Linear Models (GLM): Provides flexible generalization of ordinary linear regression for response variables with error distribution models other than a Gaussian (normal) distribution. GLM unifies various other statistical models, including Poisson, linear, logistic, and others when using ℓ_1 and ℓ_2 regularization.

Distributed Random Forest: Averages multiple decision trees, each created on different random samples of rows and columns. It is easy to use, non-linear, and provides feedback on the importance of each predictor in the model, making it one of the most robust algorithms for noisy data.

Gradient Boosting Machine (GBM): Produces a prediction model in the form of an ensemble of weak prediction models. It builds the model in a stage-wise fashion and is generalized by allowing an arbitrary differentiable loss function. It is one of the most powerful methods available today.

Deep Learning: Models high-level abstractions in data by using non-linear transformations in a layer-by-layer method. Deep learning is an example of supervised learning, which can use unlabeled data that other algorithms cannot.

Naïve Bayes: Generates a probabilistic classifier that assumes the value of a particular feature is unrelated to the presence or absence of any other feature, given the class variable. It is often used in text categorization.

Stacked Ensembles: Using multiple models built from different algorithms, Stacked Ensembles finds the optimal combination of a collection of prediction algorithms using a process known as "stacking."

5.1.2 Unsupervised Learning

K-Means: Reveals groups or clusters of data points for segmentation. It clusters observations into k-number of points with the nearest mean.

Principal Component Analysis (PCA): The algorithm is carried out on a set of possibly collinear features and performs a transformation to produce a new set of uncorrelated features.

Generalized Low Rank Model (GLRM): The method reconstructs missing values and identifies important features in heterogeneous data. It also recognizes a number of interpretations of low rank factors, which allows clustering of examples or of features.

Anomaly Detection: Identifies the outliers in your data by invoking the deep learning autoencoder, a powerful pattern recognition model.

5.1.3 Miscellaneous

Word2vec: Takes a text corpus as an input and produces the word vectors as output. The result is an H2O Word2vec model that can be exported as a binary model or as a MOJO.

5.2 Running Models

This section describes how to run the following model types:

- Gradient Boosting Machine (GBM)
- Generalized Linear Models (GLM)
- K-means
- Principal Components Analysis (PCA)

This section also shows how to generate predictions.

5.2.1 Gradient Boosting Machine (GBM)

To generate gradient boosting machine models for creating forward-learning ensembles, use `H2OGradientBoostingEstimator`.

The construction of the estimator defines the parameters of the estimator and the call to `H2OGradientBoostingEstimator.train` trains the estimator on the specified data. This pattern is common for each of the H2O algorithms.

```
In [1]: import h2o

In [2]: h2o.init()

Checking whether there is an H2O instance running at http://localhost
    :54321..... not found.
Attempting to start a local H2O server...
  Java Version: java version "1.8.0_25"; Java(TM) SE Runtime Environment (
      build 1.8.0_25-b17); Java HotSpot(TM) 64-Bit Server VM (build 25.25-
      b02, mixed mode)
  Starting server from /usr/local/h2o_jar/h2o.jar
  Ice root: /var/folders/yl/cq5nhky53hjcl9wrqxt39kz80000gn/T/tmpHpRzVe
  JVM stdout: /var/folders/yl/cq5nhky53hjcl9wrqxt39kz80000gn/T/tmpHpRzVe/
      h2o_techwriter_started_from_python.out
  JVM stderr: /var/folders/yl/cq5nhky53hjcl9wrqxt39kz80000gn/T/tmpHpRzVe/
      h2o_techwriter_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.

In [3]: from h2o.estimators.gbm import H2OGradientBoostingEstimator

In [4]: iris_data_path = "http://h2o-public-test-data.s3.amazonaws.com/
    smalldata/iris/iris.csv" # load demonstration data

In [5]: iris_df = h2o.import_file(path=iris_data_path)

Parse Progress: [##################################] 100%

In [6]: iris_df.describe()
Rows:150 Cols:5

Chunk compression summary:
chunktype  chunkname  count  count_%  size  size_%
---------  ---------  -----  -------  ----  ------
1-Byte Int   C1         1       20    218B  18.890
1-Byte Flt   C2         4       80    936B  81.109

Frame distribution summary:
                size  rows  chunks/col  chunks
--------------  ----  ----  ----------  ------
127.0.0.1:54321  1.1KB  150         1       5
mean             1.1KB  150         1       5
min              1.1KB  150         1       5
max              1.1KB  150         1       5
stddev           0  B    0         0       0
total            1.1 KB 150         1       5

         C1              C2              C3             C4              C5
-------  --------------  --------------  -------------  --------------
     -----------
type     real            real            real           real            enum
```

```
mins     4.3             2.0             1.0             0.1             0.0
mean     5.84333333333   3.054           3.75866666667   1.19866666667   NaN
maxs     7.9             4.4             6.9             2.5             2.0
sigma    0.828066127978  0.433594311362  1.76442041995   0.763160741701  NaN
zeros    0               0               0               0               50
missing  0               0               0               0               0


In [7]: gbm_regressor = H2OGradientBoostingEstimator(distribution="gaussian",
      ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")

In [8]: gbm_regressor.train(x=range(1,iris_df.ncol), y=0, training_frame=
     iris_df)

gbm Model Build Progress: [####################] 100%

In [9]: gbm_regressor
Out[9]: Model Details
=============
H2OGradientBoostingEstimator: Gradient Boosting Machine
Model Key:  GBM_model_python_1446220160417_2

Model Summary:
    number_of_trees           |         10
    model_size_in_bytes       |         1535
    min_depth                 |         3
    max_depth                 |         3
    mean_depth                |         3
    min_leaves                |         7
    max_leaves                |         8
    mean_leaves               |         7.8

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

MSE: 0.0706936802293
RMSE: 0.265882831769
MAE: 0.219981056849
RMSLE: 0.0391855537448
Mean Residual Deviance: 0.0706936802293

Scoring History:
    timestamp            duration    number_of_trees    training_MSE
         training_deviance
--  -------------------  ----------  -----------------  --------------
     -------------------
    2015-10-30 08:50:00  0.121 sec   1                  0.472445
         0.472445
    2015-10-30 08:50:00  0.151 sec   2                  0.334868
         0.334868
    2015-10-30 08:50:00  0.162 sec   3                  0.242847
         0.242847
    2015-10-30 08:50:00  0.175 sec   4                  0.184128
         0.184128
    2015-10-30 08:50:00  0.187 sec   5                  0.14365
         0.14365
    2015-10-30 08:50:00  0.197 sec   6                  0.116814
         0.116814
    2015-10-30 08:50:00  0.208 sec   7                  0.0992098
         0.0992098
    2015-10-30 08:50:00  0.219 sec   8                  0.0864125
         0.0864125
```

```
    2015-10-30 08:50:00  0.229 sec   9                   0.077629
         0.077629
    2015-10-30 08:50:00  0.238 sec   10                  0.0706937
         0.0706937

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
C3          227.562                1                    0.894699
C2          15.1912                0.0667563            0.0597268
C5          9.50362                0.0417627            0.037365
C4          2.08799                0.00917544           0.00820926
```

To generate a classification model that uses labels, use `distribution="multinomial"`:

```
In [10]: gbm_classifier = H2OGradientBoostingEstimator(distribution="
     multinomial", ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")

In [11]: gbm_classifier.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1,
     training_frame=iris_df)

gbm Model Build Progress: [##################################] 100%

In [12]: gbm_classifier
Out[12]: Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1446220160417_4

Model Summary:
    number_of_trees    model_size_in_bytes    min_depth    max_depth
         mean_depth    min_leaves    max_leaves    mean_leaves
--  -----------------  ---------------------  -----------  -----------
     ------------  ------------  ------------  -------------
    30                 3933                   1            3
         2.93333       2             8             5.86667


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

MSE: 0.00976685303214
RMSE: 0.0988273900907
LogLoss: 0.0782480973696
Mean Per-Class Error: 0.00666666666667
Confusion Matrix: vertical: actual; across: predicted

Iris-setosa    Iris-versicolor    Iris-virginica    Error       Rate
-------------  -----------------  ----------------  ----------  -------
50             0                  0                 0           0 / 50
0              49                 1                 0.02        1 / 50
0              0                  50                0           0 / 50
50             49                 51                0.00666667  1 / 150

Top-3 Hit Ratios:
k    hit_ratio
---  -----------
1    0.993333
2    1
3    1
```

```
41
42 Scoring History:
43     timestamp            duration    number_of_trees    training_MSE
           training_logloss    training_classification_error
44 --  -------------------  ----------  -----------------  --------------
        ------------------  ------------------------------
45     2015-10-30 08:51:52  0.047 sec   1                  0.282326
           0.758411            0.0266667
46     2015-10-30 08:51:52  0.068 sec   2                  0.179214
           0.550506            0.0266667
47     2015-10-30 08:51:52  0.086 sec   3                  0.114954
           0.412173            0.0266667
48     2015-10-30 08:51:52  0.100 sec   4                  0.0744726
           0.313539            0.02
49     2015-10-30 08:51:52  0.112 sec   5                  0.0498319
           0.243514            0.02
50     2015-10-30 08:51:52  0.131 sec   6                  0.0340885
           0.19091             0.00666667
51     2015-10-30 08:51:52  0.143 sec   7                  0.0241071
           0.151394            0.00666667
52     2015-10-30 08:51:52  0.153 sec   8                  0.017606
           0.120882            0.00666667
53     2015-10-30 08:51:52  0.165 sec   9                  0.0131024
           0.0975897           0.00666667
54     2015-10-30 08:51:52  0.180 sec   10                 0.00976685
           0.0782481           0.00666667
55
56 Variable Importances:
57 variable    relative_importance    scaled_importance    percentage
58 ----------  ---------------------  -------------------  ------------
59 C4          192.761                1                    0.774374
60 C3          54.0381                0.280338             0.217086
61 C1          1.35271                0.00701757           0.00543422
62 C2          0.773032               0.00401032           0.00310549
```

5.2.2 Generalized Linear Models (GLM)

Generalized linear models (GLM) are some of the most commonly-used models for many types of data analysis use cases. While some data can be analyzed using linear models, linear models may not be as accurate if the variables are more complex. For example, if the dependent variable has a non-continuous distribution or if the effect of the predictors is not linear, generalized linear models will produce more accurate results than linear models.

Generalized Linear Models (GLM) estimate regression models for outcomes following exponential distributions in general. In addition to the Gaussian (i.e. normal) distribution, these include Poisson, binomial, gamma and Tweedie distributions. Each serves a different purpose and, depending on distribution and link function choice, it can be used either for prediction or classification.

H2O's GLM algorithm fits the generalized linear model with elastic net penalties. The model fitting computation is distributed, extremely fast,and scales extremely

well for models with a limited number ($\sim$ low thousands) of predictors with non-zero coefficients.

The algorithm can compute models for a single value of a penalty argument or the full regularization path, similar to glmnet. It can compute Gaussian (linear), logistic, Poisson, and gamma regression models. To generate a generalized linear model for developing linear models for exponential distributions, use `H2OGeneralizedLinearEstimator`. You can apply regularization to the model by adjusting the lambda and alpha parameters.

```
In [13]: from h2o.estimators.glm import H2OGeneralizedLinearEstimator

In [14]: prostate_data_path = "http://h2o-public-test-data.s3.amazonaws.com/
     smalldata/prostate/prostate.csv"

In [15]: prostate_df = h2o.import_file(path=prostate_data_path)

Parse Progress: [##################################] 100%

In [16]: prostate_df["RACE"] = prostate_df["RACE"].asfactor()

In [17]: prostate_df.describe()
Rows:380 Cols:9

Chunk compression summary:
chunk_type    chunk_name                 count    count_percentage    size
       size_percentage
------------  -------------------------  -------  ------------------  ------
     -----------------
CBS           Bits                       1        11.1111             118  B
     1.39381
C1N           1-Byte Integers (w/o NAs)  5        55.5556             2.2 KB
     26.4588
C2            2-Byte Integers            1        11.1111             828  B
     9.7803
CUD           Unique Reals               1        11.1111             2.1 KB
     25.6556
C8D           64-bit Reals               1        11.1111             3.0 KB
     36.7116

Frame distribution summary:
                 size    number_of_rows    number_of_chunks_per_column
                      number_of_chunks
---------------  ------  ----------------  ------------------------------
     ------------------
127.0.0.1:54321  8.3 KB  380               1                                9
mean             8.3 KB  380               1                                9
min              8.3 KB  380               1                                9
max              8.3 KB  380               1                                9
stddev           0  B    0                 0                                0
total            8.3 KB  380               1                                9



In [18]: glm_classifier = H2OGeneralizedLinearEstimator(family="binomial",
     nfolds=10, alpha=0.5)

In [19]: glm_classifier.train(x=["AGE","RACE","PSA","DCAPS"],y="CAPSULE",
     training_frame=prostate_df)
```

```
glm Model Build Progress: [##################################] 100%

In [20]: glm_classifier
Out[20]: Model Details
=============
H2OGeneralizedLinearEstimator :  Generalized Linear Model
Model Key:  GLM_model_python_1446220160417_6

GLM Model: summary

    family    link    regularization
         number_of_predictors_total    number_of_active_predictors
         number_of_iterations    training_frame
--  --------  ------  ------------------------------------------
     --------------------------  ----------------------------
     ----------------------  ----------------
    binomial  logit   Elastic Net (alpha = 0.5, lambda = 3.251E-4 )  6
                                    6                                 6
                               py_3


ModelMetricsBinomialGLM: glm
** Reported on train data. **

MSE: 0.202442565125
RMSE: 0.449936178947
LogLoss: 0.591121990582
Null degrees of freedom: 379
Residual degrees of freedom: 374
Null deviance: 512.288840185
Residual deviance: 449.252712842
AIC: 461.252712842
AUC: 0.718954248366
Gini: 0.437908496732
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.282384349078:
       0    1    Error    Rate
-----  ---  ---  -------  -------------
0      80   147  0.6476   (147.0/227.0)
1      19   134  0.1242   (19.0/153.0)
Total  99   281  0.4368   (166.0/380.0)

Maximum Metrics: Maximum metrics at their respective thresholds

metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                        0.282384     0.617849  276
max f2                        0.198777     0.77823   360
max f0point5                  0.415125     0.636672  108
max accuracy                  0.415125     0.705263  108
max precision                 0.998613     1         0
max recall                    0.198777     1         360
max specificity               0.998613     1         0
max absolute_mcc              0.415125     0.369123  108
max min_per_class_accuracy    0.332648     0.656388  175
max mean_per_class_accuracy   0.377454     0.67326   123
Gains/Lift Table: Avg response rate: 40.26 %


ModelMetricsBinomialGLM: glm
** Reported on cross-validation data. **
```

```
MSE: 0.209698776592
RMSE: 0.457928789871
LogLoss: 0.610086165597
Null degrees of freedom: 379
Residual degrees of freedom: 374
Null deviance: 513.330704712
Residual deviance: 463.665485854
AIC: 475.665485854
AUC: 0.688203622124
Gini: 0.376407244249
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.339885371023:
       0    1    Error    Rate
-----  ---  ---  -------  -------------
0      154  73   0.3216   (73.0/227.0)
1      53   100  0.3464   (53.0/153.0)
Total  207  173  0.3316   (126.0/380.0)
Maximum Metrics: Maximum metrics at their respective thresholds

metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                        0.339885     0.613497  172
max f2                        0.172551     0.773509  376
max f0point5                  0.419649     0.615251  105
max accuracy                  0.447491     0.692105  93
max precision                 0.998767     1         0
max recall                    0.172551     1         376
max specificity               0.998767     1         0
max absolute_mcc              0.419649     0.338849  105
max min_per_class_accuracy    0.339885     0.653595  172
max mean_per_class_accuracy   0.339885     0.666004  172
Gains/Lift Table: Avg response rate: 40.26 %


Scoring History:
    timestamp            duration    iteration    log_likelihood    objective
--  -------------------  ----------  -----------  ----------------
     -----------
    2016-08-25 12:54:20  0.000 sec   0            256.144
         0.674064
    2016-08-25 12:54:20  0.055 sec   1            226.961
         0.597573
    2016-08-25 12:54:20  0.092 sec   2            224.728
         0.591813
    2016-08-25 12:54:20  0.125 sec   3            224.627
         0.591578
    2016-08-25 12:54:20  0.157 sec   4            224.626
         0.591578
```

5.2.3 K-means

To generate a K-means model for data characterization, use `h2o.kmeans()`. This algorithm does not require a dependent variable.

```
In [21]: from h2o.estimators.kmeans import H2OKMeansEstimator

In [22]: cluster_estimator = H2OKMeansEstimator(k=3)

In [23]: cluster_estimator.train(x=[0,1,2,3], training_frame=iris_df)
```

```
kmeans Model Build Progress: [##################################] 100%

In [24]: cluster_estimator
Out[24]: Model Details
=============
H2OKMeansEstimator :  K-means
Model Key:  K-means_model_python_1446220160417_8

Model Summary:
    number_of_rows    number_of_clusters    number_of_categorical_columns
         number_of_iterations    within_cluster_sum_of_squares
         total_sum_of_squares    between_cluster_sum_of_squares
--  ---------------  --------------------  ------------------------------
     ----------------------  ------------------------------
     ----------------------  ------------------------------
    150               3                     0
         4                       190.757                           596
                             405.243

ModelMetricsClustering: kmeans
** Reported on train data. **

MSE: NaN
RMSE: NaN
Total Within Cluster Sum of Square Error: 190.756926265
Total Sum of Square Error to Grand Mean: 596.0
Between Cluster Sum of Square Error: 405.243073735

Centroid Statistics:
    centroid    size    within_cluster_sum_of_squares
--  ----------  ------  ------------------------------
    1           96      149.733
    2           32      17.292
    3           22      23.7318

Scoring History:
    timestamp            duration    iteration    avg_change_of_std_centroids
             within_cluster_sum_of_squares
--  -------------------  ----------  -----------
     ----------------------------  ------------------------------
    2016-08-25 13:03:36  0.005 sec   0            nan
                                  385.505
    2016-08-25 13:03:36  0.029 sec   1            1.37093
                                   173.769
    2016-08-25 13:03:36  0.029 sec   2            0.184617
                                  141.623
    2016-08-25 13:03:36  0.030 sec   3            0.00705735
                                  140.355
    2016-08-25 13:03:36  0.030 sec   4            0.00122272
                                  140.162
    2016-08-25 13:03:36  0.031 sec   5            0.000263918
                                  140.072
    2016-08-25 13:03:36  0.031 sec   6            0.000306555
                                  140.026
```

5.2.4 Principal Components Analysis (PCA)

To map a set of variables onto a subspace using linear transformations, use `h2o.transforms.decomposition.H2OPCA`. This is the first step in Principal Components Regression.

```
In [25]: from h2o.transforms.decomposition import H2OPCA

In [26]: pca_decomp = H2OPCA(k=2, transform="NONE", pca_method="Power")

In [27]: pca_decomp.train(x=range(0,4), training_frame=iris_df)

pca Model Build Progress: [######################################] 100%

In [28]: pca_decomp
Out[28]: Model Details
=============
H2OPCA :  Principal Component Analysis
Model Key:  PCA_model_python_1446220160417_10

Importance of components:
                        pc1      pc2
----------------------  -------  --------
Standard deviation      7.86058  1.45192
Proportion of Variance  0.96543  0.032938
Cumulative Proportion   0.96543  0.998368


ModelMetricsPCA: pca
** Reported on train data. **

MSE: NaN
RMSE: NaN

In [29]: pred = pca_decomp.predict(iris_df)

pca prediction progress: [######################################] 100%

In [30]: pred.head()  # Projection results
Out[30]:
    PC1      PC2
-------  -------
5.9122   2.30344
5.57208  1.97383
5.44648  2.09653
5.43602  1.87168
5.87507  2.32935
6.47699  2.32553
5.51543  2.07156
5.85042  2.14948
5.15851  1.77643
5.64458  1.99191
```

5.3 Grid Search

H2O supports grid search across hyperparameters:

```
In [32]: ntrees_opt = [5, 10, 15]

In [33]: max_depth_opt = [2, 3, 4]

In [34]: learn_rate_opt = [0.1, 0.2]

In [35]: hyper_parameters = {"ntrees": ntrees_opt, "max_depth":max_depth_opt,
      "learn_rate":learn_rate_opt}

In [36]: from h2o.grid.grid_search import H2OGridSearch

In [37]: gs = H2OGridSearch(H2OGradientBoostingEstimator(distribution="
     multinomial"), hyper_params=hyper_parameters)

In [38]: gs.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1, training_frame
     =iris_df, nfolds=10)

gbm Grid Build Progress: [#####################################] 100%

In [39]: print gs.sort_by('logloss', increasing=True)

Grid Search Results:
Model Id                        Hyperparameters: ['learn_rate', 'ntrees', '
     max_depth']    logloss
-------------------------------
     ------------------------------------------------------  ---------
Grid_GBM_model_1446220160417_30  ['0.2, 15, 4']
                                                    0.05105
Grid_GBM_model_1446220160417_27  ['0.2, 15, 3']
                                                    0.0551088
Grid_GBM_model_1446220160417_24  ['0.2, 15, 2']
                                                    0.0697714
Grid_GBM_model_1446220160417_29  ['0.2, 10, 4']
                                                    0.103064
Grid_GBM_model_1446220160417_26  ['0.2, 10, 3']
                                                    0.106232
Grid_GBM_model_1446220160417_23  ['0.2, 10, 2']
                                                    0.120161
Grid_GBM_model_1446220160417_21  ['0.1, 15, 4']
                                                    0.170086
Grid_GBM_model_1446220160417_18  ['0.1, 15, 3']
                                                    0.171218
Grid_GBM_model_1446220160417_15  ['0.1, 15, 2']
                                                    0.181186
Grid_GBM_model_1446220160417_28  ['0.2, 5, 4']
                                                     0.275788
Grid_GBM_model_1446220160417_25  ['0.2, 5, 3']
                                                     0.27708
Grid_GBM_model_1446220160417_22  ['0.2, 5, 2']
                                                     0.280413
Grid_GBM_model_1446220160417_20  ['0.1, 10, 4']
                                                    0.28759
Grid_GBM_model_1446220160417_17  ['0.1, 10, 3']
                                                    0.288293
Grid_GBM_model_1446220160417_14  ['0.1, 10, 2']
                                                    0.292993
Grid_GBM_model_1446220160417_16  ['0.1, 5, 3']
                                                     0.520591
```

```
Grid_GBM_model_1446220160417_19  ['0.1, 5, 4']
                                                    0.520697
Grid_GBM_model_1446220160417_13  ['0.1, 5, 2']
                                                    0.524777
```

5.4 Integration with scikit-learn

The H2O Python client can be used within scikit-learn pipelines and cross-validation searches. This extends the capabilities of both H2O and scikit-learn. Note that the sklearn and scipy packages are required to use the H2O Python client with scikit-learn.

5.4.1 Pipelines

To create a scikit-learn style pipeline using H2O transformers and estimators:

```
In [41]: from h2o.transforms.preprocessing import H2OScaler

In [42]: from sklearn.pipeline import Pipeline

In [43]: # Turn off h2o progress bars

In [44]: h2o.__PROGRESS_BAR__=False

In [45]: h2o.no_progress()

In [46]: # build transformation pipeline using sklearn's Pipeline and H2O
     transforms

In [47]: pipeline = Pipeline([("standardize", H2OScaler()),
   ....:                       ("pca", H2OPCA(k=2)),
   ....:                       ("gbm", H2OGradientBoostingEstimator(distribution="
        multinomial"))])

In [48]: pipeline.fit(iris_df[:4],iris_df[4])
Out[48]: Model Details
=============
H2OPCA :  Principal Component Analysis
Model Key:  PCA_model_python_1446220160417_32

Importance of components:
                          pc1       pc2
----------------------  --------  ---------
Standard deviation      3.22082   0.34891
Proportion of Variance  0.984534  0.0115538
Cumulative Proportion   0.984534  0.996088


ModelMetricsPCA: pca
** Reported on train data. **

MSE: NaN
RMSE: NaN
Model Details
```

```
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1446220160417_34

Model Summary:
    number_of_trees    number_of_internal_trees  model_size_in_bytes
         min_depth    max_depth    mean_depth    min_leaves    max_leaves
         mean_leaves
--  -----------------  -------------------------  ----------------------
     -----------  -----------  ------------  ------------  -------------
     -------------
    50                 150                           28170                   1
                    5            4.84          2             13
         9.97333


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

MSE: 0.00162796447355
RMSE: 0.0403480417561
LogLoss: 0.0152718656454
Mean Per-Class Error: 0.0
Confusion Matrix: vertical: actual; across: predicted

Iris-setosa    Iris-versicolor    Iris-virginica    Error    Rate
-------------  -----------------  ----------------  -------  -------
50             0                  0                 0        0 / 50
0              50                 0                 0        0 / 50
0              0                  50                0        0 / 50
50             50                 50                0        0 / 150

Top-3 Hit Ratios:
k    hit_ratio
---  -----------
1    1
2    1
3    1

Scoring History:
     timestamp            duration    number_of_trees    training_rmse
          training_logloss    training_classification_error
---  -------------------  ----------  -----------------  ----------------
     ------------------  ------------------------------
     2016-08-25 13:50:21  0.006 sec   0.0                0.666666666667
          1.09861228867       0.66
     2016-08-25 13:50:21  0.077 sec   1.0                0.603019288754
          0.924249463924      0.04
     2016-08-25 13:50:21  0.096 sec   2.0                0.545137025745
          0.788619346614      0.04
     2016-08-25 13:50:21  0.110 sec   3.0                0.492902188607
          0.679995476522      0.04
     2016-08-25 13:50:21  0.123 sec   4.0                0.446151758168
          0.591313596193      0.04
---  ---                  ---         ---                ---               ---
                        ---
     2016-08-25 13:50:21  0.419 sec   46.0               0.0489303232171
          0.0192767805328     0.0
     2016-08-25 13:50:21  0.424 sec   47.0               0.0462779490149
          0.0180720396825     0.0
     2016-08-25 13:50:21  0.429 sec   48.0               0.0444689238255
          0.0171428314531     0.0
```

```
     2016-08-25 13:50:21  0.434 sec   49.0                0.0423442541538
          0.0161938230172     0.0
     2016-08-25 13:50:21  0.438 sec   50.0                0.0403480417561
          0.0152718656454     0.0

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
PC1         448.958                1                    0.982184
PC2         8.1438                 0.0181393            0.0178162
Pipeline(steps=[('standardize', <h2o.transforms.preprocessing.H2OScaler
     object at 0x1088c6a50>), ('pca', ), ('gbm', )])
```

5.4.2 Randomized Grid Search

To create a scikit-learn style hyperparameter grid search using k-fold cross validation:

```
In [57]: from sklearn.grid_search import RandomizedSearchCV

In [58]: from h2o.cross_validation import H2OKFold

In [59]: from h2o.model.regression import h2o_r2_score

In [60]: from sklearn.metrics.scorer import make_scorer

In [61]: from sklearn.metrics.scorer import make_scorer

# Parameters to test
In [62]: params = {"standardize__center":    [True, False],
   ....:           "standardize__scale":     [True, False],
   ....:           "pca__k":                 [2,3],
   ....:           "gbm__ntrees":            [10,20],
   ....:           "gbm__max_depth":         [1,2,3],
   ....:           "gbm__learn_rate":        [0.1,0.2]}

In [63]: custom_cv = H2OKFold(iris_df, n_folds=5, seed=42)

In [64]: pipeline = Pipeline([("standardize", H2OScaler()),
   ....:                      ("pca", H2OPCA(k=2)),
   ....:                      ("gbm", H2OGradientBoostingEstimator(
        distribution="gaussian"))])

In [65]: random_search = RandomizedSearchCV(pipeline, params,
   ....:                                    n_iter=5,
   ....:                                    scoring=make_scorer(h2o_r2_score),
   ....:                                    cv=custom_cv,
   ....:                                    random_state=42,
   ....:                                    n_jobs=1)
In [66]: random_search.fit(iris_df[1:], iris_df[0])
Out[66]:
RandomizedSearchCV(cv=<h2o.cross_validation.H2OKFold instance at 0x10ba413d0
     >,
          error_score='raise',
          estimator=Pipeline(steps=[('standardize', <h2o.transforms.
               preprocessing.H2OScaler object at 0x10c0f18d0>), ('pca', ), ('
               gbm', )]),
          fit_params={}, iid=True, n_iter=5, n_jobs=1,
```

```
          param_distributions={'pca__k': [2, 3], 'gbm__ntrees': [10, 20], '
              standardize__scale': [True, False], 'gbm__max_depth': [1, 2,
              3], 'standardize__center': [True, False], 'gbm__learn_rate':
              [0.1, 0.2]},
          pre_dispatch='2*n_jobs', random_state=42, refit=True,
          scoring=make_scorer(h2o_r2_score), verbose=0)

In [67]: print random_search.best_estimator_
Model Details
=============
H2OPCA :  Principal Component Analysis
Model Key:  PCA_model_python_1446220160417_136

Importance of components:
                        pc1       pc2         pc3
----------------------  --------  ----------  ----------
Standard deviation      9.6974  0.091905     0.031356
Proportion of Variance  0.9999  8.98098e-05  1.04541e-05
Cumulative Proportion   0.9999  0.99999      1


ModelMetricsPCA: pca
** Reported on train data. **

MSE: NaN
RMSE: NaN
Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1446220160417_138

Model Summary:
    number_of_trees    number_of_internal_trees   model_size_in_bytes
         min_depth    max_depth    mean_depth    min_leaves    max_leaves
         mean_leaves
--  -----------------  -------------------------  ---------------------
     -----------  -----------  ------------  ------------  ------------
     -------------
    20                 20                         2958                    3
                    3            3             5             8
         6.85

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

RMSE: 0.193906262445
MAE: 0.155086582663
RMSLE: NaN
Mean Residual Deviance: 0.0375996386155
Scoring History:

     timestamp            duration    number_of_trees    training_rmse
          training_mse    training_deviance
--   -------------------  ----------  -----------------  --------------
     --------------  -------------------
     2016-08-25 13:58:15  0.000 sec   0.0                0.683404046309
          0.569341466973  0.467041090512
     2016-08-25 13:58:15  0.002 sec   1.0                0.571086656306
          0.469106400643  0.326139969011
     2016-08-25 13:58:15  0.003 sec   2.0                0.483508601652
          0.395952082872  0.233780567872
```

```
     2016-08-25 13:58:15  0.004 sec   3.0                0.414549015095
          0.339981133963  0.171850885916
     2016-08-25 13:58:15  0.005 sec   4.0                0.362852508373
          0.298212416346  0.131661942833
---  ---                  ---         ---                ---              ---
                   ---
     2016-08-25 13:58:15  0.017 sec   16.0               0.204549491682
          0.164292158112  0.0418404945473
     2016-08-25 13:58:15  0.018 sec   17.0               0.201762323368
          0.162030458841  0.0407080351307
     2016-08-25 13:58:15  0.019 sec   18.0               0.199709571992
          0.160735480674  0.0398839131454
     2016-08-25 13:58:15  0.019 sec   19.0               0.196739590066
          0.158067452484  0.0387064662994
     2016-08-25 13:58:15  0.020 sec   20.0               0.193906262445
          0.155086582663  0.0375996386155

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
PC1         160.092                1                    0.894701
PC3         14.8175                0.0925562            0.08281
PC2         4.0241                 0.0251361            0.0224893
Pipeline(steps=[('standardize', <h2o.transforms.preprocessing.H2OScaler
     object at 0x10c1679d0>), ('pca', ), ('gbm', )])
```

6 Acknowledgments

We would like to acknowledge the following individuals for their contributions to this booklet: Spencer Aiello, Cliff Click, Hank Roark, Ludi Rehak, and Jessica Lanford.

7 References

H2O.ai Team. **H2O website**, 2017. URL `http://h2o.ai`

H2O.ai Team. **H2O documentation**, 2017. URL `http://docs.h2o.ai`

H2O.ai Team. **H2O Python Documentation**, 2015. URL `http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Pydoc.html`

H2O.ai Team. **H2O GitHub Repository**, 2017. URL `https://github.com/h2oai`

H2O.ai Team. **H2O Datasets**, 2017. URL `http://data.h2o.ai`

H2O.ai Team. **H2O JIRA**, 2017. URL `https://jira.h2o.ai`

H2O.ai Team. **H2Ostream**, 2017. URL `https://groups.google.com/d/forum/h2ostream`