

# Machine Learning with Python and H2O

---

PASHA STETSENKO

EDITED BY: ANGELA BARTZ

---

<http://h2o.ai/resources/>

May 2021: 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

©2016-2021 H2O.ai, Inc. All Rights Reserved.

May 2021: Fifth Edition

Photos by ©H2O.ai, Inc.

All copyrights belong to their respective owners.  
While every precaution has been taken in the  
preparation of this book, the publisher and  
authors assume no responsibility for errors or  
omissions, or for damages resulting from the  
use of the information contained herein.

Printed in the United States of America.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>What is H2O?</b>	<b>5</b>
2.1	Example Code . . . . .	6
2.2	Citation . . . . .	6
<b>3</b>	<b>Installation</b>	<b>6</b>
3.1	Installation in Python . . . . .	7
<b>4</b>	<b>Data Preparation</b>	<b>7</b>
4.1	Viewing Data . . . . .	9
4.2	Selection . . . . .	10
4.3	Missing Data . . . . .	12
4.4	Operations . . . . .	13
4.5	Merging . . . . .	16
4.6	Grouping . . . . .	17
4.7	Using Date and Time Data . . . . .	18
4.8	Categoricals . . . . .	19
4.9	Loading and Saving Data . . . . .	21
<b>5</b>	<b>Machine Learning</b>	<b>21</b>
5.1	Modeling . . . . .	21
5.1.1	Supervised Learning . . . . .	22
5.1.2	Unsupervised Learning . . . . .	23
5.1.3	Miscellaneous . . . . .	23
5.2	Running Models . . . . .	23
5.2.1	Gradient Boosting Machine (GBM) . . . . .	24
5.2.2	Generalized Linear Models (GLM) . . . . .	27
5.2.3	K-means . . . . .	30
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
<b>6</b>	<b>Acknowledgments</b>	<b>38</b>
<b>7</b>	<b>References</b>	<b>38</b>

## 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

# 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>.

## 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
```

## Citation

To cite this booklet, use the following:

Aiello, S., Cliff, C., Roark, H., Rehak, L., Stetsenko, P., and Bartz, A. (May 2021). *Machine Learning with Python and H2O*. <http://h2o.ai/resources/>.

## 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.

## 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:

```
1 import h2o
2
3 # Start H2O on your local machine
4 h2o.init()
5
6 # Get help
7 help(h2o.estimators.glm.H2OGeneralizedLinearEstimator)
8 help(h2o.estimators.gbm.H2OGradientBoostingEstimator)
9
10 # Show a demo
11 h2o.demo("glm")
12 h2o.demo("gbm")
```

## 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](http://docs.h2o.ai).

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

```
1 import h2o
2 h2o.init()
```

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

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

To create an H2OFrame object from a Python tuple:

```
1 df = h2o.H2OFrame(zip(*(1, 2, 3), ('a', 'b', 'c'), (0.1, 0.2, 0.3)))
2
3 # View the H2OFrame
4 df
5
6 #      C1  C2      C3
7 # ----  ----  ----
8 #      1  a      0.1
9 #      2  b      0.2
10 #      3  c      0.3
11 #
12 # [3 rows x 3 columns]
```

To create an H2OFrame object from a Python list:

```
1 df = h2o.H2OFrame(zip(*[[1, 2, 3], ['a', 'b', 'c'], [0.1, 0.2, 0.3]]))
2
3 # View the H2OFrame
4 df
5
6 #      C1      C2      C3
7 # ---- ---- ----
8 #      1      a      0.1
9 #      2      b      0.2
10 #      3      c      0.3
11 #
12 # [3 rows x 3 columns]
```

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

```
1 df = h2o.H2OFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c'], 'C': [0.1, 0.2, 0.3]})
2
3 # View the H2OFrame
4 df
5
6 #      A      C      B
7 # ---  ---  ---
8 #    1  0.1  a
9 #    2  0.2  b
10 #    3  0.3  c
11 #
12 # [3 rows x 3 columns]
```

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

[illegible]



```

8 df2
9
10 #   A   C   B   D
11 # ---  ---  ---  ---
12 #   1 hello a   1.42618e+12
13 #   2 all  a   1.42627e+12
14 #   3 world b   1.42636e+12
15 #
16 # [3 rows x 4 columns]

```

To display the column types:

```

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

```

## Viewing Data

To display the top and bottom of an H2OFrame:

```

1 import numpy as np
2 df = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
   list('ABCD'))
3
4 # View top 10 rows of the H2OFrame
5 df.head()
6
7 #           A           B           C           D
8 # -----
9 # -0.613035 -0.425327 -1.92774 -2.1201
10 # -1.26552 -0.241526 -0.0445104 1.90628
11 # 0.763851 0.0391609 -0.500049 0.355561
12 # -1.24842 0.912686 -0.61146 1.94607
13 # 2.1058 -1.83995 0.453875 -1.69911
14 # 1.7635 0.573736 -0.309663 -1.51131
15 # -0.781973 0.051883 -0.403075 0.569406
16 # 1.40085 1.91999 0.514212 -1.47146
17 # -0.746025 -0.632182 1.27455 -1.35006
18 # -1.12065 0.374212 0.232229 -0.602646
19 #
20 # [10 rows x 4 columns]
21
22 # View bottom 5 rows of the H2OFrame
23 df.tail(5)
24
25 #           A           B           C           D
26 # -----
27 # 1.00098 -1.43183 -0.322068 0.374401
28 # 1.16553 -1.23383 -1.71742 1.01035
29 # -1.62351 -1.13907 2.1242 -0.275453
30 # -0.479005 -0.0048988 0.224583 0.219037
31 # -0.74103 1.13485 0.732951 1.70306
32 #
33 # [5 rows x 4 columns]

```

To display the column names:

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

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

```
1 df.describe()
2
3 # Rows: 100 Cols: 4
4 #
5 # Chunk compression summary:
6 # chunk_type      chunkname    count    count_%    size    size_%
7 # -----
8 # 64-bit Reals      C8D         4        100      3.4 KB    100
9 #
10 # Frame distribution summary:
11 #
12 #      size    #_rows    #_chunks_per_col    #_chunks
13 # -----
14 # 127.0.0.1:54321    3.4 KB    100          1          4
15 # mean              3.4 KB    100          1          4
16 # min               3.4 KB    100          1          4
17 # max               3.4 KB    100          1          4
18 # stddev            0 B       0           0           0
19 # total             3.4 KB    100          1          4
20 #
21 #      A      B      C      D
22 # -----
23 # type    real    real    real    real
24 # mins   -2.49822 -2.37446 -2.45977 -3.48247
25 # mean   -0.01062 -0.23159  0.11423 -0.16228
26 # maxs    2.59380  1.91998  3.13014  2.39057
27 # sigma   1.04354  0.90576  0.96133  1.02608
28 # zeros    0        0        0        0
29 # missing  0        0        0        0
```

## Selection

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

```
1 df['A']
2
3 #      A
4 # -----
5 # -0.613035
6 # -1.265520
7 #  0.763851
8 # -1.248425
9 #  2.105805
10 #  1.763502
11 # -0.781973
12 #  1.400853
13 # -0.746025
14 # -1.120648
15 #
16 # [100 rows x 1 column]
```

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

```

1 df[1]
2
3 #           B
4 # -----
5 # -0.425327
6 # -0.241526
7 #  0.039161
8 #  0.912686
9 # -1.839950
10 #  0.573736
11 #  0.051883
12 #  1.919987
13 # -0.632182
14 #  0.374212
15 #
16 # [100 rows x 1 column]
```

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

```

1 df[['B', 'C']]
2
3 #           B           C
4 # -----
5 # -0.425327 -1.927737
6 # -0.241526 -0.044510
7 #  0.039161 -0.500049
8 #  0.912686 -0.611460
9 # -1.839950  0.453875
10 #  0.573736 -0.309663
11 #  0.051883 -0.403075
12 #  1.919987  0.514212
13 # -0.632182  1.274552
14 #  0.374212  0.232229
15 #
16 # [100 rows x 2 columns]
```

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

```

1 df[0:2]
2
3 #           A           B
4 # -----
5 # -0.613035 -0.425327
6 # -1.265520 -0.241526
7 #  0.763851  0.039161
8 # -1.248425  0.912686
9 #  2.105805 -1.839950
10 #  1.763502  0.573736
11 # -0.781973  0.051883
12 #  1.400853  1.919987
13 # -0.746025 -0.632182
14 # -1.120648  0.374212
15 #
16 # [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:

```

1 df[2:7, :]
2
3 #           A           B           C           D
4 # -----
5 #  1.31828    0.316926    0.970535    0.218061
6 # -0.18547    0.207064    1.3229     -0.432614
7 # -0.424018  -1.72759    0.356871    0.206214
8 #  1.3377     1.10761   -0.280443    0.0964197
9 # -0.385682    0.190449    0.760816    1.92447
10 #
11 # [5 rows x 4 columns]
```

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

```

1 df2[ df2["B"] == "a", :]
2
3 #   A   C   B           D
4 # ---  ---  ---  -----
5 #  1  hello a   1.42618e+12
6 #  2  all  a   1.42627e+12
7 #
8 # [2 rows x 4 columns]
```

## 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:

```

1 df3 = h2o.H2OFrame.from_python(
2     {'A': [1, 2, 3, None, ''],
3      'B': ['a', 'a', 'b', 'NA', 'NA'],
4      'C': ['hello', 'all', 'world', None, None],
5      'D': ['12MAR2015:11:00:00', None,
6           '13MAR2015:12:00:00', None,
7           '14MAR2015:13:00:00']},
8     column_types=['numeric', 'enum', 'string', 'time'])
```

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

```

1 df3["A"].isna()
2
3 #   C1
4 # ---
5 #    0
6 #    0
```

```

7 # 0
8 # 1
9 # 1
10 #
11 # [5 rows x 1 column]

```

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

```

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

```

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

```

1 df3.isna()
2
3 #   C1    C2    C3    C4
4 # ---  ---  ---  ---
5 #  0    0    0    0
6 #  0    0    0    1
7 #  0    0    0    0
8 #  0    0    0    1
9 #  0    0    0    0
10 #
11 # [5 rows x 4 columns]

```

## 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:

```

1 df4 = h2o.H2OFrame.from_python(
2     {'A': [1, 2, 3, None, ''],
3      'B': ['a', 'a', 'b', 'NA', 'NA'],
4      'C': ['hello', 'all', 'world', None, None],
5      'D': ['12MAR2015:11:00:00', None,
6           '13MAR2015:12:00:00', None,
7           '14MAR2015:13:00:00']},
8     column_types=['numeric', 'enum', 'string', 'time'])
9
10 df4.mean(na_rm=True)
11 # [2.0, nan, nan, nan]

```

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

```

1 df4["A"].mean()
2 # [nan]
3
4 df4["A"].mean(na_rm=True)
5 # [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:

```

1 df5 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
2   list('ABCD'))
3 df5.apply(lambda x: x.mean(na_rm=True))
4
5 # H2OFrame:
6 #           A           B           C           D
7 # -----
8 # 0.0304506  0.0334168  -0.0374976  0.0520486
9 #
10 # [1 row x 4 columns]
```

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

```

1 df5.apply(lambda row: row.sum(), axis=1)
2
3 # H2OFrame:
4 #           C1
5 # -----
6 # -0.388512
7 #  1.67669
8 # -2.56216
9 # -0.277616
10 #  1.13655
11 # -0.575992
12 # -3.49258
13 #  0.776883
14 # -0.778604
15 #  2.30617
16 #
17 # [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:

```

1 df6 = h2o.H2OFrame.from_python(np.random.randn(100,1).tolist())
2
3 df6.hist(plot=False)
4
5 # Parse Progress: [#####] 100%
6 #   breaks      counts      mids_true      mids      density
7 # -----
8 # -1.51121          nan          nan          nan          0
9 # -0.868339          9      -1.07704      -1.18977      0.139997
10 # -0.225468         12      -0.73561      -0.546904      0.186663
11 #  0.417403         18      -0.413093      0.0959675      0.279994
12 #  1.06027          26      -0.10108      0.738839      0.404436
13 #  1.70315          22      0.214337      1.38171      0.342215
14 #  2.34602           7      0.607727      2.02458      0.108887
15 #  2.98889           6      0.860969      2.66745      0.0933313
16 #
17 # [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:

```

1 df7 = h2o.H2OFrame.from_python(['Hello', 'World', 'Welcome', 'To', 'H2O', '
  World'])
2
3 # View the H2OFrame
4 df7
5
6 # C1      C2      C3      C4      C5      C6
7 # ----
8 # Hello  World  Welcome  To      H2O    World
9 #
10 # [1 row x 6 columns]
11
12 # Find how many times "l" appears in each string
13 df7.countmatches('l')
14
15 # C1      C2      C3      C4      C5      C6
16 # ----
17 # 2      1      1      0      0      1
18 #
19 # [1 row x 6 columns]
```

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

```

1 df7.sub('l','x')
2
3 # C1      C2      C3      C4      C5      C6
4 # ----
5 # 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:

```

1 df7.strsplit('(l)+')
2
3 # C1      C2      C3      C4      C5      C6      C7      C8      C9      C10
4 # ----
5 # He      o      Wor      d      We      come      To      H2O      Wor      d
6 #
7 # [1 row x 10 columns]
```

## Merging

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

```

1 # Create a frame of random numbers w/ 100 rows
2 df8 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
   list('ABCD'))
3
4 # Create a second frame of random numbers w/ 100 rows
5 df9 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(), column_names=
   list('ABCD'))
6
7 # Combine the two frames, adding the rows from df9 to df8
8 df8.rbind(df9)
9
10 #
11 # ----- A ----- B ----- C ----- D -----
12 # 1.11442 1.31272 0.250418 1.73182
13 # -1.61876 0.428622 -1.16684 -0.032936
14 # 0.637249 -0.48904 1.55848 0.669266
15 # 0.00355574 -0.40736 -0.979222 -0.395017
16 # 0.218243 -0.154004 -0.219537 -0.750664
17 # -0.047789 0.306318 0.557441 -0.319108
18 # -1.45013 -0.614564 0.472257 -0.456181
19 # -0.594333 -0.435832 -0.0257311 0.548708
20 # 0.571215 -1.22759 -2.01855 -0.491638
21 # -0.697252 -0.864301 -0.542508 -0.152953
22 #
23 # [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:

```

1 df8.cbind(df9)
2
3 #      A      B      C      D      A0      B0      C0      D0
4 # -----
5 # -0.09 0.944 0.160 0.271 -0.351 1.66 -2.32 -0.86
6 # -0.95 0.669 0.664 1.535 -0.633 -1.78 0.32 1.27
7 # 0.17 0.657 0.970 -0.419 -1.413 -0.51 0.64 -1.25
8 # 0.58 -0.516 -1.598 -1.346 0.711 1.09 0.05 0.63
9 # 1.04 -0.281 -0.411 0.959 -0.009 -0.47 0.41 -0.52
10 # 0.49 0.170 0.124 -0.170 -0.722 -0.79 -0.91 -2.09
11 # 1.42 -0.409 -0.525 2.155 -0.841 -0.19 0.13 0.63
12 # 0.94 1.192 -1.075 0.017 0.167 0.54 0.52 1.42
13 # -0.53 0.777 -1.090 -2.237 -0.693 0.24 -0.56 1.45
14 # 0.34 -0.456 -1.220 -0.456 -0.315 1.10 1.38 -0.05
15 #
16 # [100 rows x 8 columns]

```



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

```

1 df10 = h2o.H2OFrame.from_python( {
2     'A': ['Hello', 'World', 'Welcome', 'To', 'H2O', 'World'],
3     'n': [0,1,2,3,4,5]} )
4
5 # Create a single-column, 100-row frame
6 # Include random integers from 0-5
7 df11 = h2o.H2OFrame.from_python(np.random.randint(0,6,(100,1)), column_names=
8     list('n'))
9
10 # Combine column "n" from both datasets
11 df11.merge(df10)
12
13 #      n  A
14 # ---  ---
15 #    2 Welcome
16 #    5 World
17 #    4 H2O
18 #    2 Welcome
19 #    3 To
20 #    3 To
21 #    1 World
22 #    1 World
23 #    3 To
24 #    1 World
25 # [100 rows x 2 columns]
```

## 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:

```

1 df12 = h2o.H2OFrame(
2     {'A' : ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
3      'B' : ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
4      'C' : np.random.randn(8).tolist(),
5      'D' : np.random.randn(8).tolist()})
6
7 # View the H2OFrame
8 df12
9
10 #      A          C      B          D
11 # ---  ---
12 # foo -0.710095    one    0.253189
13 # bar -0.165891    one   -0.433233
14 # foo -1.51996    two    1.12321
15 # bar  2.25083    three   0.512449
16 # foo -0.618324    two    1.35158
17 # bar  0.0817828    two    0.00830419
```

```

18 # foo 0.634827 one 1.25897
19 # foo 0.879319 three 1.48051
20 #
21 # [8 rows x 4 columns]
22
23 df12.group_by('A').sum().frame
24
25 # A sum_C sum_B sum_D
26 # ---
27 # bar 2.16672 3 0.0875206
28 # foo -1.33424 5 5.46746
29 #
30 # [2 rows x 4 columns]

```

To group by multiple columns and then apply a function:

```

1 df13 = df12.group_by(['A', 'B']).sum().frame
2
3 # View the H2OFrame
4 df13
5
6 # A B sum_C sum_D
7 # ---
8 # bar one -0.165891 -0.433233
9 # bar three 2.25083 0.512449
10 # bar two 0.0817828 0.00830419
11 # foo one -0.0752683 1.51216
12 # foo three 0.879319 1.48051
13 # foo two -2.13829 2.47479
14 #
15 # [6 rows x 4 columns]

```

Use merge to join the results into the original H2OFrame:

```

1 df12.merge(df13)
2
3 # A B C D sum_C sum_D
4 # ---
5 # foo one -0.710095 0.253189 -0.0752683 1.51216
6 # bar one -0.165891 -0.433233 -0.165891 -0.433233
7 # foo two -1.51996 1.12321 -2.13829 2.47479
8 # bar three 2.25083 0.512449 2.25083 0.512449
9 # foo two -0.618324 1.35158 -2.13829 2.47479
10 # bar two 0.0817828 0.00830419 0.0817828 0.00830419
11 # foo one 0.634827 1.25897 -0.0752683 1.51216
12 # foo three 0.879319 1.48051 0.879319 1.48051
13 #
14 # [8 rows by 6 columns]

```

## 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:

```

1 df14 = h2o.H2OFrame.from_python(
2     {'D': ['18OCT2015:11:00:00',
3          '19OCT2015:12:00:00',
4          '20OCT2015:13:00:00']},
5     column_types=['time'])
6
7 df14.types
8 # {u'D': u'time'}
```

To display the day of the month:

```

1 df14['D'].day()
2
3 # D
4 # ---
5 # 18
6 # 19
7 # 20
```

To display the day of the week:

```

1 df14['D'].dayOfWeek()
2
3 # D
4 # ---
5 # Sun
6 # Mon
7 # Tue
```

## 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:

```

1 df12.types
2 # {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:

```

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

To view the categorical levels in a single column:

```

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

To create categorical interaction features:

```

1 df12.interaction(['A','B'], pairwise=False, max_factors=3, min_occurrence=1)
2
3 # A_B
4 # -----
5 # foo_one
6 # bar_one
7 # foo_two
8 # other
9 # foo_two
10 # other
11 # foo_one
12 # other
13 #
14 # [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:

```

1 bb_df = df12.interaction(['B','B'], pairwise=False, max_factors=2,
2 min_occurrence=1)
3
4 # View H2OFrame
5 bb_df
6
7 # B_B
8 # -----
9 # one
10 # one
11 # two
12 # other
13 # two
14 # two
15 # one
16 # other
17 #
18 # [8 rows x 1 column]
```

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

```

1 df15 = df12.cbind(bb_df)
2
3 # View H2OFrame
4 df15
5
6 # A C B D B_B
7 # ---
8 # foo -0.809171 one 1.79059 one
9 # bar 0.216644 one 2.88524 one
10 # foo -0.033664 two 0.61205 two
11 # bar 0.985545 three 0.357742 other
12 # foo -2.15563 two 0.0456449 two
13 # bar -0.0170454 two -1.33625 two
14 # foo 1.32524 one 0.308092 one
15 # foo -0.546305 three -0.92675 other
16 #
17 # [8 rows x 5 columns]
```

## 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 | • ARFF |
| • ORC                   | • XLS  |
| • SVMLite               | • XLSX |
| • Parquet               | • AVRO |

To load data from the same machine running H2O:

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

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

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

To save an H2OFrame on the machine running H2O:

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

To save an H2OFrame on the machine running Python:

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

## Machine Learning

The following sections describe some common model types and features.

### 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](http://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
- XGBoost
- 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](http://www.h2o.ai) to see the latest additions.

## 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 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."

**XGBoost:** XGBoost is an optimized gradient boosting library that implements machine learning algorithms under the Gradient Boosting Machine (GBM) framework. For many problems, XGBoost is the one of the best GBM frameworks today. In other cases, the H2O GBM algorithm comes out on top. Both implementations are available on the H2O platform.

## 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.

## 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.

## 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.

## 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.

```
1 In [1]: import h2o
2
3 In [2]: h2o.init()
4
5 Checking whether there is an H2O instance running at http://localhost:54321
6 ..... not found.
7 Attempting to start a local H2O server...
8   Java Version: java version "1.8.0_25"; Java(TM) SE Runtime Environment (
9     build 1.8.0_25-b17); Java HotSpot(TM) 64-Bit Server VM (build 25.25-
10      b02, mixed mode)
11   Starting server from /usr/local/h2o_jar/h2o.jar
12   Ice root: /var/folders/yl/cq5nhky53hjc19wrqxt39kz80000gn/T/tmpHpRzVe
13   JVM stdout: /var/folders/yl/cq5nhky53hjc19wrqxt39kz80000gn/T/tmpHpRzVe/
14     h2o_techwriter_started_from_python.out
15   JVM stderr: /var/folders/yl/cq5nhky53hjc19wrqxt39kz80000gn/T/tmpHpRzVe/
16     h2o_techwriter_started_from_python.err
17   Server is running at http://127.0.0.1:54321
18 Connecting to H2O server at http://127.0.0.1:54321 ... successful.
19
20 In [3]: from h2o.estimators.gbm import H2OGradientBoostingEstimator
21
22 In [4]: iris_data_path = "http://h2o-public-test-data.s3.amazonaws.com/
23     smalldata/iris/iris.csv" # load demonstration data
24
25 In [5]: iris_df = h2o.import_file(path=iris_data_path)
26
27 Parse Progress: [#####] 100%
28
29 In [6]: iris_df.describe()
30 Rows:150 Cols:5
31
32 Chunk compression summary:
33 chunktype chunkname count count_% size size_%
34 -----
35 1-Byte Int C1 1 20 218B 18.890
36 1-Byte Flt C2 4 80 936B 81.109
37
38 Frame distribution summary:
39 size rows chunks/col chunks
40 -----
41 127.0.0.1:54321 1.1KB 150 1 5
42 mean 1.1KB 150 1 5
43 min 1.1KB 150 1 5
44 max 1.1KB 150 1 5
45 stddev 0 B 0 0
46 total 1.1 KB 150 1 5
47
48 C1 C2 C3 C4 C5
49 -----
50 type real real real real enum
```



```
45 mins      4.3      2.0      1.0      0.1      0.0
46 mean      5.8433333333 3.054      3.75866666667 1.19866666667 NaN
47 maxs      7.9      4.4      6.9      2.5      2.0
48 sigma     0.828066127978 0.433594311362 1.76442041995 0.763160741701 NaN
49 zeros      0      0      0      0      50
50 missing    0      0      0      0      0
51
52
53 In [7]: gbm_regressor = H2OGradientBoostingEstimator(distribution="gaussian",
54               ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")
55
56 In [8]: gbm_regressor.train(x=range(1,iris_df.ncol), y=0, training_frame=
57               iris_df)
58
59 gbm Model Build Progress: [#####] 100%
60
61 In [9]: gbm_regressor
62 Out[9]: Model Details
63 =====
64 H2OGradientBoostingEstimator: Gradient Boosting Machine
65 Model Key: GBM_model_python_1446220160417_2
66
67 Model Summary:
68   number_of_trees      |      10
69   model_size_in_bytes  |     1535
70   min_depth            |      3
71   max_depth            |      3
72   mean_depth           |      3
73   min_leaves           |      7
74   max_leaves           |      8
75   mean_leaves          |     7.8
76
77 ModelMetricsRegression: gbm
78 ** Reported on train data. **
79
80 MSE: 0.0706936802293
81 RMSE: 0.265882831769
82 MAE: 0.219981056849
83 RMSLE: 0.039185537448
84 Mean Residual Deviance: 0.0706936802293
85
86 Scoring History:
87   timestamp      duration      number_of_trees      training_MSE
88   training_deviance
89   -----
90   2015-10-30 08:50:00 0.121 sec  1      0.472445
91   0.472445
92   2015-10-30 08:50:00 0.151 sec  2      0.334868
93   0.334868
94   2015-10-30 08:50:00 0.162 sec  3      0.242847
95   0.242847
96   2015-10-30 08:50:00 0.175 sec  4      0.184128
97   0.184128
98   2015-10-30 08:50:00 0.187 sec  5      0.14365
99   0.14365
100  2015-10-30 08:50:00 0.197 sec  6      0.116814
101  0.116814
102  2015-10-30 08:50:00 0.208 sec  7      0.0992098
103  0.0992098
104  2015-10-30 08:50:00 0.219 sec  8      0.0864125
105  0.0864125
```

95	2015-10-30 08:50:00	0.229 sec	9	0.077629
	0.077629			
96	2015-10-30 08:50:00	0.238 sec	10	0.0706937
	0.0706937			
97				
98	Variable Importances:			
99	variable	relative_importance	scaled_importance	percentage
100	-----	-----	-----	-----
101	C3	227.562	1	0.894699
102	C2	15.1912	0.0667563	0.0597268
103	C5	9.50362	0.0417627	0.037365
104	C4	2.08799	0.00917544	0.00820926

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

```
1 In [10]: gbm_classifier = H2OGradientBoostingEstimator(distribution="
2         multinomial", ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")
3
4 In [11]: gbm_classifier.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1,
5         training_frame=iris_df)
6
7 gbm Model Build Progress: [#####] 100%
8
9 In [12]: gbm_classifier
10 Out[12]: Model Details
11
12 =====
13 H2OGradientBoostingEstimator : Gradient Boosting Machine
14 Model Key: GBM_model_python_1446220160417_4
15
16 Model Summary:
17
18   number_of_trees  model_size_in_bytes  min_depth  max_depth
19   mean_depth      min_leaves    max_leaves  mean_leaves
20   -----
21   30              3933              1           3
22   2.93333        2              8         5.86667
23
24 ModelMetricsMultinomial: gbm
25 ** Reported on train data. **
26
27 MSE: 0.00976685303214
28 RMSE: 0.0988273900907
29 LogLoss: 0.0782480973696
30 Mean Per-Class Error: 0.00666666666667
31 Confusion Matrix: vertical: actual; across: predicted
32
33   Iris-setosa  Iris-versicolor  Iris-virginica  Error  Rate
34   -----
35   50          0                0            0    0 / 50
36   0           49                1           0.02  1 / 50
37   0           0                50            0    0 / 50
38   50          49                51           0.00666667 1 / 150
39
40 Top-3 Hit Ratios:
41 k    hit_ratio
42 ---
43 1    0.993333
44 2     1
45 3     1
```

41	Scoring History:			
42	timestamp	duration	number_of_trees	training_MSE
43	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	Variable Importances:			
56	variable	relative_importance	scaled_importance	percentage
57	-----	-----	-----	-----
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

## 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)
```

```
38
39 glm Model Build Progress: [#####] 100%
40
41 In [20]: glm_classifier
42 Out[20]: Model Details
43 =====
44 H2OGeneralizedLinearEstimator : Generalized Linear Model
45 Model Key: GLM_model_python_1446220160417_6
46
47 GLM Model: summary
48
49     family      link      regularization
50     number_of_predictors_total  number_of_active_predictors
51     number_of_iterations      training_frame
52
53     -----
54     binomial  logit      Elastic Net (alpha = 0.5, lambda = 3.251E-4 ) 6
55                                     6
56                                     py_3
57
58 ModelMetricsBinomialGLM: glm
59 ** Reported on train data. **
60
61 MSE: 0.202442565125
62 RMSE: 0.449936178947
63 LogLoss: 0.591121990582
64 Null degrees of freedom: 379
65 Residual degrees of freedom: 374
66 Null deviance: 512.288840185
67 Residual deviance: 449.252712842
68 AIC: 461.252712842
69 AUC: 0.718954248366
70 Gini: 0.437908496732
71
72 Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.282384349078:
73
74     0      1      Error      Rate
75     ----
76 0      80     147  0.6476  (147.0/227.0)
77 1      19     134  0.1242  (19.0/153.0)
78 Total  99     281  0.4368  (166.0/380.0)
79
80 Maximum Metrics: Maximum metrics at their respective thresholds
81
82 metric                threshold      value      idx
83 -----
84 max f1                 0.282384      0.617849    276
85 max f2                 0.198777      0.77823     360
86 max f0point5           0.415125      0.636672    108
87 max accuracy           0.415125      0.705263    108
88 max precision          0.998613      1           0
89 max recall             0.198777      1           360
90 max specificity         0.998613      1           0
91 max absolute_mcc       0.415125      0.369123    108
92 max min_per_class_accuracy 0.332648      0.656388    175
93 max mean_per_class_accuracy 0.377454      0.67326     123
94
95 Gains/Lift Table: Avg response rate: 40.26 %
96
97 ModelMetricsBinomialGLM: glm
98 ** Reported on cross-validation data. **
99
```

```
94 MSE: 0.209698776592
95 RMSE: 0.457928789871
96 LogLoss: 0.610086165597
97 Null degrees of freedom: 379
98 Residual degrees of freedom: 374
99 Null deviance: 513.330704712
100 Residual deviance: 463.665485854
101 AIC: 475.665485854
102 AUC: 0.688203622124
103 Gini: 0.376407244249
104 Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.339885371023:
105      0      1      Error      Rate
106 -----
107 0      154    73    0.3216    (73.0/227.0)
108 1       53   100    0.3464    (53.0/153.0)
109 Total  207   173    0.3316    (126.0/380.0)
110 Maximum Metrics: Maximum metrics at their respective thresholds
111
112 metric                        threshold      value      idx
113 -----
114 max f1                        0.339885      0.613497    172
115 max f2                        0.172551      0.773509    376
116 max f0point5                  0.419649      0.615251    105
117 max accuracy                  0.447491      0.692105     93
118 max precision                 0.998767       1           0
119 max recall                    0.172551       1          376
120 max specificity               0.998767       1           0
121 max absolute_mcc              0.419649      0.338849    105
122 max min_per_class_accuracy    0.339885      0.653595    172
123 max mean_per_class_accuracy  0.339885      0.666004    172
124 Gains/Lift Table: Avg response rate: 40.26 %
125
126
127 Scoring History:
128      timestamp      duration      iteration      log_likelihood      objective
129  -----
130  2016-08-25 12:54:20  0.000 sec    0          256.144
131      0.674064
132  2016-08-25 12:54:20  0.055 sec    1          226.961
133      0.597573
134  2016-08-25 12:54:20  0.092 sec    2          224.728
135      0.591813
136  2016-08-25 12:54:20  0.125 sec    3          224.627
137      0.591578
138  2016-08-25 12:54:20  0.157 sec    4          224.626
139      0.591578
```

K-means

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

```
1 In [21]: from h2o.estimators.kmeans import H2OKMeansEstimator
2
3 In [22]: cluster_estimator = H2OKMeansEstimator(k=3)
4
5 In [23]: cluster_estimator.train(x=[0,1,2,3], training_frame=iris_df)
```

```

6
7 kmeans Model Build Progress: [#####] 100%
8
9 In [24]: cluster_estimator
10 Out[24]: Model Details
11 =====
12 H2OKMeansEstimator : K-means
13 Model Key: K-means_model_python_1446220160417_8
14
15 Model Summary:
16      number_of_rows  number_of_clusters  number_of_categorical_columns
17      number_of_iterations  within_cluster_sum_of_squares
18      total_sum_of_squares  between_cluster_sum_of_squares
19
20      -----
21      150              3              0
22      4              190.757              596
23      405.243
24
25 ModelMetricsClustering: kmeans
26 ** Reported on train data. **
27
28 MSE: NaN
29 RMSE: NaN
30 Total Within Cluster Sum of Square Error: 190.756926265
31 Total Sum of Square Error to Grand Mean: 596.0
32 Between Cluster Sum of Square Error: 405.243073735
33
34 Centroid Statistics:
35      centroid  size  within_cluster_sum_of_squares
36      -----
37      1          96  149.733
38      2          32  17.292
39      3          22  23.7318
40
41 Scoring History:
42      timestamp  duration  iteration  avg_change_of_std_centroids
43      within_cluster_sum_of_squares
44      -----
45      2016-08-25 13:03:36  0.005 sec  0  nan
46      385.505
47      2016-08-25 13:03:36  0.029 sec  1  1.37093
48      173.769
49      2016-08-25 13:03:36  0.029 sec  2  0.184617
50      141.623
51      2016-08-25 13:03:36  0.030 sec  3  0.00705735
52      140.355
53      2016-08-25 13:03:36  0.030 sec  4  0.00122272
54      140.162
55      2016-08-25 13:03:36  0.031 sec  5  0.000263918
56      140.072
57      2016-08-25 13:03:36  0.031 sec  6  0.000306555
58      140.026

```

## Principal Components Analysis (PCA)

To map a set of variables onto a subspace using linear transformations, use `H2OPrincipalComponentAnalysisEstimator`. This is the first step in Principal Components Regression.

```

1 In [25]: from h2o.estimators.pca import
          H2OPrincipalComponentAnalysisEstimator
2
3 In [26]: pca_decomp = H2OPrincipalComponentAnalysisEstimator(k=2, transform="
          NONE", pca_method="Power", impute_missing=True)
4
5 In [27]: pca_decomp.train(x=range(0,4), training_frame=iris_df)
6
7 pca Model Build Progress: [#####] 100%
8
9 In [28]: pca_decomp
10 Out[28]: Model Details
11 =====
12 H2OPCA : Principal Component Analysis
13 Model Key: PCA_model_python_1446220160417_10
14
15 ModelMetricsPCA: pca
16 ** Reported on train data. **
17
18 MSE: NaN
19 RMSE: NaN
20
21 Scoring History from Power SVD:
22 timestamp          duration    iterations    err
23 -----
24 2018-01-18 08:35:44  0.002 sec    0          29.6462    1
25 2018-01-18 08:35:44  0.002 sec    1          0.733806    1
26 2018-01-18 08:35:44  0.002 sec    2          0.0249718    1
27 2018-01-18 08:35:44  0.002 sec    3          0.000851969    1
28 2018-01-18 08:35:44  0.002 sec    4          2.90753e-05    1
29 2018-01-18 08:35:44  0.002 sec    5          1.3487e-06    1
30 2018-01-18 08:35:44  0.002 sec    6          nan          1
31 2018-01-18 08:35:44  0.003 sec    7          1.02322      2
32 2018-01-18 08:35:44  0.003 sec    8          0.0445794    2
33 2018-01-18 08:35:44  0.003 sec    9          0.00164307   2
34 2018-01-18 08:35:44  0.003 sec   10          6.27379e-05   2
35 2018-01-18 08:35:44  0.003 sec   11          2.40329e-06   2
36 2018-01-18 08:35:44  0.003 sec   12          9.88431e-08   2
37 2018-01-18 08:35:44  0.003 sec   13          nan          2
38 <bound method H2OPrincipalComponentAnalysisEstimator.train of >
39
40 In [29]: pred = pca_decomp.predict(iris_df)
41
42 pca prediction progress: [#####] 100%
43
44 In [30]: pred.head() # Projection results
45 Out[30]:
46      PC1      PC2
47 -----
48 -5.9122  2.30344
49 -5.57208 1.97383
50 -5.44648 2.09653
51 -5.43602 1.87168

```



```

52 -5.87507 2.32935
53 -6.47699 2.32553
54 -5.51543 2.07156
55 -5.85042 2.14948
56 -5.15851 1.77643
57 -5.64458 1.99191

```

## Grid Search

H2O supports grid search across hyperparameters:

```

1 In [32]: ntrees_opt = [5, 10, 15]
2
3 In [33]: max_depth_opt = [2, 3, 4]
4
5 In [34]: learn_rate_opt = [0.1, 0.2]
6
7 In [35]: hyper_parameters = {"ntrees": ntrees_opt, "max_depth":max_depth_opt,
8                               "learn_rate":learn_rate_opt}
9
10 In [36]: from h2o.grid.grid_search import H2OGridSearch
11
12 In [37]: gs = H2OGridSearch(H2OGradientBoostingEstimator(distribution="
13                               multinomial"), hyper_params=hyper_parameters)
14
15 In [38]: gs.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1, training_frame
16                               =iris_df, nfolds=10)
17
18 gbm Grid Build Progress: [#####] 100%
19
20 In [39]: print gs.sort_by('logloss', increasing=True)
21
22 Grid Search Results:
23 Model Id                                Hyperparameters: ['learn_rate', 'ntrees', '
24     max_depth']      logloss
25 -----
26 -----
27 Grid_GBM_model_1446220160417_30  ['0.2, 15, 4']
28                                0.05105
29 Grid_GBM_model_1446220160417_27  ['0.2, 15, 3']
30                                0.0551088
31 Grid_GBM_model_1446220160417_24  ['0.2, 15, 2']
32                                0.0697714
33 Grid_GBM_model_1446220160417_29  ['0.2, 10, 4']
34                                0.103064
35 Grid_GBM_model_1446220160417_26  ['0.2, 10, 3']
36                                0.106232
37 Grid_GBM_model_1446220160417_23  ['0.2, 10, 2']
38                                0.120161
39 Grid_GBM_model_1446220160417_21  ['0.1, 15, 4']
40                                0.170086
41 Grid_GBM_model_1446220160417_18  ['0.1, 15, 3']
42                                0.171218
43 Grid_GBM_model_1446220160417_15  ['0.1, 15, 2']
44                                0.181186
45 Grid_GBM_model_1446220160417_28  ['0.2, 5, 4']
46                                0.275788
47 Grid_GBM_model_1446220160417_25  ['0.2, 5, 3']
48                                0.27708

```

33	Grid_GBM_model_1446220160417_22	['0.2, 5, 2']	0.280413
34	Grid_GBM_model_1446220160417_20	['0.1, 10, 4']	0.28759
35	Grid_GBM_model_1446220160417_17	['0.1, 10, 3']	0.288293
36	Grid_GBM_model_1446220160417_14	['0.1, 10, 2']	0.292993
37	Grid_GBM_model_1446220160417_16	['0.1, 5, 3']	0.520591
38	Grid_GBM_model_1446220160417_19	['0.1, 5, 4']	0.520697
39	Grid_GBM_model_1446220160417_13	['0.1, 5, 2']	0.524777

## 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.

## Pipelines

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

```

1 In [41]: from h2o.transforms.preprocessing import H2OScaler
2
3 In [42]: from sklearn.pipeline import Pipeline
4
5 In [44]: # Turn off h2o progress bars
6
7 In [45]: h2o.__PROGRESS_BAR__=False
8
9 In [46]: h2o.no_progress()
10
11 In [47]: # build transformation pipeline using sklearn's Pipeline and H2O
           transforms
12
13 In [48]: pipeline = Pipeline([("standardize", H2OScaler()),
14     ....:                      ("pca", H2OPrincipalComponentAnalysisEstimator(k=2)
15     ....:                      ),
16     ....:                      ("gbm", H2OGradientBoostingEstimator(distribution="
           multinomial"))])
17
18 In [49]: pipeline.fit(iris_df[:4],iris_df[4])
19 Out[49]: Model Details
20 =====
21 H2OPCA : Principal Component Analysis
22 Model Key: PCA_model_python_1446220160417_32
23
24 Importance of components:
25 -----
           pc1          pc2
-----

```



```

76      2016-08-25 13:50:21 0.110 sec 3.0 0.492902188607
77      0.679995476522 0.04
77      2016-08-25 13:50:21 0.123 sec 4.0 0.446151758168
78      0.591313596193 0.04
78      --- --- --- --- --- ---
79      2016-08-25 13:50:21 0.419 sec 46.0 0.0489303232171
80      0.0192767805328 0.0
80      2016-08-25 13:50:21 0.424 sec 47.0 0.0462779490149
81      0.0180720396825 0.0
81      2016-08-25 13:50:21 0.429 sec 48.0 0.0444689238255
82      0.0171428314531 0.0
82      2016-08-25 13:50:21 0.434 sec 49.0 0.0423442541538
83      0.0161938230172 0.0
83      2016-08-25 13:50:21 0.438 sec 50.0 0.0403480417561
84      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', )])

## Randomized Grid Search

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

```

1  In [57]: from sklearn.model_selection import RandomizedSearchCV
2
3  In [58]: from h2o.cross_validation import H2OKFold
4
5  In [59]: from h2o.model.regression import h2o_r2_score
6
7  In [60]: from sklearn.metrics.scorer import make_scorer
8
9  # Parameters to test
10 In [61]: params = {"standardize__center": [True, False],
11     ....:           "standardize__scale": [True, False],
12     ....:           "pca__k": [2,3],
13     ....:           "gbm__ntrees": [10,20],
14     ....:           "gbm__max_depth": [1,2,3],
15     ....:           "gbm__learn_rate": [0.1,0.2]}
16
17 In [62]: custom_cv = H2OKFold(iris_df, n_folds=5, seed=42)
18
19 In [63]: pipeline = Pipeline([("standardize", H2OScaler()),
20     ....:                       ("pca", H2OPrincipalComponentAnalysisEstimator(
21     ....:                           k=2)),
22     ....:                       ("gbm", H2OGradientBoostingEstimator(
23     ....:                           distribution="gaussian"))])
24
25 In [64]: random_search = RandomizedSearchCV(pipeline, params,
26     ....:                                     n_iter=5,
27     ....:                                     scoring=make_scorer(h2o_r2_score),
28     ....:                                     cv=custom_cv,

```

```

27     ....:                                     random_state=42,
28     ....:                                     n_jobs=1)
29 In [65]: random_search.fit(iris_df[1:], iris_df[0])
30 Out [65]:
31 RandomizedSearchCV(cv=<h2o.cross_validation.H2OKFold instance at 0x10ba413d0
32     >,
33     error_score='raise',
34     estimator=Pipeline(steps=[('standardize', <h2o.transforms.
35     preprocessing.H2OScaler object at 0x10c0f18d0>), ('pca', ), ('
36     gbm', )]),
37     fit_params={}, iid=True, n_iter=5, n_jobs=1,
38     param_distributions={'pca__k': [2, 3], 'gbm__ntrees': [10, 20], '
39     standardize__scale': [True, False], 'gbm__max_depth': [1, 2,
40     3], 'standardize__center': [True, False], 'gbm__learn_rate':
41     [0.1, 0.2]}},
42     pre_dispatch='2*n_jobs', random_state=42, refit=True,
43     scoring=make_scorer(h2o_r2_score), verbose=0)
44
45 In [66]: print random_search.best_estimator_
46 Model Details
47 =====
48 H2OPCA : Principal Component Analysis
49 Model Key: PCA_model_python_1446220160417_136
50
51 Importance of components:
52
53 ----- pc1 ----- pc2 ----- pc3 -----
54 Standard deviation 9.6974 0.091905 0.031356
55 Proportion of Variance 0.9999 8.98098e-05 1.04541e-05
56 Cumulative Proportion 0.9999 0.99999 1
57
58 ModelMetricsPCA: pca
59 ** Reported on train data. **
60
61 MSE: NaN
62 RMSE: NaN
63 Model Details
64 =====
65 H2OGradientBoostingEstimator : Gradient Boosting Machine
66 Model Key: GBM_model_python_1446220160417_138
67
68 Model Summary:
69
70 number_of_trees number_of_internal_trees model_size_in_bytes
71 min_depth max_depth mean_depth min_leaves max_leaves
72 mean_leaves
73
74 -----
75
76 -----
77
78 20 20 2958 3
79
80 6.85 3 3 5 8
81
82
83 ModelMetricsRegression: gbm
84 ** Reported on train data. **
85
86 RMSE: 0.193906262445
87 MAE: 0.155086582663
88 RMSLE: NaN
89 Mean Residual Deviance: 0.0375996386155
90 Scoring History:
91

```

77	timestamp	duration	number_of_trees	training_rmse
78	training_mse	training_deviance		
79	2016-08-25 13:58:15	0.000 sec	0.0	0.683404046309
80	0.569341466973	0.467041090512		
81	2016-08-25 13:58:15	0.002 sec	1.0	0.571086656306
82	0.469106400643	0.326139969011		
83	2016-08-25 13:58:15	0.003 sec	2.0	0.483508601652
84	0.395952082872	0.233780567872		
85	2016-08-25 13:58:15	0.004 sec	3.0	0.414549015095
86	0.339981133963	0.171850885916		
87	2016-08-25 13:58:15	0.005 sec	4.0	0.362852508373
88	0.298212416346	0.131661942833		
89	2016-08-25 13:58:15	0.017 sec	16.0	0.204549491682
90	0.164292158112	0.0418404945473		
91	2016-08-25 13:58:15	0.018 sec	17.0	0.201762323368
92	0.162030458841	0.0407080351307		
93	2016-08-25 13:58:15	0.019 sec	18.0	0.199709571992
94	0.160735480674	0.0398839131454		
95	2016-08-25 13:58:15	0.019 sec	19.0	0.196739590066
96	0.158067452484	0.0387064662994		
97	2016-08-25 13:58:15	0.020 sec	20.0	0.193906262445
	0.155086582663	0.0375996386155		
91	Variable Importances:			
92	variable	relative_importance	scaled_importance	percentage
93	-----	-----	-----	-----
94	PC1	160.092	1	0.894701
95	PC3	14.8175	0.0925562	0.08281
96	PC2	4.0241	0.0251361	0.0224893
97	Pipeline(steps=[('standardize', <h2o.transforms.preprocessing.H2OScaler object at 0x10c1679d0>), ('pca', ), ('gbm', )])			

# 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.

# References

H2O.ai Team. **H2O website**, 2021. URL <http://h2o.ai>

H2O.ai Team. **H2O documentation**, 2021. URL <http://docs.h2o.ai>

H2O.ai Team. **H2O Python Documentation**, 2015. URL [http://h2o-release.s3.amazonaws.com/h2o/latest\\_stable\\_Pydoc.html](http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Pydoc.html)

H2O.ai Team. **H2O GitHub Repository**, 2021. URL <https://github.com/h2oai>

H2O.ai Team. **H2O Datasets**, 2021. URL <http://data.h2o.ai>

H2O.ai Team. **H2O JIRA**, 2021. URL <https://jira.h2o.ai>

H2O.ai Team. **H2Ostream**, 2021. URL <https://groups.google.com/d/forum/h2ostream>