

MACHINE LEARNING WITH PYTHON AND H2O

Spencer Aiello, Cliff Click, Hank Roark & Ludi Rehak

Edited by: Jessica Lanford



pythonTM

```
> 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/>

November 2015: First 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

©2015 H2O.ai, Inc. All Rights Reserved.

November 2015: First 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

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	17
4.7	Using Date and Time Data	19
4.8	Categoricals	19
4.9	Loading and Saving Data	21
5	Machine Learning	21
5.1	Modeling	21
5.1.1	Supervised Learning	22
5.1.2	Unsupervised Learning	23
5.2	Running Models	23
5.2.1	Gradient Boosting Models (GBM)	23
5.2.2	Generalized Linear Models (GLM)	27
5.2.3	K-means	30
5.2.4	Principal Components Analysis (PCA)	31
5.3	Grid Search	31
5.4	Integration with scikit-learn	33
5.4.1	Pipelines	33
5.4.2	Randomized Grid Search	35
6	References	38
7	Authors	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., and Rehak, L. (Nov. 2015) *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 3.5.0.99999.)

3.1 Installation in Python

To load a recent H2O package from PyPI, run:

```
1 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")

```

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:

```

1 In [1]: import h2o
2
3 In [2]: h2o.init()
4
5
6 No instance found at ip and port: localhost:54321. Trying to start local jar
7     ...
8
9 JVM stdout: /var/folders/wg/3qx1qchx1jsfjqqbzmz3stj7c0000gn/T/tmpof5ZIZ/
   h2o_hank_started_from_python.out
10 JVM stderr: /var/folders/wg/3qx1qchx1jsfjqqbzmz3stj7c0000gn/T/tmpk4uayp/
   h2o_hank_started_from_python.err
11 Using ice_root: /var/folders/wg/3qx1qchx1jsfjqqbzmz3stj7c0000gn/T/tmpKylWmt
12
13
14 Java Version: java version "1.8.0_40"
15 Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
16 Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

```



```

17
18
19 Starting H2O JVM and connecting: ..... Connection sucessful!
20 -----
21 H2O cluster uptime:          1 seconds 591 milliseconds
22 H2O cluster version:        3.2.0.5
23 H2O cluster name:           H2O_started_from_python
24 H2O cluster total nodes:    1
25 H2O cluster total memory:    3.56 GB
26 H2O cluster total cores:     4
27 H2O cluster allowed cores:   4
28 H2O cluster healthy:        True
29 H2O Connection ip:          127.0.0.1
30 H2O Connection port:        54321
31 -----

```

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

```

1 2 foo    two -1.200572  0.970428    two
2 3 bar    three -1.066722 -0.311055   other

```

To create an H2OFrame object from a Python tuple:

```

1 In [3]: df = h2o.H2OFrame(((1, 2, 3),
2 ...:                       ('a', 'b', 'c'),
3 ...:                       (0.1, 0.2, 0.3)))
4
5 Parse Progress: [#####] 100%
6 Uploaded py9bccf8ce-c01e-40c8-bc73-b8e7e0b17c6a into cluster with 3 rows and
  3 cols
7
8 In [4]: df
9 Out[4]: H2OFrame with 3 rows and 3 columns:
10
11    C1  C2  C3
12 ---  ---  ---
13  1  a    0.1
14  2  b    0.2
   3  c    0.3

```

To create an H2OFrame object from a Python list:

```

1 In [5]: df = h2o.H2OFrame([[1, 2, 3],
2 ...:                       ['a', 'b', 'c'],
3 ...:                       [0.1, 0.2, 0.3]])
4
5 Parse Progress: [#####] 100%
6 Uploaded py2c9ccb17-a86e-47d7-bela-a7950b338870 into cluster with 3 rows and
  3 cols
7
8 In [6]: df
9 Out[6]: H2OFrame with 3 rows and 3 columns:
10
11    C1  C2  C3
12 ---  ---  ---
13  1  a    0.1
14  2  b    0.2
   3  c    0.3

```

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

```

1 In [7]: df = h2o.H2OFrame({'A': [1, 2, 3],
2   ...:                     'B': ['a', 'b', 'c'],
3   ...:                     'C': [0.1, 0.2, 0.3]})
4
5 Parse Progress: [#####] 100%
6 Uploaded py2714e8a2-67c7-45a3-9d47-247120c5d931 into cluster with 3 rows and
   3 cols
7
8 In [8]: df
9 Out[8]: H2OFrame with 3 rows and 3 columns:
10  A      C  B
11  ---  ---  ---
12  1    0.1  a
13  2    0.2  b
14  3    0.3  c

```

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

```

1 In [14]: df2 = h2o.H2OFrame.from_python({'A': [1, 2, 3],
2   ....:                                  'B': ['a', 'a', 'b'],
3   ....:                                  'C': ['hello', 'all', 'world'],
4   ....:                                  'D': ['12MAR2015:11:00:00', '13
5   ....:      MAR2015:12:00:00', '14MAR2015:13:00:00']},
6   ....:                                  column_types=['numeric', 'enum', '
7   ....:      string', 'time'])
8
9 Parse Progress: [#####] 100%
10 Uploaded pyl7ealf6d-ae83-451d-ad33-89e770061601 into cluster with 3 rows and
   4 cols
11
12 In [10]: df2
13 Out[10]: H2OFrame with 3 rows and 4 columns:
14  A      C  B      D
15  ---  ---  --  -----
16  1  hello  a  2015-03-12 11:00:00
17  2   all  a  2015-03-13 12:00:00
18  3 world  b  2015-03-14 13:00:00

```

To display the column types:

```

1 In [11]: df2.types
2 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:

```

1 In [16]: import numpy as np
2
3 In [17]: df = h2o.H2OFrame.from_python(np.random.randn(4,100).tolist(),
4     column_names=list('ABCD'))
5
6 Parse Progress: [#####] 100%
7 Uploaded py0a4d1d8d-7d04-438a-a97f-a9521f802366 into cluster with 100 rows
8 and 4 cols
9
10 In [18]: df.head()
11 H2OFrame with 100 rows and 4 columns:
12
13      A      B      C      D
14 -----
15 -0.613035 -0.425327 -1.92774 -2.1201
16 -1.26552 -0.241526 -0.0445104 1.90628
17 0.763851 0.0391609 -0.500049 0.355561
18 -1.24842 0.912686 -0.61146 1.94607
19 2.1058 -1.83995 0.453875 -1.69911
20 1.7635 0.573736 -0.309663 -1.51131
21 -0.781973 0.051883 -0.403075 0.569406
22 1.40085 1.91999 0.514212 -1.47146
23 -0.746025 -0.632182 1.27455 -1.35006
24 -1.12065 0.374212 0.232229 -0.602646
25
26 In [19]: df.tail(5)
27 H2OFrame with 100 rows and 4 columns:
28
29      A      B      C      D
30 -----
31 1.00098 -1.43183 -0.322068 0.374401
32 1.16553 -1.23383 -1.71742 1.01035
33 -1.62351 -1.13907 2.1242 -0.275453
34 -0.479005 -0.0048988 0.224583 0.219037
35 -0.74103 1.13485 0.732951 1.70306

```

To display the column names:

```

1 In [20]: df.columns
2 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:

```

1 In [21]: df.describe()
2 Rows: 100 Cols: 4
3
4 Chunk compression summary:
5 chunk_type      chunkname      count      count_%      size      size_%
6 -----
7 64-bit Reals      C8D      4      100      3.4 KB      100
8
9 Frame distribution summary:
10 size      #_rows      #_chunks_per_col      #_chunks
11 -----
12 127.0.0.1:54321      3.4 KB      100      1      4
13 mean      3.4 KB      100      1      4

```

```

14 min          3.4 KB  100      1          4
15 max          3.4 KB  100      1          4
16 stddev       0 B    0        0          0
17 total        3.4 KB  100      1          4
18
19 Column-by-Column Summary: (floats truncated)
20
21 -----
22      A          B          C          D
23 type    real    real    real    real
24 mins   -2.49822 -2.37446 -2.45977 -3.48247
25 maxs    2.59380  1.91998  3.13014  2.39057
26 mean   -0.01062 -0.23159  0.11423 -0.16228
27 sigma    1.04354  0.90576  0.96133  1.02608
28 zero_count  0        0        0        0
29 missing_count 0        0        0        0

```

4.2 Selection

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

```

1 In [23]: df['A']
2 Out[23]: H2OFrame with 100 rows and 1 columns:
3      A
4 0 -0.613035
5 1 -1.265520
6 2  0.763851
7 3 -1.248425
8 4  2.105805
9 5  1.763502
10 6 -0.781973
11 7  1.400853
12 8 -0.746025
13 9 -1.120648

```

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

```

1 In [24]: df[1]
2 Out[24]: H2OFrame with 100 rows and 1 columns:
3      B
4 0 -0.425327
5 1 -0.241526
6 2  0.039161
7 3  0.912686
8 4 -1.839950
9 5  0.573736
10 6  0.051883
11 7  1.919987
12 8 -0.632182
13 9  0.374212

```

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

```

1 In [25]: df[['B','C']]
2 Out[25]: H2OFrame with 100 rows and 2 columns:
3           B           C
4 0 -0.425327 -1.927737
5 1 -0.241526 -0.044510
6 2  0.039161 -0.500049
7 3  0.912686 -0.611460
8 4 -1.839950  0.453875
9 5  0.573736 -0.309663
10 6  0.051883 -0.403075
11 7  1.919987  0.514212
12 8 -0.632182  1.274552
13 9  0.374212  0.232229

```

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

```

1 In [26]: df[0:2]
2 Out[26]: H2OFrame with 100 rows and 2 columns:
3           A           B
4 0 -0.613035 -0.425327
5 1 -1.265520 -0.241526
6 2  0.763851  0.039161
7 3 -1.248425  0.912686
8 4  2.105805 -1.839950
9 5  1.763502  0.573736
10 6 -0.781973  0.051883
11 7  1.400853  1.919987
12 8 -0.746025 -0.632182
13 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:

```

1 In [27]: df[2:7, :]
2 Out[27]: H2OFrame with 5 rows and 4 columns:
3           A           B           C           D
4 0  0.763851  0.039161 -0.500049  0.355561
5 1 -1.248425  0.912686 -0.611460  1.946068
6 2  2.105805 -1.839950  0.453875 -1.699112
7 3  1.763502  0.573736 -0.309663 -1.511314
8 4 -0.781973  0.051883 -0.403075  0.569406

```

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

```

1 In [28]: df2[ df2["B"] == "a", :]
2 Out[28]: H2OFrame with 2 rows and 4 columns:
3           A           C           B           D
4 0 1  hello  a 2015-03-12 11:00:00
5 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:

```

1 In [46]: 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'])
9
10 Parse Progress: [#####] 100%
11 Uploaded py9fdee149-dce2-4ace-91d8-e14e0d0c306a into cluster with 5 rows and
    4 cols
12
13 In [47]: df3
14 Out[47]: H2OFrame with 5 rows and 4 columns:
15      A      C      B      D
16 0  1  hello    a  1.426183e+12
17 1  2    all    a             NaN

```

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

```

1 3 NaN      NaN      NaN      NaN
2 4 NaN      NaN      NaN  1.426363e+12
3
4 In [49]: df3["A"].isna()
5 Out[49]: H2OFrame with 5 rows and 1 columns:
6      C1
7 0      0
8 1      0

```

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

```

1
2 In [41]: df3[ df3["A"].isna(), "A"] = 5
3
4 In [52]: df3
5 Out[52]: H2OFrame with 5 rows and 4 columns:
6      A      C      B      D
7 0  1  hello    a  1.426183e+12
8 1  2    all    a             NaN

```

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

```

1 3 5 NaN NaN NaN
2 4 5 NaN NaN 1.426363e+12
3
4 In [53]: df3.isna()
5 Out[53]: H2OFrame with 5 rows and 4 columns:
6      C1  C2  C3  C4
7      0  0  0  0  0
8      1  0  0  0  1

```

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:

```

1 3 0 1 0 1
2 4 0 1 0 0
3
4 In [60]: df3 = h2o.H2OFrame.from_python(
5     {'A': [1, 2, 3, None, ''],
6      'B': ['a', 'a', 'b', 'NA', 'NA'],
7      'C': ['hello', 'all', 'world', None, None],
8      'D': ['12MAR2015:11:00:00', None,
9           '13MAR2015:12:00:00', None,
10          '14MAR2015:13:00:00']},
11     column_types=['numeric', 'enum', 'string', 'time'])

```

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

```

1 Uploaded py560240ff-1668-4445-8ae1-538a0b3daf53 into cluster with 5 rows and
  4 cols
2
3 In [61]: df4.mean(na_rm=True)
4 Out[61]: [2.0, u'NaN', u'NaN', u'NaN']

```

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 In [64]: df4["A"].mean(na_rm=True)
2 Out[64]: [2.0]
3
4 In [5]: df5 = h2o.H2OFrame.from_python(
5     np.random.randn(4,100).tolist(),
6     column_names=list('ABCD'))
7 Parse Progress: [#####] 100%

```

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

```

1
2 In [6]: df5.apply(lambda x: x.mean(na_rm=True))
3 Out[6]: H2OFrame with 1 rows and 4 columns:
4         A         B         C         D
5 0  0.020849 -0.052978 -0.037272 -0.01664
6
7 In [26]: df5.apply(lambda row: sum(row), axis=1)
8 Out[26]: H2OFrame with 100 rows and 1 columns:
9         C1
10 0  0.906854
11 1  0.790760
12 2 -0.217604
13 3 -0.978141

```

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

```

1 5 -2.420732
2 6 0.875716
3 7 -1.077747
4 8 2.321706
5 9 -0.700436
6
7 In [49]: df6 = h2o.H2OFrame(
8         np.random.randint(0, 7, size=100).tolist())
9
10 Parse Progress: [#####] 100%
11 Uploaded py5b584604-73ff-4037-9618-c53122cd0343 into cluster with 100 rows
    and 1 cols
12
13 In [50]: df6.hist(plot=False)
14
15 Parse Progress: [#####] 100%
16 Uploaded py8a993d29-e354-44cf-b10e-d97aa6fd74 into cluster with 8 rows and
    1 cols
17 Out[50]: H2OFrame with 8 rows and 5 columns:
18 breaks counts mids_true mids density
19 0      0.75    NaN      NaN    0.000000

```

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 2      2.25      6      0.5  1.875  0.070000
2 3      3.00     17      1.0  2.625  0.198333
3 4      3.75      0      0.0  3.375  0.000000
4 5      4.50     16      1.5  4.125  0.186667
5 6      5.25     19      2.0  4.875  0.221667
6 7      6.00     32      2.5  5.625  0.373333
7
8 In [62]: df7 = h2o.H2OFrame.from_python(
9         ['Hello', 'World', 'Welcome', 'To', 'H2O', 'World'])
10
11 Parse Progress: [#####] 100%
12 Uploaded py95985523-6984-4f61-bd9e-c436aa8c8004 into cluster with 6 rows and
    1 cols
13

```



```
14 In [63]: df7
15 Out[63]: H2OFrame with 6 rows and 1 columns:
16      C1
17 0   Hello
18 1   World
19 2  Welcome
20 3      To
21 4    H2O
22 5   World
23
24 In [65]: df7.countmatches('l')
```

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

```
1      C1
2 0    2
3 1    1
4 2    1
5 3    0
6 4    0
7 5    1
8
9 In [89]: df7.sub('l','x')
```

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

To split strings based on a regular expression:

```
1      C1
2 0   Hexlo
3 1   Worxd
4 2  Wexcome
5 3      To
6 4    H2O
7 5   Worxd
8
9 In [86]: df7.strsplit('(l)+')
```

4.5 Merging

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

```
1      C1      C2
2 0   He      o
3 1  Wor      d
4 2   We  come
5 3   To    NaN
6 4  H2O    NaN
7 5  Wor      d
8
9 In [98]: df8 = h2o.H2OFrame.from_python(np.random.randn(100,4).tolist(),
      column_names=list('ABCD'))
```

```

10
11 Parse Progress: [#####] 100%
12 Uploaded py9607f2cc-087a-4d99-ba9f-917ca852clf2 into cluster with 100 rows
    and 4 cols
13
14 In [99]: df9 = h2o.H2OFrame.from_python(
15         np.random.randn(100,4).tolist(),
16         column_names=list('ABCD'))
17
18 Parse Progress: [#####] 100%
19 Uploaded pycb8b3aba-77d6-4383-88dd-4729f1f2c314 into cluster with 100 rows
    and 4 cols
20
21 In [100]: df8.rbind(df9)
22 Out[100]: H2OFrame with 200 rows and 4 columns:
23           A          B          C          D

```

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:

```

1         'A': ['Hello', 'World',
2              'Welcome', 'To',
3              'H2O', 'World'],
4         'n': [0,1,2,3,4,5]} )
5
6 Parse Progress: [#####] 100%
7 Uploaded py57e84cb6-ce29-4d13-afe4-4333b2186c72 into cluster with 6 rows and
    2 cols
8
9 In [109]: df11 = h2o.H2OFrame.from_python(np.random.randint(0, 10, size=100).
    tolist9), column_names=['n'])
10
11 Parse Progress: [#####] 100%
12 Uploaded py090fa929-b434-43c0-81bd-b9c61b553a31 into cluster with 100 rows
    and 1 cols
13
14 In [112]: df11.merge(df10)
15 Out[112]: H2OFrame with 100 rows and 2 columns:
16     n      A
17 0  7   NaN
18 1  3    To
19 2  0 Hello
20 3  9   NaN
21 4  9   NaN
22 5  3    To
23 6  4   H2O
24 7  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:

```

1  9  4    H2O
2
3  In [123]: df12 = h2o.H2OFrame(
4      {'A' : ['foo', 'bar', 'foo', 'bar',
5      'foo', 'bar', 'foo', 'foo'],
6      'B' : ['one', 'one', 'two', 'three',
7      'two', 'two', 'one', 'three'],
8      'C' : np.random.randn(8),
9      'D' : np.random.randn(8)})
10
11 Parse Progress: [#####] 100%
12 Uploaded pyd297bab5-4e4e-4a89-9b85-f8fecf37f264 into cluster with 8 rows and
   4 cols
13
14 In [124]: df12
15 Out[124]: H2OFrame with 8 rows and 4 columns:
16
17   A      C      B      D
18 0  foo  1.583908  one -0.441779
19 1  bar  1.055763  one  1.733467
20 2  foo -1.200572  two  0.970428
21 3  bar -1.066722 three -0.311055
22 4  foo -0.023385  two  0.077905
23 5  bar  0.758202  two  0.521504
24 6  foo  0.098259  one -1.391587
25 7  foo  0.412450 three -0.050374
26
27 In [125]: df12.group_by('A').sum().frame
28 Out[125]: H2OFrame with 2 rows and 4 columns:

```

To group by multiple columns and then apply a function:

```

1  0  bar  0.747244      3  1.943915
2  1  foo  0.870661      5 -0.835406
3
4  In [127]: df13 = df12.group_by(['A', 'B']).sum().frame
5
6  In [128]: df13
7  Out[128]: H2OFrame with 6 rows and 4 columns:
8
9      A      B      sum_C      sum_D
10 0  bar  one  1.055763  1.733467
11 1  bar  two  0.758202  0.521504
12 2  foo three  0.412450 -0.050374

```

To join the results into the original H2OFrame:

```

1  4  foo  two -1.223957  1.048333
2  5  bar three -1.066722 -0.311055
3
4  In [129]: df12.merge(df13)
5  Out[129]: H2OFrame with 8 rows and 6 columns:
6
7      A      B      C      D      sum_C      sum_D
8 0  foo  one  1.583908 -0.441779  1.682168 -1.833366
9 1  bar  one  1.055763  1.733467  1.055763  1.733467
10 2  foo  two -1.200572  0.970428 -1.223957  1.048333
11 3  bar three -1.066722 -0.311055 -1.066722 -0.311055
12 4  foo  two -0.023385  0.077905 -1.223957  1.048333

```

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:

```

1 6 foo one 0.098259 -1.391587 1.682168 -1.833366
2 7 foo three 0.412450 -0.050374 0.412450 -0.050374
3
4 In [140]: df14 = h2o.H2OFrame.from_python(
5           {'D': ['18OCT2015:11:00:00',
6                 '19OCT2015:12:00:00',
7                 '20OCT2015:13:00:00']},
8           column_types=['time'])

```

To display the day of the month:

```

1 Parse Progress: [#####] 100%
2 Uploaded py60bef051-8017-49cf-af57-2ed6d68db6d0 into cluster with 3 rows and
  1 cols
3
4 In [141]: df14.types
5 Out[141]: {u'D': u'time'}

```

To display the day of the week:

```

1 Out[142]: H2OFrame with 3 rows and 1 columns:
2      D
3 0 18
4 1 19
5 2 20

```

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:

```

1 Out[143]: H2OFrame with 3 rows and 1 columns:
2      D

```

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

```

1 1 Mon
2 2 Tue

```

To view the categorical levels in a single column:

```
1 In [145]: df12.types
2 Out[145]: {u'A': u'Enum', u'B': u'Enum',
```

To create categorical interaction features:

```
1
2 In [148]: df12.anyfactor()
3 Out[148]: True
4
5 In [149]: df12["A"].levels()
6 Out[149]: ['bar', 'foo']
7
8 In [163]: df12.interaction(['A','B'], pairwise=False, max_factors=3,
9           min_occurrence=1)
10 Interactions Progress: [#####] 100%
11 Out[163]: H2OFrame with 8 rows and 1 columns:
12     A_B
13 0  foo_one
```

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 2  foo_two
2 3  other
3 4  foo_two
4 5  other
5 6  foo_one
6 7  other
7
8 In [168]: bb_df = df12.interaction(['B','B'], pairwise=False, max_factors=2,
9           min_occurrence=1)
10 Interactions Progress: [#####] 100%
11
12 In [169]: bb_df
13 Out[169]: H2OFrame with 8 rows and 1 columns:
14     B_B
15 0  one
```

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

```
1 2  two
2 3  other
3 4  two
4 5  two
5 6  one
6 7  other
7
8 In [170]: df15 = df12.cbind(bb_df)
9
10 In [171]: df15
11 Out[171]: H2OFrame with 8 rows and 5 columns:
12     A      B      C      D      B_B
13 0  foo  one  1.583908 -0.441779  one
```

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:

```
1 6  foo    one  0.098259 -1.391587    one
2 7  foo  three  0.412450 -0.050374  other
```

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

```
1
2 In[2]: h2o.init(ip="123.45.67.89", port=54321)
```

To save an H2OFrame on the machine running H2O:

```
1
2 ##### Saving and loading files section
```

To save an H2OFrame on the machine running Python:

```
1
2 In[172]: df = h2o.upload_file("/pathToFile/fileName")
```

5 Machine Learning

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. The following list describes some common model types and features.

5.1.1 Supervised Learning

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

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

Gradient Boosting (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.

```

1 In [1]: import h2o
2
3 In [2]: h2o.init()
4
5 Java Version: java version "1.8.0_40"
6 Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
7 Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
8
9
10 Starting H2O JVM and connecting: ..... Connection successful!
11 -----
12 H2O cluster uptime:      1 seconds 738 milliseconds
13 H2O cluster version:    3.5.0.3238
14 H2O cluster name:       H2O_started_from_python
15 H2O cluster total nodes: 1

```



```

16 H2O cluster total memory: 3.56 GB
17 H2O cluster total cores: 4
18 H2O cluster allowed cores: 4
19 H2O cluster healthy: True
20 H2O Connection ip: 127.0.0.1
21 H2O Connection port: 54321
22 -----
23
24 In [3]: from h2o.estimators.gbm import H2OGradientBoostingEstimator
25
26 In [4]: iris_data_path = h2o.system_file("iris.csv") # load demonstration
        data
27
28 In [5]: iris_df = h2o.import_file(path=iris_data_path)
29
30 Parse Progress: [#####] 100%
31 Imported /Users/hank/PythonEnvs/h2obleeding/bin/../h2o_data/iris.csv. Parsed
    150 rows and 5 cols
32
33 In [6]: iris_df.describe()
34 Rows:150 Cols:5
35
36 Chunk compression summary:
37 chunktype chunkname count count_% size size_%
38 -----
39 1-Byte Int C1 1 20 218B 18.890
40 1-Byte Flt C2 4 80 936B 81.109
41
42 Frame distribution summary:
43 size rows chunks/col chunks
44 -----
45 127.0.0.1:54321 1.1KB 150 1 5
46 mean 1.1KB 150 1 5
47 min 1.1KB 150 1 5
48 max 1.1KB 150 1 5
49 stddev 0 B 0 0 0
50 total 1.1 KB 150 1 5
51
52 In [7]: gbm_regressor = H2OGradientBoostingEstimator(distribution="gaussian",
        ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")
53
54 In [8]: gbm_regressor.train(x=range(1,iris_df.ncol), y=0, training_frame=
        iris_df)
55
56 gbm Model Build Progress: [#####] 100%
57
58 In [9]: gbm_regressor
59 Out[9]: Model Details
60 =====
61 H2OGradientBoostingEstimator: Gradient Boosting Machine
62 Model Key: GBM_model_python_1446220160417_2
63
64 Model Summary:
65 number_of_trees | 10
66 model_size_in_bytes | 1535
67 min_depth | 3
68 max_depth | 3
69 mean_depth | 3
70 min_leaves | 7
71 max_leaves | 8
72 mean_leaves | 7.8
73

```

```

74 ModelMetricsRegression: gbm
75 ** Reported on train data. **
76
77 MSE: 0.0706936802293
78 R^2: 0.896209989184
79 Mean Residual Deviance: 0.0706936802293
80
81 Scoring History:
82      timestamp      duration      number_of_trees      training_MSE
83      -----
84      2015-10-30 08:50:00  0.121 sec  1      0.472445
85      2015-10-30 08:50:00  0.151 sec  2      0.334868
86      2015-10-30 08:50:00  0.162 sec  3      0.242847
87      2015-10-30 08:50:00  0.175 sec  4      0.184128
88      2015-10-30 08:50:00  0.187 sec  5      0.14365
89      2015-10-30 08:50:00  0.197 sec  6      0.116814
90      2015-10-30 08:50:00  0.208 sec  7      0.0992098
91      2015-10-30 08:50:00  0.219 sec  8      0.0864125
92      2015-10-30 08:50:00  0.229 sec  9      0.077629
93      2015-10-30 08:50:00  0.238 sec  10     0.0706937
94
95 Variable Importances:
96 variable      relative_importance      scaled_importance      percentage
97 -----

```

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

```

1  C2      15.1912      0.0667563      0.0597268
2  C5      9.50362      0.0417627      0.037365
3  C4      2.08799      0.00917544     0.00820926
4
5  In [10]: gbm_classifier = H2OGradientBoostingEstimator(distribution="
6             multinomial", ntrees=10, max_depth=3, min_rows=2, learn_rate="0.2")
7
8  In [11]: gbm_classifier.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1,
9             training_frame=iris_df)
10
11 gbm Model Build Progress: [#
12     #####] 100%
13
14 In [12]: gbm_classifier
15 Out[12]: Model Details
16 =====
17 H2OGradientBoostingEstimator : Gradient Boosting Machine
18 Model Key: GBM_model_python_1446220160417_4
19
20 Model Summary:

```

```
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: 0.00976685294679
28 R^2: 0.98534972058
29 LogLoss: 0.0782480971236
30
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
46
47 Scoring History:
48      timestamp      duration      number_of_trees      training_MSE
49      training_logloss      training_classification_error
50      -----
51      2015-10-30 08:51:52      0.047 sec      1      0.282326
52      0.758411      0.0266667
53      2015-10-30 08:51:52      0.068 sec      2      0.179214
54      0.550506      0.0266667
55      2015-10-30 08:51:52      0.086 sec      3      0.114954
56      0.412173      0.0266667
57      2015-10-30 08:51:52      0.100 sec      4      0.0744726
58      0.313539      0.02
59      2015-10-30 08:51:52      0.112 sec      5      0.0498319
60      0.243514      0.02
61      2015-10-30 08:51:52      0.131 sec      6      0.0340885
62      0.19091      0.00666667
63      2015-10-30 08:51:52      0.143 sec      7      0.0241071
64      0.151394      0.00666667
65      2015-10-30 08:51:52      0.153 sec      8      0.017606
66      0.120882      0.00666667
67      2015-10-30 08:51:52      0.165 sec      9      0.0131024
68      0.0975897      0.00666667
69      2015-10-30 08:51:52      0.180 sec      10      0.00976685
70      0.0782481      0.00666667
71
72 Variable Importances:
73 variable      relative_importance      scaled_importance      percentage
74 -----
```

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.

```

1 C3          54.0381          0.280338          0.217086
2 C1          1.35271         0.00701757         0.00543422
3 C2          0.773032         0.00401032         0.00310549
4
5 In [13]: from h2o.estimators.glm import H2OGeneralizedLinearEstimator
6
7 In [14]: prostate_data_path = h2o.system_file("prostate.csv")
8
9 In [15]: prostate_df = h2o.import_file(path=prostate_data_path)
10
11 Parse Progress: [#####] 100%
12 Imported /Users/hank/PythonEnvs/h2obleeding/bin/../h2o_data/prostate.csv.
   Parsed 380 rows and 9 cols
13
14 In [16]: prostate_df["RACE"] = prostate_df["RACE"].asfactor()
15
16 In [17]: prostate_df.describe()
17 Rows:380 Cols:9
18
19 Chunk compression summary:
20 chunk_type  chunk_name          count  count_percentage  size
   size_percentage
21 -----
22 CBS          Bits              1         11.1111         118 B
   1.39381
23 C1N          1-Byte Integers (w/o NAs)  5         55.5556         2.2 KB
   26.4588

```

```
24 C2          2-Byte Integers          1          11.1111          828 B
25   9.7803
25 CUD          Unique Reals            1          11.1111          2.1 KB
26   25.6556
26 C8D          64-bit Reals            1          11.1111          3.0 KB
27   36.7116
27
28 Frame distribution summary:
29           size  number_of_rows  number_of_chunks_per_column
30           -----  -----  -----
31           -----  -----  -----
31 127.0.0.1:54321  8.3 KB  380          1          9
32 mean           8.3 KB  380          1          9
33 min           8.3 KB  380          1          9
34 max           8.3 KB  380          1          9
35 stddev         0 B    0          0          0
36 total          8.3 KB  380          1          9
37
38
39
40 In [18]: glm_classifier = H2OGeneralizedLinearEstimator(family="binomial",
41               nfold=10, alpha=0.5)
42
43 In [19]: glm_classifier.train(x=["AGE", "RACE", "PSA", "DCAPS"], y="CAPSULE",
44               training_frame=prostate_df)
45
46 glm Model Build Progress: [#
47               #####] 100%
48
49 In [20]: glm_classifier
50 Out[20]: Model Details
51
52 =====
53 H2OGeneralizedLinearEstimator : Generalized Linear Model
54 Model Key: GLM_model_python_1446220160417_6
55
56 GLM Model: summary
57
58           family  link  regularization
59           number_of_predictors_total  number_of_active_predictors
60           number_of_iterations  training_frame
61
62 --  -----  -----  -----
63
64 binomial  logit  Elastic Net (alpha = 0.5, lambda = 3.251E-4 )  6
65                                     6                                     6
66                                     py_3
67
68 ModelMetricsBinomialGLM: glm
69 ** Reported on train data. **
70
71 MSE: 0.202434568594
72 R^2: 0.158344081513
73 LogLoss: 0.59112610879
74 Null degrees of freedom: 379
75 Residual degrees of freedom: 374
76 Null deviance: 512.288840185
77 Residual deviance: 449.25584268
78 AIC: 461.25584268
79 AUC: 0.719098211972
80 Gini: 0.438196423944
```

```

72
73 Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.28443600654:
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                0.284436      0.617512    273
85 max f2                0.199001      0.77823     360
86 max f0point5          0.415159      0.636672    108
87 max accuracy          0.415159      0.705263    108
88 max precision         0.998619      1           0
89 max absolute_MCC      0.415159      0.369123    108
90 max min_per_class_accuracy 0.33266      0.656388    175
91
92 ModelMetricsBinomialGLM: glm
93 ** Reported on cross-validation data. **
94
95 MSE: 0.209974707772
96 R^2: 0.126994679038
97 LogLoss: 0.609520995116
98 Null degrees of freedom: 379
99 Residual degrees of freedom: 373
100 Null deviance: 515.693473211
101 Residual deviance: 463.235956288
102 AIC: 477.235956288
103 AUC: 0.686706400622
104 Gini: 0.373412801244
105
106 Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.326752491231:
107      0      1      Error      Rate
108 -----
109 0     135     92    0.4053    (92.0/227.0)
110 1      48     105    0.3137    (48.0/153.0)
111 Total 183     197    0.3684    (140.0/380.0)
112
113 Maximum Metrics: Maximum metrics at their respective thresholds
114
115 metric                threshold      value      idx
116 -----
117 max f1                0.326752      0.6         196
118 max f2                0.234718      0.774359    361
119 max f0point5          0.405529      0.632378    109
120 max accuracy          0.405529      0.702632    109
121 max precision         0.999294      1           0
122 max absolute_MCC      0.405529      0.363357    109
123 max min_per_class_accuracy 0.336043      0.627451    176
124
125 Scoring History:
126      timestamp                duration      iteration      log_likelihood      objective
127 -----
128      2015-10-30 08:53:01    0.000 sec      0                256.482            0.674952
129      2015-10-30 08:53:01    0.004 sec      1                226.784            0.597118

```

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.

```
1 2015-10-30 08:53:01 0.005 sec 3 224.629 0.59158
2 2015-10-30 08:53:01 0.005 sec 4 224.628 0.591579
3 2015-10-30 08:53:01 0.006 sec 5 224.628 0.591579
4
5 In [21]: from h2o.estimators.kmeans import H2OKMeansEstimator
6
7 In [22]: cluster_estimator = H2OKMeansEstimator(k=3)
8
9 In [23]: cluster_estimator.train(x=[0,1,2,3], training_frame=iris_df)
10
11 kmeans Model Build Progress: [#
12 #####] 100%
13
14 In [24]: cluster_estimator
15 Out[24]: Model Details
16 =====
17 H2OKMeansEstimator : K-means
18 Model Key: K-means_model_python_1446220160417_8
19
20 Model Summary:
21
22 number_of_rows    number_of_clusters    number_of_categorical_columns
23 number_of_iterations    within_cluster_sum_of_squares
24 total_sum_of_squares    between_cluster_sum_of_squares
25 -----
26
27 150                3                0
28 4                190.757                596
29 405.243
30
31 ModelMetricsClustering: kmeans
32 ** Reported on train data. **
33
34 MSE: NaN
35 Total Within Cluster Sum of Square Error: 190.756926265
36 Total Sum of Square Error to Grand Mean: 596.0
37 Between Cluster Sum of Square Error: 405.243073735
38
39 Centroid Statistics:
40
41 centroid    size    within_cluster_sum_of_squares
42 -----
43 1            96    149.733
44 2            32    17.292
45 3            22    23.7318
46
47 Scoring History:
48
49 timestamp    duration    iteration    avg_change_of_std_centroids
50 within_cluster_sum_of_squares
51 -----
```

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.

```

1      2015-10-30 08:54:39  0.047 sec  1      2.09788
2                                191.282
3      2015-10-30 08:54:39  0.049 sec  2      0.00316006
4                                190.82
5      2015-10-30 08:54:39  0.050 sec  3      0.000846952
6                                190.757
7
8 In [25]: from h2o.transforms.decomposition import H2OPCA
9
10 In [26]: pca_decomp = H2OPCA(k=2, transform="NONE", pca_method="Power")
11
12 In [27]: pca_decomp.train(x=range(0,4), training_frame=iris_df)
13
14 pca Model Build Progress: [#
15     #####] 100%
16
17 In [28]: pca_decomp
18 Out[28]: Model Details
19 =====
20 H2OPCA : Principal Component Analysis
21 Model Key: PCA_model_python_1446220160417_10
22
23 Importance of components:
24
25      pc1      pc2
26 -----
27 Standard deviation  7.86058  1.45192
28 Proportion of Variance  0.96543  0.032938
29 Cumulative Proportion  0.96543  0.998368
30
31 ModelMetricsPCA: pca
32 ** Reported on train data. **
33
34 MSE: NaN
35
36 In [29]: pred = pca_decomp.predict(iris_df)
37
38 In [30]: pred.head() # Projection results
39 Out[30]:
40      PC1      PC2
41 -----
42 5.9122  2.30344
43 5.57208  1.97383
44 5.44648  2.09653
45 5.43602  1.87168
46 5.87507  2.32935
47 6.47699  2.32553

```

5.3 Grid Search

H2O supports grid search across hyperparameters:


```
1 5.85042 2.14948
2 5.15851 1.77643
3 5.64458 1.99191
4
5 In [32]: ntrees_opt = [5, 10, 15]
6
7 In [33]: max_depth_opt = [2, 3, 4]
8
9 In [34]: learn_rate_opt = [0.1, 0.2]
10
11 In [35]: hyper_parameters = {"ntrees": ntrees_opt, "max_depth":max_depth_opt,
12     "learn_rate":learn_rate_opt}
13
14 In [36]: from h2o.grid.grid_search import H2OGridSearch
15
16 In [37]: gs = H2OGridSearch(H2OGradientBoostingEstimator(distribution="
17     multinomial"), hyper_params=hyper_parameters)
18
19 In [38]: gs.train(x=range(0,iris_df.ncol-1), y=iris_df.ncol-1, training_frame
20     =iris_df, nfold=10)
21
22 gbm Grid Build Progress: [#####]
23     100%
24
25 In [39]: print gs.sort_by('logloss', increasing=True)
26
27 Grid Search Results:
28 Model Id Hyperparameters: ['learn_rate', 'ntrees', '
29     max_depth'] logloss
30 -----
31
32 GBM_model_1446220160417_30 ['0.2, 15, 4']
33                                     0.05105
34 GBM_model_1446220160417_27 ['0.2, 15, 3']
35                                     0.0551088
36 GBM_model_1446220160417_24 ['0.2, 15, 2']
37                                     0.0697714
38 GBM_model_1446220160417_29 ['0.2, 10, 4']
39                                     0.103064
40 GBM_model_1446220160417_26 ['0.2, 10, 3']
41                                     0.106232
42 GBM_model_1446220160417_23 ['0.2, 10, 2']
43                                     0.120161
44 GBM_model_1446220160417_21 ['0.1, 15, 4']
45                                     0.170086
46 GBM_model_1446220160417_18 ['0.1, 15, 3']
47                                     0.171218
48 GBM_model_1446220160417_15 ['0.1, 15, 2']
49                                     0.181186
50 GBM_model_1446220160417_28 ['0.2, 5, 4']
51                                     0.275788
52 GBM_model_1446220160417_25 ['0.2, 5, 3']
53                                     0.27708
54 GBM_model_1446220160417_22 ['0.2, 5, 2']
55                                     0.280413
56 GBM_model_1446220160417_20 ['0.1, 10, 4']
57                                     0.28759
58 GBM_model_1446220160417_17 ['0.1, 10, 3']
59                                     0.288293
```

5.4 Integration with scikit-learn

The H2O Python client can be used within scikit-learn pipelines and cross validation searches. This extends the power of both H2O and scikit-learn.

5.4.1 Pipelines

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

```

1 GBM_model_1446220160417_16  ['0.1, 5, 3']
2                               0.520591
3 GBM_model_1446220160417_19  ['0.1, 5, 4']
4                               0.520697
5 GBM_model_1446220160417_13  ['0.1, 5, 2']
6                               0.524777
7
8 In [41]: from h2o.transforms.preprocessing import H2OScaler
9
10 In [42]: from sklearn.pipeline import Pipeline
11
12 In [43]: # Turn off h2o progress bars
13
14 In [44]: h2o.__PROGRESS_BAR__=False
15
16 In [45]: h2o.no_progress()
17
18 In [46]: # build transformation pipeline using sklearn's Pipeline and H2O
19           transforms
20
21 In [47]: pipeline = Pipeline([("standardize", H2OScaler()),
22     ....:                      ("pca", H2OPCA(k=2)),
23     ....:                      ("gbm", H2OGradientBoostingEstimator(distribution="
24                               multinomial"))])
25
26 In [48]: pipeline.fit(iris_df[:4],iris_df[4])
27 Out[48]: Model Details
28 =====
29 H2OPCA : Principal Component Analysis
30 Model Key: PCA_model_python_1446220160417_32
31
32 Importance of components:
33
34 -----
35                pc1                pc2
36 -----
37 Standard deviation    3.22082    0.34891
38 Proportion of Variance 0.984534    0.0115538
39 Cumulative Proportion 0.984534    0.996088
40
41
42 ModelMetricsPCA: pca
43 ** Reported on train data. **
44
45 MSE: NaN
46 Model Details
47 =====
48 H2OGradientBoostingEstimator : Gradient Boosting Machine
49 Model Key: GBM_model_python_1446220160417_34
50
51 Model Summary:

```

```
45      number_of_trees      model_size_in_bytes      min_depth      max_depth
46      mean_depth      min_leaves      max_leaves      mean_leaves
47      -----
48      150      27014      1      5      4.84
49      2      13      9.99333
50
51 ModelMetricsMultinomial: gbm
52 ** Reported on train data. **
53
54 MSE: 0.00162796438754
55 R^2: 0.997558053419
56 LogLoss: 0.0152718654494
57
58 Confusion Matrix: vertical: actual; across: predicted
59
60 Iris-setosa      Iris-versicolor      Iris-virginica      Error      Rate
61 -----
62 50      0      0      0      0 / 50
63 0      50      0      0      0 / 50
64 0      0      50      0      0 / 50
65 50      50      50      0      0 / 150
66
67 Top-3 Hit Ratios:
68 k      hit_ratio
69 --- -----
70 1      1
71 2      1
72 3      1
73
74 Scoring History:
75      timestamp      duration      number_of_trees      training_MSE
76      training_logloss      training_classification_error
77 -----
78 2015-10-30 09:00:31 0.007 sec 1.0      0.36363226261
79      0.924249463924      0.04
80 2015-10-30 09:00:31 0.011 sec 2.0      0.297174376838
81      0.788619346614      0.04
82 2015-10-30 09:00:31 0.014 sec 3.0      0.242952566898
83      0.679995475248      0.04
84 2015-10-30 09:00:31 0.017 sec 4.0      0.199051390695
85      0.591313594921      0.04
86 2015-10-30 09:00:31 0.021 sec 5.0      0.163730865044
87      0.517916553872      0.04
88 --- --- --- ---
89 2015-10-30 09:00:31 0.191 sec 46.0      0.00239417625265
90      0.0192767794713      0.0
91 2015-10-30 09:00:31 0.195 sec 47.0      0.00214164838414
92      0.0180720391174      0.0
93 2015-10-30 09:00:31 0.198 sec 48.0      0.00197748500569
94      0.0171428309311      0.0
95 2015-10-30 09:00:31 0.202 sec 49.0      0.00179303578037
96      0.0161938228014      0.0
97 2015-10-30 09:00:31 0.205 sec 50.0      0.00162796438754
98      0.0152718654494      0.0
99
100 Variable Importances:
101 variable      relative_importance      scaled_importance      percentage
```

5.4.2 Randomized Grid Search

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

```

1  PC1          448.958          1          0.982184
2  PC2          8.1438          0.0181393      0.0178162
3  Pipeline(steps=[('standardize', <h2o.transforms.preprocessing.H2OScaler
   object at 0x1085cec90>), ('pca', ), ('gbm', )])
4
5  In [57]: from sklearn.grid_search import RandomizedSearchCV
6
7  In [58]: from h2o.cross_validation import H2OKFold
8
9  In [59]: from h2o.model.regression import h2o_r2_score
10
11 In [60]: from sklearn.metrics.scorer import make_scorer
12
13 In [61]: from sklearn.metrics.scorer import make_scorer
14
15 In [62]: params = {"standardize__center":    [True, False],          #
   Parameters to test
16     ....:         "standardize__scale":    [True, False],
17     ....:         "pca__k":                [2,3],
18     ....:         "gbm__ntrees":           [10,20],
19     ....:         "gbm__max_depth":        [1,2,3],
20     ....:         "gbm__learn_rate":       [0.1,0.2]}
21
22 In [63]: custom_cv = H2OKFold(iris_df, n_folds=5, seed=42)
23
24 In [64]: pipeline = Pipeline([("standardize", H2OScaler()),
   ....:                        ("pca", H2OPCA(k=2)),
25     ....:                        ("gbm", H2OGradientBoostingEstimator(
26     distribution="gaussian"))])
27
28 In [65]: random_search = RandomizedSearchCV(pipeline, params,
   ....:                                     n_iter=5,
29     ....:                                     scoring=make_scorer(h2o_r2_score)
30     ,
31     ....:                                     cv=custom_cv,
32     ....:                                     random_state=42,
33     ....:                                     n_jobs=1)
34 In [66]: random_search.fit(iris_df[1:], iris_df[0])
35 Out[66]:
36 RandomizedSearchCV(cv=<h2o.cross_validation.H2OKFold instance at 0x108d59200
   >,
37     error_score='raise',
38     estimator=Pipeline(steps=[('standardize', <h2o.transforms.
   preprocessing.H2OScaler object at 0x108d50150>), ('pca', ), ('
   gbm', )]),
39     fit_params={}, iid=True, n_iter=5, n_jobs=1,
40     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]},
41     pre_dispatch='2*n_jobs', random_state=42, refit=True,
42     scoring=make_scorer(h2o_r2_score), verbose=0)
43
44 In [67]: print random_search.best_estimator_
45 Model Details
46 =====

```

```
47 H2OPCA : Principal Component Analysis
48 Model Key: PCA_model_python_1446220160417_136
49
50 Importance of components:
51 -----
52          pc1          pc2          pc3
53 -----
54 Standard deviation    3.16438    0.180179    0.143787
55 Proportion of Variance 0.994721    0.00322501    0.00205383
56 Cumulative Proportion 0.994721    0.997946    1
57
58 ModelMetricsPCA: pca
59 ** Reported on train data. **
60
61 MSE: NaN
62 Model Details
63 =====
64 H2OGradientBoostingEstimator : Gradient Boosting Machine
65 Model Key: GBM_model_python_1446220160417_138
66
67 Model Summary:
68   number_of_trees  model_size_in_bytes  min_depth  max_depth
69   mean_depth      min_leaves      max_leaves  mean_leaves
70 -----
71          20          2743          3          3          3
72          4          8          6.35
73
74 ModelMetricsRegression: gbm
75 ** Reported on train data. **
76
77 MSE: 0.0566740346323
78 R^2: 0.916793146878
79 Mean Residual Deviance: 0.0566740346323
80
81 Scoring History:
82   timestamp      duration  number_of_trees  training_MSE
83   training_deviance
84 -----
85   2015-10-30 09:04:46  0.001 sec  1          0.477453
86   0.477453
87   2015-10-30 09:04:46  0.002 sec  2          0.344635
88   0.344635
89   2015-10-30 09:04:46  0.003 sec  3          0.259176
90   0.259176
91   2015-10-30 09:04:46  0.004 sec  4          0.200125
92   0.200125
93   2015-10-30 09:04:46  0.005 sec  5          0.160051
94   0.160051
95   2015-10-30 09:04:46  0.006 sec  6          0.132315
96   0.132315
97   2015-10-30 09:04:46  0.006 sec  7          0.114554
98   0.114554
99   2015-10-30 09:04:46  0.007 sec  8          0.100317
100  0.100317
101  2015-10-30 09:04:46  0.008 sec  9          0.0890903
102  0.0890903
103  2015-10-30 09:04:46  0.009 sec  10         0.0810115
104  0.0810115
```

93	2015-10-30 09:04:46	0.009 sec	11	0.0760616
	0.0760616			
94	2015-10-30 09:04:46	0.010 sec	12	0.0725191
	0.0725191			
95	2015-10-30 09:04:46	0.011 sec	13	0.0694355
	0.0694355			
96	2015-10-30 09:04:46	0.012 sec	14	0.06741
	0.06741			
97	2015-10-30 09:04:46	0.012 sec	15	0.0655487
	0.0655487			
98	2015-10-30 09:04:46	0.013 sec	16	0.0624041
	0.0624041			
99	2015-10-30 09:04:46	0.014 sec	17	0.0615533
	0.0615533			
100	2015-10-30 09:04:46	0.015 sec	18	0.058708
	0.058708			
101	2015-10-30 09:04:46	0.015 sec	19	0.0579205
	0.0579205			
102	2015-10-30 09:04:46	0.016 sec	20	0.056674
	0.056674			

103	Variable Importances:			
104				
105	variable	relative_importance	scaled_importance	percentage
106	-----	-----	-----	-----

6 References

H2O.ai Team. **H2O website**, 2015. URL <http://h2o.ai>

H2O.ai Team. **H2O documentation**, 2015. 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**, 2015. URL <https://github.com/h2oai>

H2O.ai Team. **H2O Datasets**, 2015. URL <http://data.h2o.ai>

H2O.ai Team. **H2O JIRA**, 2015. URL <https://jira.h2o.ai>

H2O.ai Team. **H2Ostream**, 2015. URL <https://groups.google.com/d/forum/h2ostream>

7 Authors

Spencer Aiello

Spencer comes from an unconventional background. After studying Physics and Math as an undergraduate at UCSC, he came to San Francisco to continue his education, earning his MS in Analytics from USF. Spencer has worked on a number of projects related to analytics in R, Python, Java, and SQL. At H2O, he works primarily on the R front-end and the backend Java R-interpreter.

Cliff Click

Cliff Click is the CTO and Co-Founder of H2O, makers of H2O, the open-source math and machine learning engine for Big Data. Cliff is invited to speak regularly at industry and academic conferences and has published many papers about HotSpot technology. He holds a PhD in Computer Science from Rice University and about 15 patents.

Hank Roark

Hank is a Data Scientist and Hacker at H2O. Hank comes to H2O with a background turning data into products and system solutions and loves helping others find value in their data. Hank has an SM from MIT in Engineering and Management and BS Physics from Georgia Tech.

Ludi Rehak

Ludi is a hacker at H2O, where she helps make machine learning algorithms fast. She enjoys harnessing the unreasonable effectiveness of data in projects that apply data science to daily life. Ludi holds a Masters degree in Statistics from Stanford University and a Bachelors in Biology from Cornell University with a concentration in Computational Biology. Previously, she was a software engineer on the search team at Jive Software, an EMT, a plant biology researcher, and a tennis instructor.

Jessica Lanford

Jessica is a word hacker and seasoned technical communicator at H2O.ai. She brings our product to life by documenting the many features and functionality of H2O. Having worked for some of the top companies in technology including Dell, AT&T, and Lam Research, she is an expert at translating complex ideas to digestible articles.

