Machine Learning with Sparkling Water: H2O + Spark

Michal Malohlava Nidhi Mehta

Edited by: Brandon Hill & Vinod Iyengar

---

http://h2o.ai/resources

July 2016: First Edition

Machine Learning with Sparkling Water: H2O + Spark
by Michal Malohlava & Nidhi Mehta
Edited by: Brandon Hill & Vinod Iyengar

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

July 2016: First Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

1 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 Sparkling Water Introduction

Sparkling Water allows users to combine the fast, scalable machine learning algorithms of H2O with the capabilities of Spark. With Sparkling Water, users can drive computation from Scala, R, or Python and use the H2O Flow UI, providing an ideal machine learning platform for application developers.

Spark is an elegant and powerful general-purpose, open-source, in-memory platform with tremendous momentum. H2O is an in-memory application for machine learning that is reshaping how people apply math and predictive analytics to their business problems.

Integrating these two open-source environments provides a seamless experience for users who want to make a query using Spark SQL, feed the results into H2O to build a model and make predictions, and then use the results again in Spark. For any given problem, better interoperability between tools provides a better experience.

For additional examples, please visit the Sparkling Water GitHub repository at `https://github.com/h2oai/sparkling-water/tree/master/examples`.

2.1 Typical Use Cases

Sparkling Water excels in leveraging existing Spark-based workflows needed to call advanced machine learning algorithms. We identified three the most common use-cases which are described below.

2.1.1 Model Building

A typical example involves multiple data transformations with help of Spark API, where a final form of data is transformed into H2O frame and passed to

an H2O algorithm. The constructed model estimates different metrics based on the testing data or gives a prediction that can be used in the rest of the data pipeline (see Figure 1).

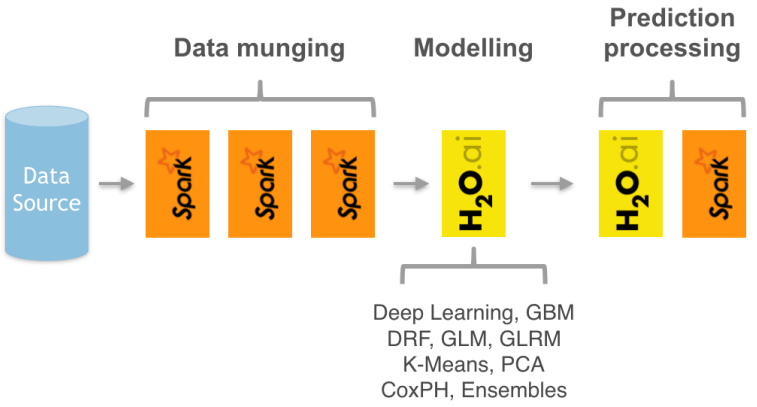


Figure 1: Sparkling Water extends existing Spark data pipeline with advanced machine learning algorithms.

2.1.2 Data Munging

Another use-case includes Sparkling Water as a provider of ad-hoc data transformations. Figure 2 shows a data pipeline benefiting from H2O's parallel data load and parse capabilities, while Spark API is used as another provider of data transformations. Furthermore, H2O can be used as in-place data transformer.

2.1.3 Stream Processing

The last use-case depicted on Figure 3 introduces two data pipelines. The first one, called an off-line training pipeline, is invoked regularly (e.g., every hour or every day), utilizes Spark as well as H2O API and provides an H2O model as output. The H2O API allows the model to be exported in a source code form. The second one processes streaming data (with help of Spark Streaming or Storm) and utilizes the model trained in the first pipeline to score the incoming data. Since the model is exported as a code, the streaming pipeline can be lightweight and independent on H2O or Sparkling Water infrastructure.

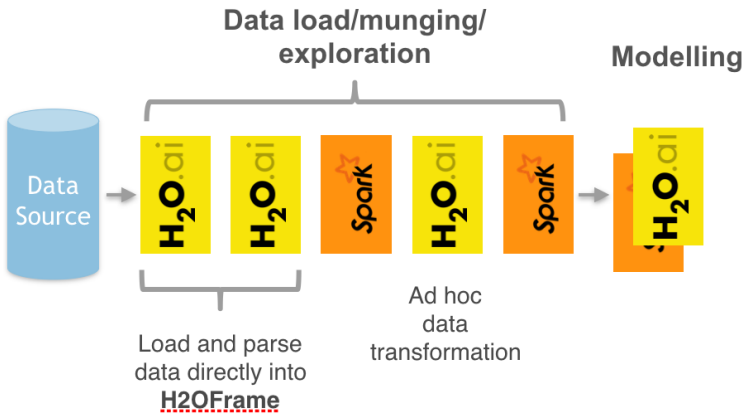


Figure 2: Sparkling Water introduces H2O parallel load and parse into Spark pipelines.

2.2 Features

Sparkling Water provides transparent integration for the H2O engine and its machine learning algorithms into the Spark platform, enabling:

- Use of H2O algorithms in Spark workflow
- Transformation between H2O and Spark data structures
- Use of Spark RDDs and DataFrames as input for H2O algorithms
- Use of H2OFrames as input for MLlib algorithms
- Transparent execution of Sparkling Water applications on top of Spark

2.3 Supported Data Sources

Currently, Sparkling Water can use the following data source types:

- Standard Resilient Distributed Dataset (RDD) API for loading data and transforming it into H2OFrames
- H2O API for loading data directly into H2OFrame from file(s) stored on:
  - local filesystems
  - HDFS
  - S3

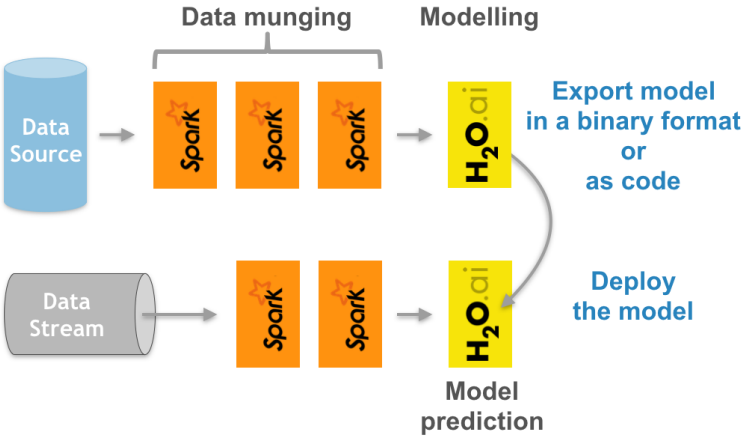


Figure 3: Sparkling Water used as an off-line model producer feeding models into a stream-based data pipeline.

  - HTTP/HTTPS

For more details, please refer to the H2O documentation at `http://docs.h2o.ai`.

2.4 Supported Data Formats

Sparkling Water can read data stored in the following formats:

- CSV
- SVMLight
- ARFF

For more details, please refer to the H2O documentation at `http://docs.h2o.ai`.

2.5 Supported Spark Execution Environments

Sparkling Water can run on top of Spark in the following ways:

- as a local cluster (where the master node is `local`, `local[*]`, or `local-cluster[...]`)

- as a standalone cluster[1]
- in a YARN environment[2]

[1]Refer to the Spark standalone documentation `http://spark.apache.org/docs/latest/spark-standalone.html`

[2]Refer to the Spark YARN documentation `http://spark.apache.org/docs/latest/running-on-yarn.html`

3 Design

Sparkling Water is designed to be executed as a regular Spark application. It provides a way to initialize H2O services on each node in the Spark cluster and access data stored in data structures of Spark and H2O.

Since Sparkling Water is primarily designed as Spark application, it is launched inside a Spark executor created after submitting the application. At this point, H2O starts services, including distributed key-value (K/V) store and memory manager, and orchestrates them into a cloud. The topology of the created cloud replicates the topology of the underlying Spark cluster.

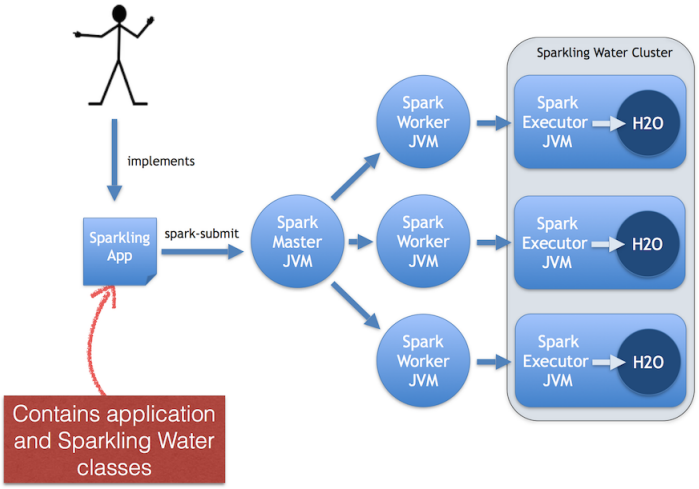


Figure 4: Sparkling Water design depicting deployment of the Sparkling Water application to the standalone Spark cluster.

3.1 Data Sharing between Spark and H2O

Sparkling Water enables transformation between different types of RDDs and H2O's `H2OFrame`, and vice versa.

When converting from an `H2OFrame` to an RDD, a wrapper is created around the `H2OFrame` to provide an RDD-like API. In this case, data is not duplicated but served directly from the underlying `H2OFrame`.

Converting from an RDD/DataFrame to an `H2OFrame` requires data duplication because it transfers data from the RDD storage into `H2OFrame`. However,

data stored in an `H2OFrame` is heavily compressed and does not need to be preserved in RDD.

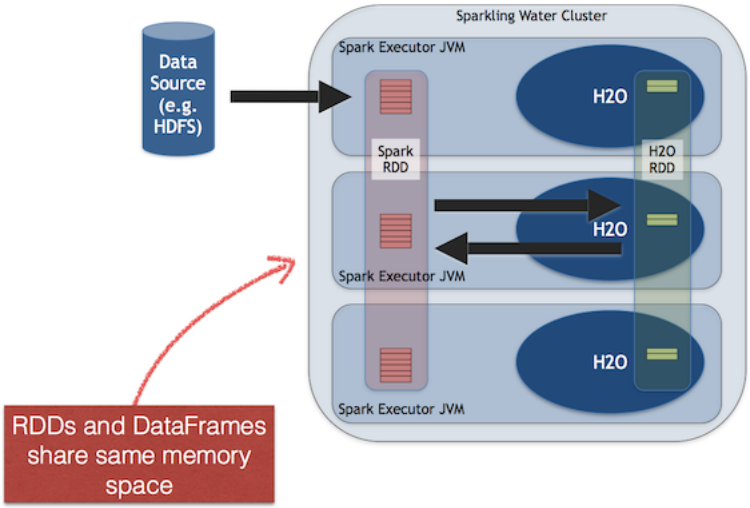


Figure 5: Sharing between Spark and H2O inside an executor JVM.

3.2 Provided Primitives

Sparkling Water provides several primitives (for more information, refer to Table 1). Before using H2O algorithms and data structures, the first step is to create and start the `H2OContext` instance using the `val hc = new H2OContext(sc).start()` call.

The `H2OContext` contains the necessary information for running H2O services and exposes methods for data transformation between the Spark RDD or `DataFrame` and the `H2OFrame`. Starting `H2OContext` involves a distributed operation that contacts all accessible Spark executor nodes and initializes H2O services (such as the key-value store and RPC) inside the executors' JVMs.

When `H2OContext` is running, H2O data structures and algorithms can be manipulated. The key data structure is `H2OFrame`, which represents a distributed table composed of vectors. A new `H2OFrame` can be created using one of the following methods:

- loading a cluster local file (a file located on each node of the cluster):

```
1 val h2oFrame = new H2OFrame(new File("/data/iris.
    csv"))
```

- loading a file from HDFS/S3/S3N/S3A:

```
1 val h2oFrame = new H2OFrame(URI.create("hdfs://
    data/iris.csv"))
```

- loading multiple files from HDFS/S3/S3N/S3A:

```
1 val h2oFrame = new H2OFrame(URI.create("hdfs://
    data/iris/01.csv"), URI.create("hdfs://data/
    iris/02.csv"))
```

- transforming Spark RDD or `DataFrame`:

```
1 val h2oFrame = h2oContext.asH2OFrame(rdd)
```

- referencing existing `H2OFrame` by its key

```
1 val h2oFrame = new H2OFrame("iris.hex")
```

Concept	API Representation	Description
H2O Context	`H2OContext`	Contains the H2O state and provides primitives to publish `RDD` as `H2OFrame` and vice versa. Follows design principles of Spark primitives such as `SparkContext` or `SQLContext`.
H2O Entry Point	`water.H2O`	Represents the entry point for accessing H2O services. Contains information about running H2O services, including a list of nodes and the status of the distributed K/V datastore.
H2O Frame	`water.fvec.H2OFrame`	A data structure representing a table of values. The table is column-based and provides column and row accessors.
H2O Algorithm	package `hex`	Represents the H2O machine learning algorithms library, including DeepLearning, GBM, GLM, DRF, and other algorithms.

Table 1: Sparkling Water primitives

When the `H2OContext` is running, any H2O algorithm can be called. Most of provided algorithms are located in the `hex` package. Calling an algorithm is composed of two steps:

- Specifying parameters:

```
1 val train: H2OFrame = new H2OFrame(new File("
     prostate.csv"))
2 val gbmParams = new GBMParameters()
3 gbmParams._train = train
4 gbmParams._response_column = 'CAPSULE
5 gbmParams._ntrees = 10
```

- Creating the model builder and launching computations. The `trainModel` method is non-blocking and returns a job representing the computation.

```
1 val gbmModel = new GBM(gbmParams).trainModel.get
```

4 Programming API

4.1 Starting H2O Services

```
val sc: SparkContext = ...
val hc = H2OContext.getOrCreate(sc)
```

or:

```
val sc: SparkContext = ...
val hc = new H2OContext(sc).start()
```

When the number of Spark nodes is known, it can be specified in the `getOrCreate` call:

```
val hc = H2OContext.getOrCreate(sc, numOfSparkNodes)
```

or, in start method of `H2OContext`:

```
val hc = new H2OContext(sc).start(numOfSparkNodes)
```

The former variant is preferred, because it initiates and starts `H2OContext` in one call and can be used to obtain already existing `H2OContext`. It is semantically the same as the latter variant though.

4.2 Memory Allocation

H2O resides in the same executor JVM as Spark. The memory provided for H2O is configured via Spark; refer to Spark configuration for more details.

Generic configuration

- Configure the Executor memory (i.e., memory available for H2O) via the Spark configuration property `spark.executor.memory`. For example, `bin/sparkling-shell --conf spark.executor.memory=5g` or configure the property in `$SPARK_HOME/conf/spark-defaults.conf`
- Configure the Driver memory (i.e., memory available for H2O client running inside the Spark driver) via the Spark configuration property `spark.driver.memory`. For example, `bin/sparkling-shell --conf spark.driver.memory=4g` or configure the property in `$SPARK_HOME/conf/spark-defaults.conf`.

Yarn specific configuration

- Refer to the Spark documentation `https://spark.apache.org/docs/latest/running-on-yarn.html`
- For JVMs that require a large amount of memory, we strongly recommend configuring the maximum amount of memory available for individual mappers.

4.3 Converting H2OFrame into RDD[T]

The `H2OContext` class provides the explicit conversion, `asRDD`, which creates an RDD-like wrapper around the provided `H2OFrame`:

```
1 def asRDD[A <: Product: TypeTag: ClassTag](fr:
    H2OFrame): RDD[A]
```

The call expects the type `A` to create a correctly-typed RDD. The conversion requires type `A` to be bound by `Product` interface. The relationship between the columns of `H2OFrame` and the attributes of class `A` is based on name matching.

Example

```
1 val df: H2OFrame = ...
2 val rdd = asRDD[Weather](df)
```

4.4 Converting H2OFrame into DataFrame

The `H2OContext` class provides the explicit conversion, `asDataFrame`, which creates a `DataFrame`-like wrapper around the provided `H2OFrame`. Technically, it provides the `RDD[sql.Row]` RDD API:

```
1 def asDataFrame(fr: H2OFrame)(implicit sqlContext:
    SQLContext): DataFrame
```

This call does not require any type of parameters, but since it creates `DataFrame` instances, it requires access to an instance of `SQLContext`. In this case, the instance is provided as an implicit parameter of the call. The parameter can be passed in two ways: as an explicit parameter or by introducing an implicit variable into the current context.

The schema of the created instance of the `DataFrame` is derived from the column name and the types of `H2OFrame` specified.

Example

Using an explicit parameter in the call to pass `sqlContext`:

```
val sqlContext = new SQLContext(sc)
val schemaRDD = asDataFrame(h2oFrame)(sqlContext)
```

or as implicit variable provided by actual environment:

```
implicit val sqlContext = new SQLContext(sc)
val schemaRDD = asDataFrame(h2oFrame)
```

4.5 Converting RDD[T] into H2OFrame

The `H2OContext` provides implicit conversion from the specified `RDD[A]` to `H2OFrame`. As with conversion in the opposite direction, the type `A` has to satisfy the upper bound expressed by the type `Product`. The conversion will create a new `H2OFrame`, transfer data from the specified RDD, and save it to the H2O K/V data store.

```
implicit def asH2OFrame[A <: Product: TypeTag](rdd:
    RDD[A]): H2OFrame
```

The API also provides explicit version which allows for specifying name for resulting `H2OFrame`.

```
def asH2OFrame[A <: Product: TypeTag](rdd: RDD[A],
    frameName: String): H2OFrame
```

Example

```
val rdd: RDD[Weather] = ...
import h2oContext._
// Implicit call of H2OContext.asH2OFrame[Weather](rdd
    ) is used
val hf: H2OFrame = rdd
// Explicit call of of H2OContext API with name for
    resulting H2OFrame
val hfNamed: H2OFrame = h2oContext.asH2OFrame(rdd, "
    hfNamed")
```

4.6 Converting DataFrame into H2OFrame

The `H2OContext` provides **implicit** conversion from the specified `DataFrame` to `H2OFrame`. The conversion will create a new `H2OFrame`, transfer data from the specified `DataFrame`, and save it to the H2O K/V data store.

```
1 implicit def asH2OFrame(rdd: DataFrame): H2OFrame
```

The API also provides explicit version which allows for specifying name for resulting `H2OFrame`.

```
1 def asH2OFrame(rdd: DataFrame, frameName: String):
    H2OFrame
```

Example

```
1 val df: DataFrame = ...
2 import h2oContext._
3 // Implicit call of H2OContext.asH2OFrame(srdd) is
    used
4 val hf: H2OFrame = df
5 // Explicit call of H2Context API with name for
    resulting H2OFrame
6 val hfNamed: H2OFrame = h2oContext.asH2OFrame(df, "
    hfNamed")
```

4.7 Creating H2OFrame from an Existing Key

If the H2O cluster already contains a loaded `H2OFrame` referenced by the key `train.hex`, it is possible to reference it from Sparkling Water by creating a proxy `H2OFrame` instance using the key as the input:

```
1 val trainHF = new H2OFrame("train.hex")
```

4.8 Type Map Between H2OFrame and Spark DataFrame Types

For all primitive Scala types or Spark SQL types (see `org.apache.spark.sql.types`) which can be part of Spark RDD/DataFrame, we provide mapping into H2O vector types (numeric, categorical, string, time, UUID - see `water.fvec.Vec`):

Scala type	SQL type	H2O type
NA	BinaryType	Numeric
Byte	ByteType	Numeric
Short	ShortType	Numeric
Integer	IntegerType	Numeric
Long	LongType	Numeric
Float	FloatType	Numeric
Double	DoubleType	Numeric
String	StringType	String
Boolean	BooleanType	Numeric
java.sql.TimeStamp	TimestampType	Time

4.9 Calling H2O Algorithms

1. Create the parameters object that holds references to input data and parameters specific for the algorithm:

```
val train: RDD = ...
val valid: H2OFrame = ...

val gbmParams = new GBMParameters()
gbmParams._train = train
gbmParams._valid = valid
gbmParams._response_column = 'bikes
gbmParams._ntrees = 500
gbmParams._max_depth = 6
```

2. Create a model builder:

```
val gbm = new GBM(gbmParams)
```

3. Invoke the model build job and block until the end of computation (`trainModel` is an asynchronous call by default):

```
val gbmModel = gbm.trainModel.get
```

4.10 Using Spark Data Sources with H2OFrame

Spark SQL provides configurable data source for SQL tables. Sparkling Water enable `H2OFrame` to be used as data source to load/save data from/to Spark SQL table.

4.10.1 Reading from `H2OFrame`

Let's suppose we have a `H2OFrame`. The shortest way to load a `DataFrame` from `H2OFrame` with default settings is:

```
1 val df = sqlContext.read.h2o(frame.key)
```

There are two more ways to load a `DataFrame` from `H2OFrame` allowing us to specify additional options:

```
1 val df = sqlContext.read.format("h2o").option("key",
    frame.key.toString).load()
```

or

```
1 val df = sqlContext.read.format("h2o").load(frame.key.
    toString)
```

4.10.2 Saving to `H2OFrame`

Let's suppose we have `DataFrame` df. The shortest way to save the `DataFrame` as `H2OFrame` with default settings is:

```
1 df.write.h2o("new_key")
```

There are two more ways to save the `DataFrame` as `H2OFrame` allowing us to specify additional options:

```
1 df.write.format("h2o").option("key","new_key").save()
```

or

```
1 df.write.format("h2o").save("new_key")
```

All three variants save the `DataFrame` as `H2OFrame` with the key "new_key". They won't succeed if a `H2OFrame` with the same key already exists.

4.10.3 Loading and Saving Options

If the key is specified as 'key' option, and also in the load/save method, the option 'key' is preferred:

```
1 val df = sqlContext.read.from("h2o").option("key","
    key_one").load("key_two")
```

or

```
1 val df = sqlContext.read.from("h2o").option("key","
    key_one").save("key_two")
```

In both examples, "key_one" is used.

4.10.4 Specifying Saving Mode

There are four save modes available when saving data using Data Source API- see `http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes`

- If "append" mode is used, an existing `H2OFrame` with the same key is deleted, and a new one created with the same key. The new frame contains the union of all rows from the original `H2OFrame` and the appended `DataFrame`.
- If "overwrite" mode is used, an existing `H2OFrame` with the same key is deleted, and new one with the new rows is created with the same key.
- If "error" mode is used, and a `H2OFrame` with the specified key already exists, an exception is thrown.
- If "ignore" mode is used, and a `H2OFrame` with the specified key already exists, no data are changed.

5 Deployment

Since Sparkling Water is designed as a regular Spark application, its deployment cycle is strictly driven by Spark deployment strategies (refer to Spark documentation[3]). Spark applications are deployed by the `spark-submit` [4] script that handles all deployment scenarios:

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --conf <key>=<value> \
  ... # other options \
  <application-jar> [application-arguments]
```

- `--class`: Name of main class with `main` method to be executed. For example, the `water.SparklingWaterDriver` application launches H2O services.
- `--master`: Location of Spark cluster
- `--conf`: Specifies any configuration property using the format `key=value`
- `application-jar`: Jar file with all classes and dependencies required for application execution
- `application-arguments`: Arguments passed to the main method of the class via the `--class` option

5.1 Referencing Sparkling Water

5.1.1 Using Fatjar

The Sparkling Water archive provided at `http://h2o.ai/download` contains a Fatjar with all classes required for Sparkling Water run.

An application submission with Sparkling Water Fatjar is using the `--jars` option which references included fatjar.

[3] Spark deployment guide `http://spark.apache.org/docs/latest/cluster-overview.html`

[4] Submitting Spark applications `http://spark.apache.org/docs/latest/submitting-applications.html`

```
$SPARK_HOME/bin/spark-submit \
  --jars assembly/build/libs/sparkling-water-assembly
     -1.6.1-all.jar \
  --class org.apache.spark.examples.h2o.
     CraigslistJobTitlesStreamingApp \
  /dev/null
```

5.1.2 Using Spark Package

Sparkling Water is also published as a Spark package. The benefit of using the package is that you can use it directly from your Spark distribution without need to download Sparkling Water.

For example, if you have Spark version 1.6 and would like to use Sparkling Water version 1.6.1 and launch example `CraigslistJobTitlesStreamingApp`, then you can use the following command:

```
$SPARK_HOME/bin/spark-submit \
  --packages ai.h2o:sparkling-water-core_2.10:1.6.1,ai
     .h2o:sparkling-water-examples_2.10:1.6.1 \
  --class org.apache.spark.examples.h2o.
     CraigslistJobTitlesStreamingApp \
  /dev/null
```

The Spark option `--packages` points to coordinate of published Sparkling Water package in Maven repository.

The similar command works for spark-shell:

```
$SPARK_HOME/bin/spark-shell \
 --packages ai.h2o:sparkling-water-core_2.10:1.6.1,ai.
    h2o:sparkling-water-examples_2.10:1.6.1
```

The same command works for Python programs:

```
$SPARK_HOME/bin/spark-submit \
 --packages ai.h2o:sparkling-water-core_2.10:1.6.1,ai.
    h2o:sparkling-water-examples_2.10:1.6.1\
  example.py
```

Note: When you are using Spark packages you do not need to download Sparkling Water distribution! Spark installation is sufficient!

5.2 Target Deployment Environments

Sparkling Water supports deployments to the following Spark cluster types:

- Local cluster
- Standalone cluster
- YARN cluster

5.2.1 Local cluster

The local cluster is identified by the following master URLs - `local`, `local[K]`, or `local[*]`. In this case, the cluster is composed of a single JVM and is created during application submission.

For example, the following command will run the ChicagoCrimeApp application inside a single JVM with a heap size of 5g:

```
1 $SPARK_HOME/bin/spark-submit \
2   --conf spark.executor.memory=5g \
3   --conf spark.driver.memory=5g \
4   --master local[*] \
5   --packages ai.h2o:sparkling-water-examples_2
       .10:1.6.1 \
6   --class org.apache.spark.examples.h2o.
       ChicagoCrimeApp \
7   /dev/null
```

5.2.2 On Standalone Cluster

For AWS deployments or local private clusters, the standalone cluster deployment[5] is typical. Additionally, a Spark standalone cluster is also provided by Hadoop distributions like CDH or HDP. The cluster is identified by the URL `spark://IP:PORT`.

The following command deploys the `ChicagoCrimeApp` on a standalone cluster where the master node is exposed on IP `machine-foo.bar.com` and port `7077`:

[5]Refer to Spark documentation `http://spark.apache.org/docs/latest/spark-standalone.html`

```
$SPARK_HOME/bin/spark-submit \
  --conf spark.executor.memory=5g \
  --conf spark.driver.memory=5g \
  --master spark://machine-foo.bar.com:7077 \
  --packages ai.h2o:sparkling-water-examples_2
     .10:1.6.1 \
  --class org.apache.spark.examples.h2o.
     ChicagoCrimeApp \
  /dev/null
```

In this case, the standalone Spark cluster must be configured to provide the requested 5g of memory per executor node.

5.2.3 On YARN Cluster

Because it provides effective resource management and control, most production environments use YARN for cluster deployment.[6] In this case, the environment must contain the shell variable `HADOOP_CONF_DIR` or `YARN_CONF_DIR` which point to Hadoop configuration directory (e.g., `/etc/hadoop/conf`).

```
$SPARK_HOME/bin/spark-submit \
  --conf spark.executor.memory=5g \
  --conf spark.driver.memory=5g \
  --num-executors 5 \
  --master yarn-client \
  --packages ai.h2o:sparkling-water-examples_2
     .10:1.6.1 \
  --class org.apache.spark.examples.h2o.
     ChicagoCrimeApp \
  /dev/null
```

The command in the example above creates a YARN job and requests for 5 nodes, each with 5G of memory. The `yarn-client` option forces driver to run in the client process.

[6] See Spark documentation `http://spark.apache.org/docs/latest/running-on-yarn.html`

5.3 Sparkling Water Configuration Properties

The following configuration properties can be passed to Spark to configure Sparking Water:

Generic parameters

Property name	Default value	Description
spark.ext.h2o.flatfile	true	Use flatfile (instead of multicast) for creating H2O cloud.
spark.ext.h2o.cluster.size	-1	Expected number of workers of H2O cloud. -1 automatically detects the cluster size. This number must be equal to number of Spark workers.
spark.ext.h2o.port.base	54321	Base port used for individual H2O node configuration.
spark.ext.h2o.port.incr	2	Increment added to base port to find the next available port.
spark.ext.h2o.cloud.timeout	60*1000	Timeout (in msec) for cloud.
spark.ext.h2o.spreadrdd.retries	10	Number of retries for creation of an RDD covering all existing Spark executors.
spark.ext.h2o.cloud.name	sparkling-water-	Name of H2O cloud.
spark.ext.h2o.network.mask	–	Subnet selector for H2O if IP detection fails. Useful for detecting correct IP if 'spark.ext.h2o.flatfile' is false.*
spark.ext.h2o.nthreads	-1	Limit for number of threads used by H2O. -1 means unlimited.
spark.ext.h2o.disable.ga	false	Disable Google Analytics tracking for embedded H2O.

H2O server node parameters

Property name	Default value	Description
spark.ext.h2o.node.log.level	INFO	Set H2O node internal logging level.
spark.ext.h2o.node.log.dir	System.getProperty ("user.dir") + File.separator + "h2ologs" or YARN container dir	Location of h2o logs on executor machine.

H2O client parameters

Property name	Default value	Description
spark.ext.h2o.client.log.level	INFO	Set H2O client internal logging level (running inside Spark driver).
spark.ext.h2o.client.log.dir	System.getProperty ("user.dir") + File.separator + "h2ologs"	Location of h2o logs on driver machine.
spark.ext.h2o.client.web.port	-1	Exact client port to access web UI. -1 triggers automatic search for free port starting at spark.ext.h2o.port.base.

6 Building a Standalone Application

Sparkling Water Example Project

This is a simple example project to start coding with Sparkling Water.

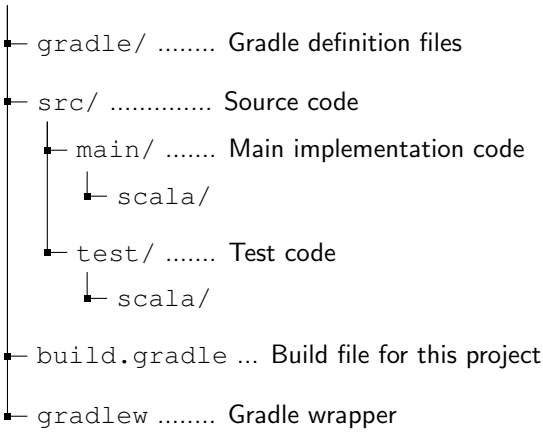
Dependencies

This droplet uses Sparkling Water 1.6 which integrates:

- Spark 1.6
- H2O 3.8 Tukey

For more details see `build.gradle`.

Project structure



Project building

For building, please, use provided gradlew command:

```
1 ./gradlew build
```

Run demo

For running a simple application:

```
1 ./gradlew run
```

Starting with IDEA

There are two ways to open this project in IntelliJ IDEA

Using Gradle build file directly:

Open the project's `build.gradle` in IDEA via File → Open

or using Gradle generated project files:

1. Generate Idea configuration files via `./gradlew idea`
2. Open project in Idea via File → Open

Note: To clean up Idea project files please launch `./gradlew cleanIdea`

Starting with Eclipse

1. Generate Eclipse project files via `./gradlew eclipse`
2. Open project in Eclipse via File → Import → Existing Projects into Workspace

Running tests

To run tests, please, run:

```
./gradlew test
```

Checking code style

To check codestyle:

```
./gradlew scalaStyle
```

Creating and Running Spark Application

Create application assembly which can be directly submitted to Spark cluster:

```
./gradlew shadowJar
```

The command creates jar file `build/libs/sparkling-water-droplet-app.ja` containing all necessary classes to run application on top of Spark cluster.

Submit application to Spark cluster (in this case, local cluster is used):

```
export MASTER='local-cluster[3,2,1024]'
$SPARK_HOME/bin/spark-submit --class water.droplets.
    SparklingWaterDroplet build/libs/sparkling-water-
    droplet-all.jar
```

7 What is PySparkling Water?

PySparkling Water is an integration of Python with Sparkling water. It allows the user to start H2O services on a spark cluster from Python API.

In the PySparkling Water driver program, the `SparkContext` (sc) uses Py4J to start the driver JVM and the JAVA `SparkContext` is used to create `H2OContext` (hc). This in turn starts the H2O cloud in the Spark ecosystem. Once the H2O cluster is up, H2O-Python package is used to interact with the cloud and run H2O algorithms. All pure H2O calls are executed via H2O's REST API interface. Users can easily integrate their regular PySpark workflow with H2O algorithms using PySparkling Water.

PySparkling Water programs can be launched as an application, or in an interactive shell, or notebook environment.

7.1 Getting Started:

1. Download Spark (if not already installed) from the Spark Downloads Page.

   Choose Spark release : 1.6.0

   Choose a package type: Pre-built for Hadoop 2.4 and later

2. Point SPARK_HOME to the existing installation of Spark and export variable MASTER.

```
export SPARK_HOME="/path/to/spark/installation"
```

   Launch a local Spark cluster with 3 worker nodes with 2 cores and 1g per node.

```
export MASTER="local-cluster[3,2,1024]"
```

3. From your terminal, run:

```
cd ~/Downloads
unzip sparkling-water-1.6.1.zip
cd sparkling-water-1.6.1
```

   Start an interactive Python terminal:

```
bin/pysparkling
```

Or start a notebook:

```
IPYTHON_OPTS="notebook" bin/pysparkling
```

4. Create an H2O cloud inside the Spark cluster and import H2O-Python package:

```
from pysparkling import *
hc= H2OContext(sc).start()
import h2o
```

5. Follow this demo (https://github.com/h2oai/h2o-world-2015-trai blob/master/tutorials/pysparkling/Chicago_Crime_Demo.ipynb), which imports Chicago crime, census and weather data and predicts the probability of arrest.

Alternatively, to launch on YARN:

```
wget http://h2o-release.s3.amazonaws.com/sparkling-water/rel-1.6/1/sparkling-water-1.6.1.zip
unzip sparkling-water-1.6.1.zip

export SPARK_HOME="/path/to/spark/installation"
export HADOOP_CONF_DIR=/etc/hadoop/conf
export SPARKLING_HOME="/path/to/SparklingWater/installation"
$SPARKLING_HOME/bin/pysparkling --num-executors 3 --executor-memory 20g --executor-cores 10 --driver-memory 20g --master yarn-client
```

Then create an H2O cloud inside the Spark cluster and import H2O-Python package:

```
from pysparkling import *
hc= H2OContext(sc).start()
import h2o
```

Or to launch as a Spark Package application:

```
$SPARK_HOME/bin/spark-submit --packages ai.h2o:sparkling-water-core_2.10:1.6.1 --py-files $SPARKLING_HOME/py/dist/pySparkling-1.6.1-py2.7.egg
$SPARKLING_HOME/py/examples/scripts/H2OContextDemo.py
```

7.2 Using Spark Data Sources

The way that a `H2OFrame` can be used as Spark's data source differs a little bit in Python from Scala.

7.2.1 Reading from `H2OFrame`

Let's suppose we have an `H2OFrame`. There are two ways how the `DataFrame` can be loaded from `H2OFrame` in pySparkling:

```
df = sqlContext.read.format("h2o").option("key",frame.
    frame_id).load()
```

or

```
df = sqlContext.read.format("h2o").load(frame.frame_id
    )
```

7.2.2 Saving to `H2OFrame`

Let's suppose we have a `DataFrame` df. There are two ways how `DataFrame` can be saved as `H2OFrame` in pySparkling:

```
df.write.format("h2o").option("key","new_key").save()
```

or

```
df.write.format("h2o").save("new_key")
```

Both variants save `DataFrame` as a `H2OFrame` with key `new_key`. They won't succeed if a `H2OFrame` with the same key already exists.

7.2.3 Loading and Saving Options

If the key is specified as 'key' option, and also in the load/save method, the option 'key' is preferred:

```
df = sqlContext.read.from("h2o").option("key","key_one
    ").load("key_two")
```

or

```
df = sqlContext.read.from("h2o").option("key","key_one
    ").save("key_two")
```

In both examples, `key_one` is used.

8 A Use Case Example

8.1 Predicting Arrival Delay in Minutes - Regression

What is the task?

As Chief Air Traffic Controller, your job is come up with a prediction engine that can be used to tell passengers whether an incoming flight will be delayed by X number of minutes. To accomplish this task, we have an airlines dataset containing ∼44k flights since 1987 with features such as: Origin and Destination codes, distance traveled, carrier, etc. The key variable we are trying to predict is 'ArrDelay' (arrival delay) in minutes. We will do this leveraging H2O and the Spark SQL library.

Spark SQL

One of the many cool features about the Spark project is the ability to initiate a SQL context within our application that enables us to write SQL-like queries against an existing `DataFrame`. Given the ubiquitous nature of SQL, this is very appealing to data scientists who may not be comfortable yet with Scala / Java / Python, but want to perform complex manipulations of their data.

Within the context of this example, we are going to first read in the airlines dataset and then process a weather file which contains the weather data at the arriving city. Joining the two tables will require a SQL context such that we can write an INNER JOIN against the two independent `DataFrame`s. Let's get started!

Data Ingest

Our first order of business is to process both files, the flight data and the weather data:

```
1  object AirlinesWithWeatherDemo extends
       SparkContextSupport {
2
3    def main(args: Array[String]): Unit = {
4      // Configure this application
5      val conf: SparkConf = configure("Sparkling Water:
           Join of Airlines with Weather Data")
6
7      // Create SparkContext to execute application on
           Spark cluster
8      val sc = new SparkContext(conf)
```

```
9     val h2oContext = H2OContext.getOrCreate(sc)
10    import h2oContext._
11    // Setup environment
12    addFiles(sc,
13      absPath("examples/smalldata/
            Chicago_Ohare_International_Airport.csv"),
14      absPath("examples/smalldata/allyears2k_headers.
            csv.gz"))
15
16    val wrawdata = sc.textFile(SparkFiles.get("
          Chicago_Ohare_International_Airport.csv"),3).
          cache()
17    val weatherTable = wrawdata.map(_.split(",")).map(
          row => WeatherParse(row)).filter(!_.isWrongRow
          ())
18
19    // Load H2O from CSV file (i.e., access directly
          H2O cloud)
20    val airlinesData = new H2OFrame(new File(
          SparkFiles.get("allyears2k_headers.csv.gz")))
21
22    val airlinesTable: RDD[Airlines] = asRDD[Airlines
          ](airlinesData)
```

The flight data file is imported directly into H2O already as an `H2OFrame`. The weather table, however, is first processed in Spark where we do some parsing of the data and data scrubbing.

After both files have been processed, we then take the airlines data that currently sits in H2O and it pass back into Spark whereby we filter for those flights ONLY arriving at Chicago's O'Hare International Airport:

```
1 val flightsToORD = airlinesTable.filter(f => f.Dest ==
      Some("ORD"))
2
3 flightsToORD.count
4 println(s"\nFlights to ORD: ${flightsToORD.count}\n")
```

At this point, we are ready to join these two tables which are currently Spark RDDs. The workflow required for this is as follows:

- Convert the RDD into a `DataFrame` and register the resulting `DataFrame` as 'TempTable'

```
val sqlContext = new SQLContext(sc)
// Import implicit conversions
import sqlContext.implicits._
flightsToORD.toDF.registerTempTable("FlightsToORD")
weatherTable.toDF.registerTempTable("WeatherORD")
```

- Join the two temp tables using Spark SQL

```
val bigTable = sqlContext.sql(
   """SELECT
     |f.Year,f.Month,f.DayofMonth,
     |f.CRSDepTime,f.CRSArrTime,f.CRSElapsedTime,
     |f.UniqueCarrier,f.FlightNum,f.TailNum,
     |f.Origin,f.Distance,
     |w.TmaxF,w.TminF,w.TmeanF,w.PrcpIn,w.SnowIn,w.CDD,w.HDD,w.GDD,
     |f.ArrDelay
     |FROM FlightsToORD f
     |JOIN WeatherORD w
     |ON f.Year=w.Year AND f.Month=w.Month AND f.DayofMonth=w.Day
     |WHERE f.ArrDelay IS NOT NULL""".stripMargin)
```

- Transfer the joined table from Spark back to H2O to run an algorithm against

```
val train: H2OFrame = bigTable
```

H2O Deep Learning

Now we have our dataset loaded into H2O. Recall this dataset has been filtered to only include the flights and weather data on Chicago Ohare. It's now time to run a machine learning algorithm to predict flight delay in minutes. As always, we start off with the necessary imports we need followed by declaring the parameters that we wish to control:

```
val dlParams = new DeepLearningParameters()
dlParams._train = train
dlParams._response_column = 'ArrDelay
dlParams._epochs = 5
dlParams._activation = Activation.RectifierWithDropout
dlParams._hidden = Array[Int](100, 100)

val dl = new DeepLearning(dlParams)
val dlModel = dl.trainModel.get
```

More parameters for Deep Learning and all other algorithms can be found in H2O documentation at `http://docs.h2o.ai` .

Now we can run this model on our test dataset to score the model against our holdout dataset:

```
val predictionH2OFrame = dlModel.score(bigTable)('
    predict)
val predictionsFromModel = asRDD[DoubleHolder](
    predictionH2OFrame).collect.map(_.result.getOrElse
    (Double.NaN))
println(predictionsFromModel.mkString("\n===> Model
    predictions: ", ", ", ", ...\n"))
```

The full source for the application is here: `http://bit.ly/1mo3XO2`

9 FAQ

Where do I find the Spark logs?

Spark logs are located in the directory `$SPARK_HOME/work/app-<AppName>` (where `<AppName>` is the name of your application).

Spark is very slow during initialization, or H2O does not form a cluster. What should I do?

Configure the Spark variable `SPARK_LOCAL_IP`. For example:

```
1 export SPARK_LOCAL_IP='127.0.0.1'
```

How do I increase the amount of memory assigned to the Spark executors in Sparkling Shell?

Sparkling Shell accepts common Spark Shell arguments. For example, to increase the amount of memory allocated by each executor, use the `spark.executor.memory` parameter: `bin/sparkling-shell --conf "spark.executor.memory=4g"`

How do I change the base port H2O uses to find available ports?

The H2O accepts spark.ext.h2o.port.base parameter via Spark configuration properties: `bin/sparkling-shell --conf "spark.ext.h2o.port.base=13431"`. For a complete list of configuration options, refer to Devel Documentation.

How do I use Sparkling Shell to launch a Scala test.script that I created?

Sparkling Shell accepts common Spark Shell arguments. To pass your script, please use `-i` option of Spark Shell: `bin/sparkling-shell -i test.script`

How do I increase PermGen size for Spark driver?

Specify `--conf spark.driver.extraJavaOptions="-XX:MaxPermSize=384m"`

How do I add Apache Spark classes to Python path?

Configure the Python path variable PYTHONPATH:

```
1 export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/
    python/build:$PYTHONPATH
2 export PYTHONPATH=$SPARK_HOME/python/lib/py4j-0.8.2.1-
    src.zip:$PYTHONPATH
```

Trying to import a class from the hex package in Sparkling Shell but getting weird error:

```
error: missing arguments for method hex in object
    functions; follow this method with '_' if you want
     to treat it as a partially applied
```

In this case you are probably using Spark 1.5 which is importing SQL functions into Spark Shell environment. Please use the following syntax to import a class from the hex package:

```
import _root_.hex.tree.gbm.GBM
```

10 References

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

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

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

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

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

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

H2O.ai Team. **H2O R Package Documentation**, 2016. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html