# MACHINE LEARNING WITH PYTHON AND H2O


*Spencer Aiello, Cliff Click, Hank Roark & Ludi Rehak*

*Edited by: Jessica Lanford*


```
> pip install h2o
> import h2o
> h2o init()
> h2o.demo("glm")
```

# Machine Learning with Python and H2O


Spencer Aiello    Cliff Click

Hank Roark    Ludi Rehak

Edited by: Jessica Lanford


http://h2o.ai/resources/


February 2016: Third Edition

Machine Learning with Python and H2O
by Spencer Aiello, Cliff Click,
Hank Roark & Ludi Rehak
Edited by: Jessica Lanford

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



February 2016: Third Edition



Printed in the United States of America.

# Contents


**1 Introduction** **4**

**2 What is H2O?** **5**
2.1 Example Code . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Citation . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**3 Installation** **6**
3.1 Installation in Python . . . . . . . . . . . . . . . . . . . 6

**4 Data Preparation** **7**
4.1 Viewing Data . . . . . . . . . . . . . . . . . . . . . . . 10
4.2 Selection . . . . . . . . . . . . . . . . . . . . . . . . . 11
4.3 Missing Data . . . . . . . . . . . . . . . . . . . . . . . 13
4.4 Operations . . . . . . . . . . . . . . . . . . . . . . . . 14
4.5 Merging . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.6 Grouping . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.7 Using Date and Time Data . . . . . . . . . . . . . . . . 19
4.8 Categoricals . . . . . . . . . . . . . . . . . . . . . . . 19
4.9 Loading and Saving Data . . . . . . . . . . . . . . . . . 21

**5 Machine Learning** **22**
5.1 Modeling . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.1.1 Supervised Learning . . . . . . . . . . . . . . . . . 23
5.1.2 Unsupervised Learning . . . . . . . . . . . . . . . . 23
5.2 Running Models . . . . . . . . . . . . . . . . . . . . . 24
5.2.1 Gradient Boosting Models (GBM) . . . . . . . . . . 24
5.2.2 Generalized Linear Models (GLM) . . . . . . . . . . 27
5.2.3 K-means . . . . . . . . . . . . . . . . . . . . . . . 31
5.2.4 Principal Components Analysis (PCA) . . . . . . . . 32
5.3 Grid Search . . . . . . . . . . . . . . . . . . . . . . . . 33
5.4 Integration with scikit-learn . . . . . . . . . . . . . . . 34
5.4.1 Pipelines . . . . . . . . . . . . . . . . . . . . . . . 34
5.4.2 Randomized Grid Search . . . . . . . . . . . . . . . 36

**6 References** **39**

# 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 is fast, scalable, open-source machine learning and deep learning for smarter applications. With H2O, enterprises like PayPal, Nielsen Catalina, Cisco, and others can use all their data without sampling to get accurate predictions faster. Advanced algorithms such as deep learning, boosting, and bagging ensembles are built-in to help application designers create smarter applications through elegant APIs. Some of our initial customers have built powerful domain-specific predictive engines for recommendations, customer churn, propensity to buy, dynamic pricing, and fraud detection for the insurance, healthcare, telecommunications, ad tech, retail, and payment systems industries.

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 others. H2O also implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. 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, Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. With hundreds of meetups over the past three years, H2O has become a word-of-mouth phenomenon, growing amongst the data community by a hundred-fold, and is now used by 30,000+ users and is deployed using R, Python, Hadoop, and Spark in 2000+ corporations.

**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 meetups, training sessions, hackathons, and product updates, visit http://h2o.ai.
- Visit 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

## 2.2 Citation

To cite this booklet, use the following:

Aiello, S., Cliff, C., Roark, H., Rehak, L., and Lanford, J. (Feb 2016). *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.

(**Note**: The examples in this document were created with H2O version 3.7.0.99999.)

## 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:

```
In [1]: import h2o

In [2]: h2o.init()


No instance found at ip and port: localhost:54321. Trying to start local jar
    ...


JVM stdout: /var/folders/wg/3qx1qchx1jsfjqqbmz3stj7c0000gn/T/tmpof5ZIZ/
    h2o_hank_started_from_python.out
JVM stderr: /var/folders/wg/3qx1qchx1jsfjqqbmz3stj7c0000gn/T/tmpk4uayp/
    h2o_hank_started_from_python.err
Using ice_root: /var/folders/wg/3qx1qchx1jsfjqqbmz3stj7c0000gn/T/tmpKy1Wmt


Java Version: java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)


Starting H2O JVM and connecting: ............... Connection sucessful!
--------------------------  --------------------------
```

```
H2O cluster uptime:         1 seconds 591 milliseconds
H2O cluster version:        3.2.0.5
H2O cluster name:           H2O_started_from_python
H2O cluster total nodes:    1
H2O cluster total memory:   3.56 GB
H2O cluster total cores:    4
H2O cluster allowed cores:  4
H2O cluster healthy:        True
H2O Connection ip:          127.0.0.1
H2O Connection port:        54321
--------------------------  --------------------------
```

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

```
In[2]: h2o.init(ip="123.45.67.89", port=54321)
```

To create an H2OFrame object from a Python tuple:

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

Parse Progress: [###############################] 100%
Uploaded py9bccf8ce-c01e-40c8-bc73-b8e7e0b17c6a into cluster with 3 rows and
     3 cols

In [4]: df
Out[4]: H2OFrame with 3 rows and 3 columns:
  C1  C2      C3
----  ----  ----
   1  a      0.1
   2  b      0.2
   3  c      0.3
```

To create an H2OFrame object from a Python list:

```
In [5]: df = h2o.H2OFrame(zip(*[[1, 2, 3],
   ...:                    ['a', 'b', 'c'],
   ...:                    [0.1, 0.2, 0.3]]))

Parse Progress: [###############################] 100%
Uploaded py2c9ccb17-a86e-47d7-be1a-a7950b338870 into cluster with 3 rows and
     3 cols

In [6]: df
Out[6]: H2OFrame with 3 rows and 3 columns:
  C1  C2      C3
----  ----  ----
   1  a      0.1
   2  b      0.2
   3  c      0.3
```

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

```
In [7]: df = h2o.H2OFrame({'A': [1, 2, 3],
   ...:                    'B': ['a', 'b', 'c'],
   ...:                    'C': [0.1, 0.2, 0.3]})

Parse Progress: [###############################] 100%
Uploaded py2714e8a2-67c7-45a3-9d47-247120c5d931 into cluster with 3 rows and 3 cols

In [8]: df
Out[8]: H2OFrame with 3 rows and 3 columns:
  A    C  B
---  ---  ---
  1  0.1  a
  2  0.2  b
  3  0.3  c
```

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

```
In [14]: 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'])

Parse Progress: [###############################] 100%
Uploaded py17ea1f6d-ae83-451d-ad33-89e770061601 into cluster with 3 rows and 4 cols

In [10]: df2
Out[10]: H2OFrame with 3 rows and 4 columns:
  A      C  B                    D
--- ------ -- -------------------
  1  hello  a 2015-03-12 11:00:00
  2    all  a 2015-03-13 12:00:00
  3  world  b 2015-03-14 13:00:00
```

To display the column types:

```
In [11]: df2.types
Out[11]: {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:

```
In [16]: import numpy as np

In [17]: df = h2o.H2OFrame.from_python(np.random.randn(4,100).tolist(),
     column_names=list('ABCD'))

Parse Progress: [###############################] 100%
Uploaded py0a4d1d8d-7d04-438a-a97f-a9521f802366 into cluster with 100 rows
     and 4 cols

In [18]: df.head()
H2OFrame with 100 rows and 4 columns:
        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

In [19]: df.tail(5)
H2OFrame with 100 rows and 4 columns:
        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
```

To display the column names:

```
In [20]: df.columns
Out[20]: [u'A', u'B', u'C', u'D']
```

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

```
In [21]: 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

Column-by-Column Summary: (floats truncatede)

          A          B          C          D
-------   --------   --------   --------   --------
type      real       real       real       real
mins      -2.49822   -2.37446   -2.45977   -3.48247
maxs       2.59380    1.91998    3.13014    2.39057
mean      -0.01062   -0.23159    0.11423   -0.16228
sigma      1.04354    0.90576    0.96133    1.02608
zero_count    0          0          0          0
missing_count 0          0          0          0
```

## 4.2 Selection

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

```
In [23]: df['A']
Out[23]: H2OFrame with 100 rows and 1 columns:
          A
0 -0.613035
1 -1.265520
2  0.763851
3 -1.248425
4  2.105805
5  1.763502
6 -0.781973
7  1.400853
8 -0.746025
9 -1.120648
```

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

```
In [24]: df[1]
Out[24]: H2OFrame with 100 rows and 1 columns:
          B
0 -0.425327
1 -0.241526
2  0.039161
3  0.912686
4 -1.839950
5  0.573736
6  0.051883
7  1.919987
8 -0.632182
9  0.374212
```

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

```
In [25]: df[['B','C']]
Out[25]: H2OFrame with 100 rows and 2 columns:
          B         C
0 -0.425327 -1.927737
1 -0.241526 -0.044510
2  0.039161 -0.500049
3  0.912686 -0.611460
4 -1.839950  0.453875
5  0.573736 -0.309663
6  0.051883 -0.403075
7  1.919987  0.514212
8 -0.632182  1.274552
9  0.374212  0.232229
```

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

```
In [26]: df[0:2]
Out[26]: H2OFrame with 100 rows and 2 columns:
          A         B
0 -0.613035 -0.425327
1 -1.265520 -0.241526
2  0.763851  0.039161
3 -1.248425  0.912686
4  2.105805 -1.839950
5  1.763502  0.573736
6 -0.781973  0.051883
7  1.400853  1.919987
8 -0.746025 -0.632182
9 -1.120648  0.374212
```

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:

```
In [27]: df[2:7, :]
Out[27]: H2OFrame with 5 rows and 4 columns:
          A         B         C         D
0  0.763851  0.039161 -0.500049  0.355561
1 -1.248425  0.912686 -0.611460  1.946068
2  2.105805 -1.839950  0.453875 -1.699112
3  1.763502  0.573736 -0.309663 -1.511314
4 -0.781973  0.051883 -0.403075  0.569406
```

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

```
In [28]: df2[ df2["B"] == "a", :]
Out[28]: H2OFrame with 2 rows and 4 columns:
   A      C  B                   D
0  1  hello  a 2015-03-12 11:00:00
1  2    all  a 2015-03-13 12:00:00
```

## 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:

```
In [46]: 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'])

In [47]: df3
Out[47]: H2OFrame with 5 rows and 4 columns:
    A      C    B             D
0   1  hello    a  1.426183e+12
1   2    all    a           NaN
2   3  world    b  1.426273e+12
3 NaN    NaN  NaN           NaN
4 NaN    NaN  NaN  1.426363e+12
```

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

```
In [49]: df3["A"].isna()
Out[49]: H2OFrame with 5 rows and 1 columns:
   C1
0   0
1   0
2   0
3   1
4   1
```

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

```
In [52]: df3
Out[52]: H2OFrame with 5 rows and 4 columns:
   A      C    B             D
0  1  hello    a  1.426183e+12
1  2    all    a           NaN
2  3  world    b  1.426273e+12
3  5    NaN  NaN           NaN
4  5    NaN  NaN  1.426363e+12
```

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

```
In [53]: df3.isna()
Out[53]: H2OFrame with 5 rows and 4 columns:
   C1  C2  C3  C4
0   0   0   0   0
1   0   0   0   1
2   0   0   0   0
3   0   1   0   1
4   0   1   0   0
```

## 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:

```
In [60]: 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'])

In [61]: df4.mean(na_rm=True)
Out[61]: [2.0, u'NaN', u'NaN', u'NaN']
```

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

```
In [62]: df4["A"].mean()
Out[62]: [u'NaN']

In [64]: df4["A"].mean(na_rm=True)
Out[64]: [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:

```
In [5]: df5 = h2o.H2OFrame.from_python(
          np.random.randn(4,100).tolist(),
          column_names=list('ABCD'))
Parse Progress: [###############################] 100%

In [6]: df5.apply(lambda x: x.mean(na_rm=True))
Out[6]: H2OFrame with 1 rows and 4 columns:
          A         B         C        D
0  0.020849 -0.052978 -0.037272 -0.01664
```

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

```
In [26]: df5.apply(lambda row: sum(row), axis=1)
Out[26]: H2OFrame with 100 rows and 1 columns:
         C1
0  0.906854
1  0.790760
2 -0.217604
3 -0.978141
4  2.180175
5 -2.420732
6  0.875716
7 -1.077747
8  2.321706
9 -0.700436
```

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

```
In [49]: df6 = h2o.H2OFrame(
      np.random.randint(0, 7, size=100).tolist())

Parse Progress: [###############################] 100%
Uploaded py5b584604-73ff-4037-9618-c53122cd0343 into cluster with 100 rows
     and 1 cols

In [50]: df6.hist(plot=False)

Parse Progress: [###############################] 100%
Uploaded py8a993d29-e354-44cf-b10e-d97aa6fdfd74 into cluster with 8 rows and
     1 cols
Out[50]: H2OFrame with 8 rows and 5 columns:
   breaks  counts  mids_true   mids   density
0    0.75     NaN        NaN    NaN  0.000000
1    1.50      10        0.0  1.125  0.116667
2    2.25       6        0.5  1.875  0.070000
3    3.00      17        1.0  2.625  0.198333
4    3.75       0        0.0  3.375  0.000000
5    4.50      16        1.5  4.125  0.186667
6    5.25      19        2.0  4.875  0.221667
```

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:

```
In [62]: df7 = h2o.H2OFrame.from_python(
  ['Hello', 'World', 'Welcome', 'To', 'H2O', 'World'])

In [63]: df7
Out[63]: H2OFrame with 6 rows and 1 columns:
        C1
0    Hello
1    World
2  Welcome
3       To
4      H2O
5    World

```

```
In [65]: df7.countmatches('l')
Out[65]: H2OFrame with 6 rows and 1 columns:
   C1
0   2
1   1
2   1
3   0
4   0
5   1
```

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

```
In [89]: df7.sub('l','x')
Out[89]: H2OFrame with 6 rows and 1 columns:
        C1
0    Hexlo
1    Worxd
2  Wexcome
3       To
4      H2O
5    Worxd
```

For global substitution, use `gsub`. Both `sub` and `gsub` support regular expressions. To split strings based on a regular expression:

```
In [86]: df7.strsplit('(l)+')
Out[86]: H2OFrame with 6 rows and 2 columns:
    C1    C2
0   He     o
1  Wor     d
2   We  come
3   To   NaN
4  H2O   NaN
5  Wor     d
```

## 4.5 Merging

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

```
In [98]: df8 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(),
     column_names=list('ABCD'))

Parse Progress: [###############################] 100%
Uploaded py9607f2cc-087a-4d99-ba9f-917ca852c1f2 into cluster with 100 rows
     and 4 cols

In [99]: df9 = h2o.H2OFrame.from_python(
            np.random.randn(100,4).tolist(),
            column_names=list('ABCD'))

Parse Progress: [###############################] 100%
Uploaded pycb8b3aba-77d6-4383-88dd-4729f1f2c314 into cluster with 100 rows
     and 4 cols

In [100]: df8.rbind(df9)
```

```
Out[100]: H2OFrame with 200 rows and 4 columns:
          A         B         C         D
0 -0.095807  0.944757  0.160959  0.271681
1 -0.950010  0.669040  0.664983  1.535805
2  0.172176  0.657167  0.970337 -0.419208
3  0.589829 -0.516749 -1.598524 -1.346773
4  1.044948 -0.281243 -0.411052  0.959717
5  0.498329  0.170340  0.124479 -0.170742
6  1.422841 -0.409794 -0.525356  2.155962
7  0.944803  1.192007 -1.075689  0.017082
```

For successful row binding, the column names and column types between the two H2OFrames must match.

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

```
In [108]: df10 = h2o.H2OFrame.from_python( {
            'A': ['Hello', 'World',
                  'Welcome', 'To',
                  'H2O', 'World'],
            'n': [0,1,2,3,4,5]} )

Parse Progress: [###############################] 100%
Uploaded py57e84cb6-ce29-4d13-afe4-4333b2186c72 into cluster with 6 rows and
     2 cols

In [109]: df11 = h2o.H2OFrame.from_python(np.random.randint(0, 10, size=100).
     tolist9), column_names=['n'])

Parse Progress: [###############################] 100%
Uploaded py090fa929-b434-43c0-81bd-b9c61b553a31 into cluster with 100 rows
     and 1 cols

In [112]: df11.merge(df10)
Out[112]: H2OFrame with 100 rows and 2 columns:
   n      A
0  7    NaN
1  3     To
2  0  Hello
3  9    NaN
4  9    NaN
5  3     To
6  4    H2O
7  4    H2O
8  5  World
9  4    H2O
```

## 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:

```
In [123]: 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),
     'D' : np.random.randn(8)})

Parse Progress: [###############################] 100%
Uploaded pyd297bab5-4e4e-4a89-9b85-f8fecf37f264 into cluster with 8 rows and
     4 cols

In [124]: df12
Out[124]: H2OFrame with 8 rows and 4 columns:
     A         C      B         D
0  foo  1.583908    one -0.441779
1  bar  1.055763    one  1.733467
2  foo -1.200572    two  0.970428
3  bar -1.066722  three -0.311055
4  foo -0.023385    two  0.077905
5  bar  0.758202    two  0.521504
6  foo  0.098259    one -1.391587
7  foo  0.412450  three -0.050374

In [125]: df12.group_by('A').sum().frame
Out[125]: H2OFrame with 2 rows and 4 columns:
     A     sum_C  sum_B     sum_D
0  bar  0.747244      3  1.943915
1  foo  0.870661      5 -0.835406
```

To group by multiple columns and then apply a function:

```
In [127]: df13 = df12.group_by(['A','B']).sum().frame

In [128]: df13
Out[128]: H2OFrame with 6 rows and 4 columns:
     A      B     sum_C     sum_D
0  bar    one  1.055763  1.733467
1  bar    two  0.758202  0.521504
2  foo  three  0.412450 -0.050374
3  foo    one  1.682168 -1.833366
4  foo    two -1.223957  1.048333
5  bar  three -1.066722 -0.311055
```

To join the results into the original H2OFrame:

```
In [129]: df12.merge(df13)
Out[129]: H2OFrame with 8 rows and 6 columns:
     A      B         C         D     sum_C     sum_D
0  foo    one  1.583908 -0.441779  1.682168 -1.833366
1  bar    one  1.055763  1.733467  1.055763  1.733467
2  foo    two -1.200572  0.970428 -1.223957  1.048333
3  bar  three -1.066722 -0.311055 -1.066722 -0.311055
4  foo    two -0.023385  0.077905 -1.223957  1.048333
5  bar    two  0.758202  0.521504  0.758202  0.521504
6  foo    one  0.098259 -1.391587  1.682168 -1.833366
7  foo  three  0.412450 -0.050374  0.412450 -0.050374
```

## 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:

```
In [140]: df14 = h2o.H2OFrame.from_python(
        {'D': ['18OCT2015:11:00:00',
               '19OCT2015:12:00:00',
               '20OCT2015:13:00:00']},
         column_types=['time'])

In [141]: df14.types
Out[141]: {u'D': u'time'}
```

To display the day of the month:

```
In [142]: df14['D'].day()
Out[142]: H2OFrame with 3 rows and 1 columns:
    D
0  18
1  19
2  20
```

To display the day of the week:

```
In [143]: df14['D'].dayOfWeek()
Out[143]: H2OFrame with 3 rows and 1 columns:
     D
0  Sun
1  Mon
2  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:

```
In [145]: df12.types
Out[145]: {u'A': u'Enum', u'B': u'Enum',
           u'C': u'Numeric', u'D': u'Numeric'}
```

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

```
In [148]: df12.anyfactor()
Out[148]: True
```

To view the categorical levels in a single column:

```
In [149]: df12["A"].levels()
Out[149]: ['bar', 'foo']
```

To create categorical interaction features:

```
In [163]: df12.interaction(['A','B'], pairwise=False, max_factors=3,
    min_occurrence=1)

Interactions Progress: [########################] 100%
Out[163]: H2OFrame with 8 rows and 1 columns:
       A_B
0  foo_one
1  bar_one
2  foo_two
3    other
4  foo_two
5    other
6  foo_one
7    other
```

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:

```
In [168]: bb_df = df12.interaction(['B','B'], pairwise=False, max_factors=2,
     min_occurrence=1)

Interactions Progress: [########################] 100%

In [169]: bb_df
Out[169]: H2OFrame with 8 rows and 1 columns:
     B_B
0    one
1    one
2    two
3  other
4    two
5    two
6    one
7  other
```

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

```
In [170]: df15 = df12.cbind(bb_df)

In [171]: df15
Out[171]: H2OFrame with 8 rows and 5 columns:
     A      B         C         D    B_B
0  foo    one  1.583908 -0.441779    one
1  bar    one  1.055763  1.733467    one
2  foo    two -1.200572  0.970428    two
3  bar  three -1.066722 -0.311055  other
4  foo    two -0.023385  0.077905    two
5  bar    two  0.758202  0.521504    two
6  foo    one  0.098259 -1.391587    one
7  foo  three  0.412450 -0.050374  other
```

## 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:

```
In[172]: df = h2o.upload_file("/pathToFile/fileName")
```

To load data from the machine running Python to the machine running H2O:

```
In[173]: df = h2o.import_file("/pathToFile/fileName")
```

To save an H2OFrame on the machine running H2O:

```
In[174]: h2o.export_file(df,"/pathToFile/fileName")
```

To save an H2OFrame on the machine running Python:

```
In[175]: 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
- Generalized Linear Models (GLM)
- Gradient Boosted Regression (GBM)
- Distributed Random Forest (DRF)

The list is growing quickly, 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 $\ell_1$ and $\ell_2$ 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 (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.

### 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 Analytis (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.

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

## 5.2 Running Models

This section describes how to run the following model types:

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

as well as how to generate predictions.

### 5.2.1 Gradient Boosting Models (GBM)

To generate gradient boosting 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()

Java Version: java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)


Starting H2O JVM and connecting: ................. Connection successful!
--------------------------  --------------------------
H2O cluster uptime:         1 seconds 738 milliseconds
H2O cluster version:        3.5.0.3238
H2O cluster name:           H2O_started_from_python
H2O cluster total nodes:    1
H2O cluster total memory:   3.56 GB
H2O cluster total cores:    4
H2O cluster allowed cores:  4
H2O cluster healthy:        True
H2O Connection ip:          127.0.0.1
H2O Connection port:        54321
--------------------------  --------------------------

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

In [4]: iris_data_path = h2o.system_file("iris.csv") # load demonstration
     data

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

Parse Progress: [###############################] 100%
Imported /Users/hank/PythonEnvs/h2obleeding/bin/../h2o_data/iris.csv. Parsed
     150 rows and 5 cols

```

```
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

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
R^2: 0.896209989184
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: 0.00976685294679
R^2: 0.98534972058
LogLoss: 0.0782480971236

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

Scoring History:
    timestamp            duration    number_of_trees    training_MSE
         training_logloss    training_classification_error
--  -------------------  ----------  -----------------  ---------------
     ------------------  -------------------------------
    2015-10-30 08:51:52  0.047 sec   1                  0.282326
         0.758411            0.0266667
    2015-10-30 08:51:52  0.068 sec   2                  0.179214
         0.550506            0.0266667
    2015-10-30 08:51:52  0.086 sec   3                  0.114954
         0.412173            0.0266667
    2015-10-30 08:51:52  0.100 sec   4                  0.0744726
         0.313539            0.02
    2015-10-30 08:51:52  0.112 sec   5                  0.0498319
         0.243514            0.02
    2015-10-30 08:51:52  0.131 sec   6                  0.0340885
         0.19091             0.00666667
    2015-10-30 08:51:52  0.143 sec   7                  0.0241071
         0.151394            0.00666667
    2015-10-30 08:51:52  0.153 sec   8                  0.017606
         0.120882            0.00666667
    2015-10-30 08:51:52  0.165 sec   9                  0.0131024
         0.0975897           0.00666667
    2015-10-30 08:51:52  0.180 sec   10                 0.00976685
         0.0782481           0.00666667

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
C4          192.761                1                    0.774374
C3          54.0381                0.280338             0.217086
C1          1.35271                0.00701757           0.00543422
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 = h2o.system_file("prostate.csv")

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

Parse Progress: [##################################################] 100%
Imported /Users/hank/PythonEnvs/h2obleeding/bin/../h2o_data/prostate.csv.
     Parsed 380 rows and 9 cols

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: 0.202434568594
R^2: 0.158344081513
LogLoss: 0.59112610879
Null degrees of freedom: 379
Residual degrees of freedom: 374
Null deviance: 512.288840185
Residual deviance: 449.25584268
AIC: 461.25584268
AUC: 0.719098211972
Gini: 0.438196423944

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.28443600654:
       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                      0.284436     0.617512  273
max f2                      0.199001     0.77823   360
max f0point5                0.415159     0.636672  108
max accuracy                0.415159     0.705263  108
max precision               0.998619     1         0
max absolute_MCC            0.415159     0.369123  108
max min_per_class_accuracy  0.33266      0.656388  175

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

MSE: 0.209974707772
R^2: 0.126994679038
LogLoss: 0.609520995116
```

```
Null degrees of freedom: 379
Residual degrees of freedom: 373
Null deviance: 515.693473211
Residual deviance: 463.235956288
AIC: 477.235956288
AUC: 0.686706400622
Gini: 0.373412801244

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.326752491231:
       0    1    Error    Rate
-----  ---  ---  -------  -------------
0      135  92   0.4053   (92.0/227.0)
1      48   105  0.3137   (48.0/153.0)
Total  183  197  0.3684   (140.0/380.0)

Maximum Metrics: Maximum metrics at their respective thresholds

metric                      threshold    value     idx
--------------------------  -----------  --------  -----
max f1                      0.326752     0.6       196
max f2                      0.234718     0.774359  361
max f0point5                0.405529     0.632378  109
max accuracy                0.405529     0.702632  109
max precision               0.999294     1         0
max absolute_MCC            0.405529     0.363357  109
max min_per_class_accuracy  0.336043     0.627451  176

Scoring History:
    timestamp            duration    iteration    log_likelihood    objective
--  -------------------  ----------  -----------  ----------------
    -----------
    2015-10-30 08:53:01  0.000 sec   0            256.482           0.674952
    2015-10-30 08:53:01  0.004 sec   1            226.784           0.597118
    2015-10-30 08:53:01  0.005 sec   2            224.716           0.591782
    2015-10-30 08:53:01  0.005 sec   3            224.629           0.59158
    2015-10-30 08:53:01  0.005 sec   4            224.628           0.591579
    2015-10-30 08:53:01  0.006 sec   5            224.628           0.591579
```

### 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
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
--  -------------------  ----------  -----------
     -----------------------------  -------------------------------
    2015-10-30 08:54:39  0.011 sec   0            nan
                                 401.733
    2015-10-30 08:54:39  0.047 sec   1            2.09788
                             191.282
    2015-10-30 08:54:39  0.049 sec   2            0.00316006
                          190.82
    2015-10-30 08:54:39  0.050 sec   3            0.000846952
                         190.757
```

### 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

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

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, nfold=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
--------------------------  ------------------------------------------------------  ---------
GBM_model_1446220160417_30  ['0.2, 15, 4']    0.05105
GBM_model_1446220160417_27  ['0.2, 15, 3']    0.0551088
GBM_model_1446220160417_24  ['0.2, 15, 2']    0.0697714
GBM_model_1446220160417_29  ['0.2, 10, 4']    0.103064
GBM_model_1446220160417_26  ['0.2, 10, 3']    0.106232
GBM_model_1446220160417_23  ['0.2, 10, 2']    0.120161
GBM_model_1446220160417_21  ['0.1, 15, 4']    0.170086
GBM_model_1446220160417_18  ['0.1, 15, 3']    0.171218
GBM_model_1446220160417_15  ['0.1, 15, 2']    0.181186
GBM_model_1446220160417_28  ['0.2, 5, 4']     0.275788
GBM_model_1446220160417_25  ['0.2, 5, 3']     0.27708
GBM_model_1446220160417_22  ['0.2, 5, 2']     0.280413
GBM_model_1446220160417_20  ['0.1, 10, 4']    0.28759
GBM_model_1446220160417_17  ['0.1, 10, 3']    0.288293
GBM_model_1446220160417_14  ['0.1, 10, 2']    0.292993
GBM_model_1446220160417_16  ['0.1, 5, 3']     0.520591
```

```
GBM_model_1446220160417_19  ['0.1, 5, 4']                0.520697
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.

### 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
Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1446220160417_34

Model Summary:
```

```
    number_of_trees    model_size_in_bytes    min_depth    max_depth
         mean_depth    min_leaves    max_leaves    mean_leaves
--  -----------------  ---------------------  -----------  -----------
     ------------  ------------  ------------  -------------
    150                27014                  1            5            4.84
                2             13            9.99333


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

MSE: 0.00162796438754
R^2: 0.997558053419
LogLoss: 0.0152718654494

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_MSE
          training_logloss    training_classification_error
---  -------------------  ----------  -----------------  ----------------
     ------------------  -------------------------------
     2015-10-30 09:00:31  0.007 sec   1.0                0.36363226261
          0.924249463924      0.04
     2015-10-30 09:00:31  0.011 sec   2.0                0.297174376838
          0.788619346614      0.04
     2015-10-30 09:00:31  0.014 sec   3.0                0.242952566898
          0.679995475248      0.04
     2015-10-30 09:00:31  0.017 sec   4.0                0.199051390695
          0.591313594921      0.04
     2015-10-30 09:00:31  0.021 sec   5.0                0.163730865044
          0.517916553872      0.04
---  ---                  ---         ---                ---
     ---                 ---
     2015-10-30 09:00:31  0.191 sec   46.0               0.00239417625265
          0.0192767794713     0.0
     2015-10-30 09:00:31  0.195 sec   47.0               0.00214164838414
          0.0180720391174     0.0
     2015-10-30 09:00:31  0.198 sec   48.0               0.00197748500569
          0.0171428309311     0.0
     2015-10-30 09:00:31  0.202 sec   49.0               0.00179303578037
          0.0161938228014     0.0
     2015-10-30 09:00:31  0.205 sec   50.0               0.00162796438754
          0.0152718654494     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 0x1085cec90>), ('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

In [62]: params = {"standardize__center":    [True, False],             #
     Parameters to test
   ....:           "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 0x108d59200
     >,
          error_score='raise',
          estimator=Pipeline(steps=[('standardize', <h2o.transforms.
               preprocessing.H2OScaler object at 0x108d50150>), ('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      3.16438   0.180179    0.143787
Proportion of Variance  0.994721  0.00322501  0.00205383
Cumulative Proportion   0.994721  0.997946    1


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

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

Model Summary:
    number_of_trees    model_size_in_bytes    min_depth    max_depth    mean_depth    min_leaves    max_leaves    mean_leaves
--  -----------------  ---------------------  -----------  -----------  ------------  ------------  ------------  -------------
    20                 2743                   3            3            3             4             8             6.35


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

MSE: 0.0566740346323
R^2: 0.916793146878
Mean Residual Deviance: 0.0566740346323

Scoring History:
    timestamp            duration    number_of_trees    training_MSE    training_deviance
--  -------------------  ----------  -----------------  --------------  -------------------
    2015-10-30 09:04:46  0.001 sec   1                  0.477453        0.477453
    2015-10-30 09:04:46  0.002 sec   2                  0.344635        0.344635
    2015-10-30 09:04:46  0.003 sec   3                  0.259176        0.259176
    2015-10-30 09:04:46  0.004 sec   4                  0.200125        0.200125
    2015-10-30 09:04:46  0.005 sec   5                  0.160051        0.160051
    2015-10-30 09:04:46  0.006 sec   6                  0.132315        0.132315
    2015-10-30 09:04:46  0.006 sec   7                  0.114554        0.114554
    2015-10-30 09:04:46  0.007 sec   8                  0.100317        0.100317
    2015-10-30 09:04:46  0.008 sec   9                  0.0890903       0.0890903
    2015-10-30 09:04:46  0.009 sec   10                 0.0810115       0.0810115
    2015-10-30 09:04:46  0.009 sec   11                 0.0760616       0.0760616
    2015-10-30 09:04:46  0.010 sec   12                 0.0725191       0.0725191
```

```
    2015-10-30 09:04:46  0.011 sec   13                 0.0694355
         0.0694355
    2015-10-30 09:04:46  0.012 sec   14                 0.06741
         0.06741
    2015-10-30 09:04:46  0.012 sec   15                 0.0655487
         0.0655487
    2015-10-30 09:04:46  0.013 sec   16                 0.0624041
         0.0624041
    2015-10-30 09:04:46  0.014 sec   17                 0.0615533
         0.0615533
    2015-10-30 09:04:46  0.015 sec   18                 0.058708
         0.058708
    2015-10-30 09:04:46  0.015 sec   19                 0.0579205
         0.0579205
    2015-10-30 09:04:46  0.016 sec   20                 0.056674
         0.056674

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
PC1         237.674                1                    0.913474
PC3         12.8597                0.0541066            0.0494249
PC2         9.65329                0.0406157            0.0371014
Pipeline(steps=[('standardize', <h2o.transforms.preprocessing.H2OScaler
     object at 0x104f2a490>), ('pca', ), ('gbm', )])
```

# 6 References

H2O.ai Team. **H2O website**, 2016. URL http://h2o.ai

H2O.ai Team. **H2O documentation**, 2016. 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**, 2016. URL https://github.com/h2oai

H2O.ai Team. **H2O Datasets**, 2016. URL http://data.h2o.ai

H2O.ai Team. **H2O JIRA**, 2016. URL https://jira.h2o.ai

H2O.ai Team. **H2Ostream**, 2016. URL https://groups.google.com/d/forum/h2ostream