Machine Learning with Sparkling Water: H2O + Spark

Michal Malohlava Jakub Hava Nidhi Mehta

Edited by: Vinod Iyengar & Angela Bartz

---

http://h2o.ai/resources

April 2024: Fifth Edition

Machine Learning with Sparkling Water: H2O + Spark
by Michal Malohlava, Jakub Hava, & Nidhi Mehta
Edited by: Vinod Iyengar & Angela Bartz

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

April 2024: Fifth Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

What is H2O?

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

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

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

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

Try it out

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

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

Join the community

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

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 pre-process data using Spark, feed the results into H2O to build a model, and make predictions. Sparkling Water tries to follow Spark conventions for the H2O algorithms so the API is easy to start with for people familiar with Spark.

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

Have Questions about Sparkling Water?

- Post them on Stack Overflow using the **sparkling-water** tag at `http://stackoverflow.com/questions/tagged/sparkling-water`.
- Join the chat at `https://gitter.im/h2oai/sparkling-water`.

Typical Use Cases

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

Model Building

A typical example involves multiple data transformations with the help of Spark API, where a final form of data is transformed into an 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).

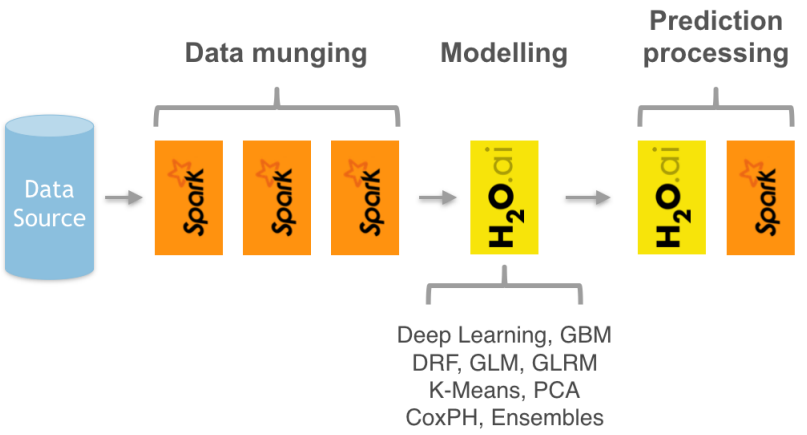


Figure 1: Sparkling Water extends existing Spark data pipeline with advanced machine learning algorithms.

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 an in-place data transformer.

Stream Processing

The last use-case depicted in 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), and utilizes both Spark and H2O API. The off-line pipeline provides an H2O model as output. The H2O API allows the model to be exported in a form independent on H2O run-time. The second pipeline 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 with

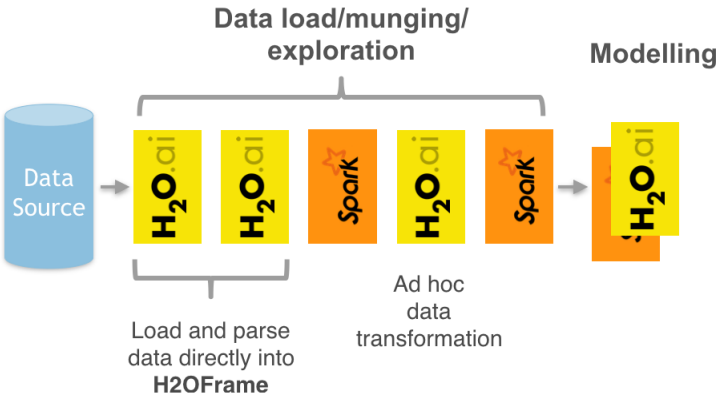


Figure 2: Sparkling Water introduces H2O parallel load and parse into Spark pipelines.

no run-time dependency on H2O, the streaming pipeline can be lightweight and independent on H2O or Sparkling Water infrastructure.

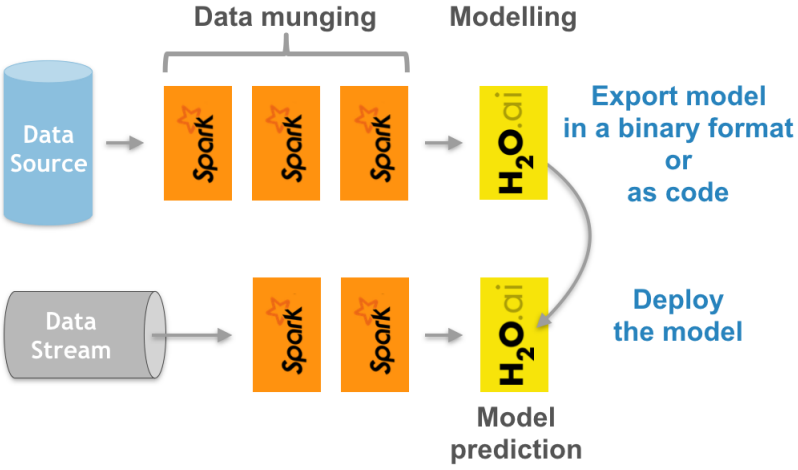


Figure 3: Sparkling Water used as an off-line model producer feeding models into a stream-based data pipeline.

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 workflows
- Transformation between H2O and Spark data structures
- Use of Spark RDDs, DataFrames, and Datasets as an input for H2O algorithms
- Transparent execution of Sparkling Water applications on top of Spark
- Use H2O algorithms in Spark pipelines
- Productionalize H2O models in Spark environment

Supported Data Sources

In Sparkling Water, you can either use Spark API or H2O to load data. This list describes all supported data sources from which Sparkling Water is able to ingest data:

- local filesystems
- HDFS
- S3
- HTTP/HTTPS
- JDBC
- Apache Hive
- DBFS (Databricks File System) when running on Databricks cluster
- Google Cloud Storage

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

Supported Data Formats

In Sparkling Water, you can decide whether to use Spark or H2O for loading the file of the specific format. When using H2O API, the following formats are supported:

- CSV (delimited) files (including GZipped CSV)
- ORC
- SVMLight
- ARFF
- XLS
- XLSX
- Avro version 1.8.0 (without multi-file parsing or column type modification)
- Parquet
- Spark RDD, Data Frame or Dataset

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

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` or `local[*]`)
- as a standalone cluster[1]
- in a YARN environment[2]

Sparkling Water Clients

Sparkling Water provides clients for R, Scala and Python languages.

- R : RSparkling
- Python : PySparkling
- Scala/Java : Sparkling Water

Whenever we show any code in this booklet, we will try to show variant in all three supported languages when possible.

[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`

Sparkling Water Requirements

Sparkling Water supports Spark 2.3, 2.4 (except Spark 2.4.2) and 3.0. In specific examples of this booklet we refer to artifacts for Spark 3.0 and use Sparkling Water 3.30.0.7, however, the Sparkling Water code is consistent across versions for each Spark.

- Linux/OS X/Windows
- Java 8 or higher
- Python 3.6+ For Python version of Sparkling Water (PySparkling)
- R 3.4+ for R version of Sparkling Water (RSparkling)
- Installed Spark and have SPARK_HOME environmental variable pointing to its home.

Design

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

Sparkling Water supports two types of backends - internal and external. In the internal backend, Sparkling Water starts H2O in Spark executors, which are created after application submission. At this point, H2O starts services, including distributed key-value (K/V) store and memory manager, and orchestrates the nodes into a cloud. The topology of the created cloud tries to match the topology of the underlying Spark cluster. The following figure represents the Internal Sparkling Water cluster.

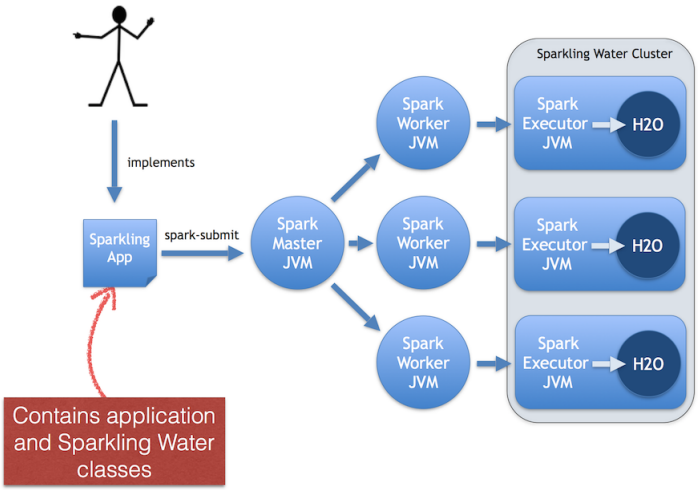


Figure 4: Sparkling Water design depicting deployment of the Sparkling Water in internal backend to the Spark cluster.

In the external backend, the H2O cluster is started separately and is connected to from the Spark driver. The following figure represents the External Sparkling Water cluster.

More information about the backends and how to start Sparkling Water in each backend is available in the next section.

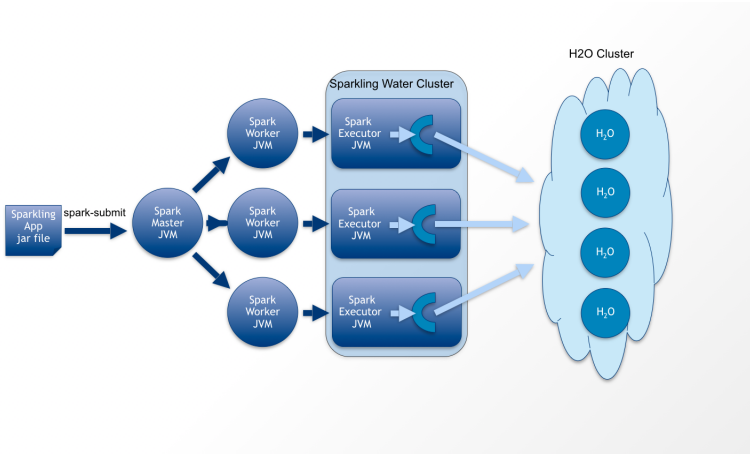


Figure 5: Sparkling Water design depicting deployment of the Sparkling Water in internal backend to the standalone Spark cluster.

Data Sharing between Spark and H2O

Sparkling Water enables transformation between Spark data structures (RDD, DataFrame, Dataset) and H2O's `H2OFrame`, and vice versa.

When converting a `H2OFrame` to an RDD or DataFrame, a wrapper is created around the `H2OFrame` to provide an RDD or DataFrame 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 a `H2OFrame`. However, data stored in a `H2OFrame` is heavily compressed and does not need to be preserved in RDD after the conversion.

The following figure shows how data is accessed when running in the internal backend mode of Sparkling Water.

In the external backend, the Spark and H2O data spaces are separated, however the separation is transparent to the user.

H2OContext

The main Sparkling Water component is `H2O Context`. H2O Context holds state and provides primitives to transfer RDD/DataFrames/Datasets into

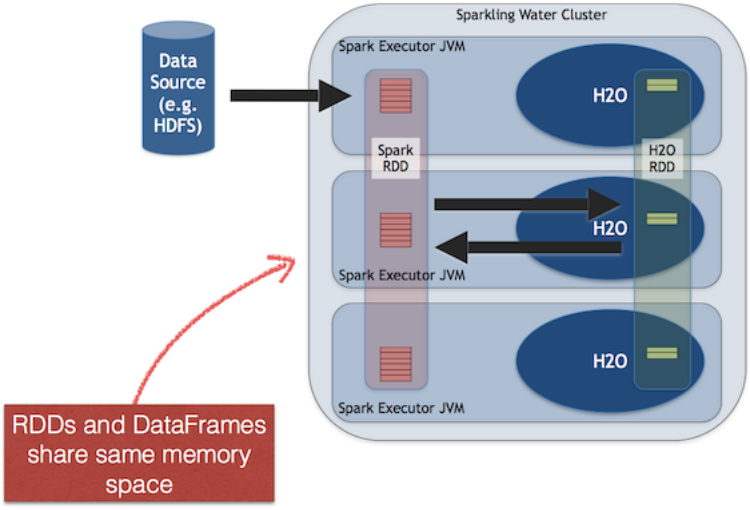


Figure 6: Sharing between Spark and H2O inside an executor JVM in Internal backend.

H2OFrames and vice versa. It follows design principles of Spark primitives such as `SparkSession`, `SparkContext` and `SQLContext`.

`H2OContext` contains the necessary information for running H2O services and exposes methods for data transformation between the Spark RDD, `DataFrame` or `Dataset`, and the `H2OFrame`. Starting `H2OContext` involves an operation that:

- In case of the internal backend, is distributed and contacts all accessible Spark executor nodes and initializes H2O services (such as the key-value store and RPC) inside the executors' JVMs.
- In case of the external backend, either starts H2O cluster on YARN and connects to it or connects to the existing H2O cluster right away (depends on the configuration).

The next sections show how to start H2OContext in all supported clients and both backends.

Starting Sparkling Water

This section focuses on how Sparkling Water can be started in both backends and all language clients. You can submit a Sparkling Water code as a Spark batch job or you can explore its functionality in an interactive shell.

Before you start, please make sure that you have downloaded Sparkling Water from `https://www.h2o.ai/download/` for your desired Spark version.

Setting up the Environment

In the case of Scala, all dependencies are already provided inside the Sparkling Water artifacts.

In the case of Python, please make sure that you have the following Python packages installed:

- requests
- tabulate

. Also please make sure that your Python environment is set-up to run regular Spark applications.

In the case of R, please make sure that SparklyR and H2O libraries are installed. The H2O version needs to match the version used in Sparkling Water. For example, when using RSparkling 3.30.0.7, make sure that you have H2O of version 3.30.0.7 installed.

Starting Interactive Shell with Sparkling Water

To start interactive Scala shell, run:

```
./bin/sparkling-shell
```

To start interactive Python shell, run:

```
./bin/pysparkling
```

To use RSparkling in an interactive environment, we suggest using RStudio.

Starting Sparkling Water in Internal Backend

In the internal backend, the H2O cluster is created automatically during the call of `H2OContext.getOrCreate`. Since it is not technically possible to get the number of executors in Spark, Sparkling Water tries to discover all executors during the initiation of `H2OContext` and starts H2O instance inside each of discovered executors. This solution is the easiest to deploy; however when Spark or YARN kills the executor, the whole H2O cluster goes down since H2O does not support high availability. Also, there are cases where Sparkling Water is not able to discover all Spark executors and will start just on the subset of executors. The shape of the cluster can not be changed later. Internal backend is the default backend for Sparkling Water. It can be changed via spark configuration property `spark.ext.h2o.backend.cluster.mode` to **external** or **internal**. Another way how to change the type of backend is by calling `setExternalClusterMode()` or `setInternalClusterMode()` method on `H2OConf` class instance. `H2OConf` is a simple wrapper around `SparkConf` and inherits all properties in spark configuration.

`H2OContext` can be started as:

- **Scala**

```
import ai.h2o.sparkling._
val conf = new H2OConf().setInternalClusterMode()
val h2oContext = H2OContext.getOrCreate(conf)
```

- **Python**

```
from pysparkling import *
conf = H2OConf().setInternalClusterMode()
h2oContext = H2OContext.getOrCreate(conf)
```

- **R**

```
library(sparklyr)
library(rsparkling)
spark_connect(master = "local", version = "3.0.0")
conf <- H2OConf()$setInternalClusterMode()
h2oContext <- H2OContext.getOrCreate(conf)
```

If `spark.ext.h2o.backend.cluster.mode` property was set to **internal** either on the command line or on the `SparkConf`, the following call is sufficient

- **Scala**

```
import ai.h2o.sparkling._
val h2oContext = H2OContext.getOrCreate()
```

- **Python**

```
from pysparkling import *
h2oContext = H2OContext.getOrCreate()
```

- **R**

```
library(sparklyr)
library(rsparkling)
spark_connect(master = "local", version = "3.0.0")
h2oContext <- H2OContext.getOrCreate()
```

External Backend

In the external cluster, we use the H2O cluster running separately from the rest of the Spark application. This separation gives us more stability because we are no longer affected by Spark executors being killed, which can lead (as in the previous mode) to h2o cluster being killed as well.

There are two deployment strategies of the external cluster: manual and automatic. In manual mode, we need to start the H2O cluster, and in the automatic mode, the cluster is started for us automatically based on our configuration. In Hadoop environments, the creation of the cluster is performed by a simple process called H2O driver. When the cluster is fully formed, the H2O driver terminates. In both modes, we have to store a path of H2O driver jar to the environment variable H2O_DRIVER_JAR.

```
H2O_DRIVER_JAR=$(./bin/get-h2o-driver.sh
    some_hadoop_distribution)
```

Automatic Mode of External Backend

In the automatic mode, the H2O cluster is started automatically. The cluster can be started automatically only in YARN environment at the moment. We recommend this approach, as it is easier to deploy external clusters in this mode, and it is also more suitable for production environments. When the H2O cluster

is started on YARN, it is started as a map-reduce job, and it always uses the flat-file approach for nodes to cloud up.

First, get H2O driver, for example, for cdh 5.8, as:

```
H2O_DRIVER_JAR=$(./bin/get-h2o-driver.sh cdh5.8)
```

To start an H2O cluster and connect to it, run:

- **Scala**

```
import ai.h2o.sparkling._
val conf = new H2OConf()
    .setExternalClusterMode()
    .useAutoClusterStart()
    .setH2ODriverPath("path_to_h2o_driver")
    .setClusterSize(1)
    .setExternalMemory("2G")
    .setYARNQueue("abc")
val hc = H2OContext.getOrCreate(conf)
```

- **Python**

```
from pysparkling import *
conf = H2OConf()
    .setExternalClusterMode()
    .useAutoClusterStart()
    .setH2ODriverPath("path_to_h2o_driver")
    .setClusterSize(1)
    .setExternalMemory("2G")
    .setYARNQueue("abc")
hc = H2OContext.getOrCreate(conf)
```

- **R**

```
library(sparklyr)
library(rsparkling)
spark_connect(master = "local", version = "3.0.0")
conf <- H2OConf()
    $setExternalClusterMode()
    $useAutoClusterStart()
    $setH2ODriverPath("path_to_h2o_driver")
    $setClusterSize(1)
    $setExternalMemory("2G")
```

```
    $setYARNQueue("abc")
hc <- H2OContext.getOrCreate(conf)
```

In case we stored the path of the driver H2O jar to environmental variable H2O_DRIVER_JAR, we do not need to call `setH2ODriverPath` as Sparkling Water will read the path from the environmental variable.

When specifying the queue, we recommend that this queue has YARN preemption off to have a stable H2O cluster.

Manual Mode of External Backend on Hadoop

In the manual mode, we need to start the H2O cluster before connecting to it manually. At this section, we will start the cluster on Hadoop.

First, get the H2O driver, for example, for cdh 5.8, as:

```
H2O_DRIVER_JAR=$(./bin/get-h2o-driver.sh cdh5.8)
```

Also, set path to sparkling-water-assembly-extensions-2.12-all.jar which is bundled in Sparkling Water for Spark 3.0 archive.

```
SW_EXTENSIONS_ASSEMBLY=/path/to/sparkling-water
    -3.30.0.7-1-3.0/jars/sparkling-water-assembly-
    extensions_2.12-3.30.0.7-1-2.4-all.jar
```

Let's start the H2O cluster on Hadoop:

```
hadoop -jar $H2O_DRIVER_JAR -libjars
    $SW_EXTENSIONS_ASSEMBLY -sw_ext_backend -jobname
    test -nodes 3 -mapperXmx 6g
```

The `-sw_ext_backend` option is required as without it, the cluster won't allow Sparkling Water client to connect to it.

After this step, we should have an H2O cluster with 3 nodes running on Hadoop.

To connect to this external cluster, run the following commands:

- **Scala**

```
import ai.h2o.sparkling._
val conf = new H2OConf()
    .setExternalClusterMode()
```

```
    .useManualClusterStart()
    .setH2OCluster("representant_ip",
        representant_port)
    .setCloudName("test")
val hc = H2OContext.getOrCreate(conf)
```

- **Python**

```
from pysparkling import *
conf = H2OConf()
    .setExternalClusterMode()
    .useManualClusterStart()
    .setH2OCluster("representant_ip",
        representant_port)
    .setCloudName("test")
hc = H2OContext.getOrCreate(conf)
```

- **R**

```
library(sparklyr)
library(rsparkling)
spark_connect(master = "local", version = "3.0.0")
conf <- H2OConf()
    $setExternalClusterMode()
    $useManualClusterStart()
    $setH2OCluster("representant_ip", representant
        _port)
    $setCloudName("test")
hc <- H2OContext.getOrCreate(conf)
```

The `representant_ip` and `representant_port` should be IP and port of the leader node of the started H2O cluster from the previous step.

Manual Mode of External Backend without Hadoop (standalone)

In the manual mode, we need to start the H2O cluster before connecting to it manually. At this section, we will start the cluster as a standalone application (without Hadoop).

First, get the assembly H2O Jar:

```
H2O_JAR=$(./bin/get-h2o-driver.sh standalone)
```

.. code:: bash

Also, set path to sparkling-water-assembly-extensions-2.12-all.jar which is bundled in Sparkling Water for Spark 3.0 archive.

```
SW_EXTENSIONS_ASSEMBLY=/path/to/sparkling-water
    -3.30.0.7-1-3.0/jars/sparkling-water-assembly-
    extensions_2.12-3.30.0.7-1-2.4-all.jar
```

To start an external H2O cluster, run:

```
java -cp "$H2O_JAR:$SW_EXTENSIONS_ASSEMBLY" water.
    H2OApp -allow_clients -name test -flatfile
    path_to_flatfile
```

where the flat-file content are lines in the format of ip:port of the nodes where H2O is supposed to run. To read more about flat-file and its format, please see `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/howto/H2O-DevCmdLine.md#flatfile`.

To connect to this external cluster, run the following commands:

- **Scala**

```
import ai.h2o.sparkling._
val conf = new H2OConf()
    .setExternalClusterMode()
    .useManualClusterStart()
    .setH2OCluster("representant_ip",
        representant_port)
    .setCloudName("test")
val hc = H2OContext.getOrCreate(conf)
```

- **Python**

```
from pysparkling import *
conf = H2OConf()
    .setExternalClusterMode()
    .useManualClusterStart()
    .setH2OCluster("representant_ip",
        representant_port)
    .setCloudName("test")
```

```
hc = H2OContext.getOrCreate(conf)
```

- **R**

```
library(sparklyr)
library(rsparkling)
spark_connect(master = "local", version = "3.0.0")
conf <- H2OConf()
    $setExternalClusterMode()
    $useManualClusterStart()
    $setH2OCluster("representant_ip", representant
        _port)
    $setCloudName("test")
hc <- H2OContext.getOrCreate(conf)
```

The `representant_ip` and `representant_port` need to be IP and port of the leader node of the started H2O cluster from the previous step.

Memory Management

In the case of the internal backend, H2O resides in the same executor JVM as Spark and the memory provided for H2O is configured via Spark. Executor memory (i.e., memory available for H2O in internal backend) can be configured via the Spark configuration property `spark.executor.memory`. For example, as:

```
./bin/sparkling-shell --conf spark.executor.memory=5g
```

or configure the property in:

```
$SPARK_HOME/conf/spark-defaults.conf
```

Driver memory (i.e., memory available for H2O client running inside the Spark driver) can be configured via the Spark configuration property `spark.driver.memory`. For example, as:

```
./bin/sparkling-shell --conf spark.driver.memory=5g
```

or configure the property in:

```
$SPARK_HOME/conf/spark-defaults.conf
```

In the external backend, only the H2O client (Scala only) is running in the Spark driver and is affected by Spark memory configuration. Memory has to be configured explicitly for the H2O nodes in the external backend via the `spark.ext.h2o.external.memory` option or `setExternalMemory` setter on `H2OConf`.

For YARN-specific configuration, refer to the Spark documentation `https://spark.apache.org/docs/latest/running-on-yarn.html`.

Data Manipulation

This section covers conversions between Spark and H2O Frames and also ways how H2O Frames can be created directly.

Creating H2O Frames

This section covers multiple ways how a H2O Frame can be created.

Convert from RDD, DataFrame or Dataset

H2OFrame can be created by converting Spark entities into it. **H2OContext** provides the **asH2OFrame** method which accepts Spark Dataset, DataFrame and RDD[T] as an input parameter. In the case of RDD, the type `T` 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.

Example of converting Spark DataFrame into H2OFrame is:

- **Scala**

```
1 h2oContext.asH2OFrame(sparkDf)
```

- **Python**

```
1 h2oContext.asH2OFrame(sparkDf)
```

- **R**

```
1 h2oContext$asH2OFrame(sparkDf)
```

You can also specify the name of the resulting H2OFrame as

- **Scala**

```
1 h2oContext.asH2OFrame(sparkDf, "frameName")
```

- **Python**

```
1 h2oContext.asH2OFrame(sparkDf, "frameName")
```

- **R**

```
h2oContext$asH2OFrame(sparkDf, "frameName")
```

The method is overloaded and accepts also RDDs and Datasets as inputs.

Creating H2OFrame from an Existing Key

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

- **Scala**

```
import ai.h2o.sparkling.H2OFrame
val frame = H2OFrame("train.hex")
```

- **Python**

```
import h2o
val frame = h2o.get_frame("train.hex")
```

- **R**

```
library(h2o)
val frame = h2o.getFrame("train.hex")
```

Create H2O Frame Directly

H2O Frame can also be created directly. To see how you can create H2O Frame directly in Python and R, please see H2O documentation at `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-munging.html`.

To create H2O Frame directly in Scala, you can:

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

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

- load file from HDFS/S3/S3N/S3A:

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

Converting H2O Frames to Spark entities

This section covers how we can convert H2O Frame to Spark entities.

Convert to RDD

Conversion to RDD is only available in the Scala API of Sparkling Water

The **H2OContext** class provides the method **asRDD**, which creates an RDD-like wrapper around the provided **H2OFrame**.

The method expects the type `T` to create a correctly-typed RDD. The conversion requires type `T` 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 of converting H2OFrame into Spark RDD is:

```
1 val df: H2OFrame = ...
2 h2oContext.asRDD[Weather](df)
```

Convert to DataFrame

The **H2OContext** class provides the method **asSparkFrame**, which creates a `DataFrame`-like wrapper around the provided `H2OFrame`.

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

Example of converting H2OFrame into Spark DataFrame is:

- **Scala**

```
1 h2oContext.asSparkFrame(h2oFrame)
```

- **Python**

```
1 h2oContext.asSparkFrame(h2oFrame)
```

- **R**

```
1 h2oContext$asSparkFrame(h2oFrame)
```

Mapping between H2OFrame And Data Frame Types

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

Primitive 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

Mapping between H2OFrame and RDD[T] Types

For converting rdd to H2OFrame and vice-versa, as type `T` in RDD[T] we support following types:

T
NA
Byte
Short
Integer
Long
Float
Double
String
Boolean
java.sql.Timestamp
Any scala class extending scala `Product`
org.apache.spark.mllib.regression.LabeledPoint
org.apache.spark.ml.linalg.Vector
org.apache.spark.mllib.linalg

Using Spark Data Sources with H2OFrame

Spark SQL provides a configurable data source for SQL tables. Sparkling Water enables `H2OFrame`s to be used as a data source to load/save data from/to Spark DataFrame. This API is supported only in Scala and Python APIs

Reading from `H2OFrame`

To read H2OFrame as Spark DataFrame, run:

- **Scala**

```
1 spark.read.format("h2o").load(frameKey)
```

- **Python**

```
1 spark.read.format("h2o").load(frameKey)
```

You can also specify the key as option:

- **Scala**

```
spark.read.format("h2o").option("key", frameKey).
    load()
```

- **Python**

```
spark.read.format("h2o").option("key", frameKey).
    load()
```

If you specify the key as the option and inside the load method, the option has the precedence.

Saving to `H2OFrame`

To save DataFrame into a H2OFrame, run:

- **Scala**

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

- **Python**

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

You can also specify the key as option:

- **Scala**

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

- **Python**

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

If you specify the key as the option and inside the save method, the option has the precedence.

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 a 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.

Calling H2O Algorithms

This section describes how to call H2O algorithms and feature transformers from Sparkling Water. In Scala and Python, Sparkling Water exposes H2O algorithms via its own API. In R, we still need to use H2O's R API.

Following feature transformers are exposed in Sparkling Water:

- H2OWord2Vec
- H2OTargetEncoder

Currently, the following algorithms are exposed:

- DeepLearning
- DRF
- GBM
- XGBoost
- AutoML
- GridSearch
- KMeans
- GLM
- GAM
- CoxPH
- Isolation Forest

In the exposed algorithms above, it is up on the users to decide whether they want to run classification or regression problem using the given algorithm. H2O decides whether it will run classification or regression based on the type of the label column. If it is categorical, classification will be performed, otherwise regression.

If you do not want to be worried about this, we also expose specific regressors and classifiers to make this more explicit. For example, if you decide to use H2OAutoMLRegressor, you can be sure that the algorithm will do regression and you do not need to worry about the type of the label column. These wrappers automatically convert the label column to required type for the given problem. The following wrappers exist:

- `H2OAutoMLClassifer` and `H2OAutoMLRegressor`

- `H2ODeepLearningClassifier` and `H2ODeepLearningRegressor`
- `H2ODRFClassifier` and `H2ODRFRegressor`
- `H2OGBMClassifier` and `H2OGBMRegressor`
- `H2OGLMClassifier` and `H2OGLMRegressor`
- `H2OXGBoostClassifier` and `H2OOXGBoostRegressor`
- `H2OGAMClassifier` and `H2OGAMRegressor`

First, let's create H2OContext:

- **Scala**

```
import ai.h2o.sparkling._
import java.net.URI
val hc = H2OContext.getOrCreate()
```

- **Python**

```
from pysparkling import *
hc = H2OContext.getOrCreate()
```

Parse the data using H2O and convert them to Spark Frame:

- **Scala**

```
val frame = new H2OFrame(new URI("https://raw.
    githubusercontent.com/h2oai/sparkling-water/
    master/examples/smalldata/prostate/prostate.
    csv"))
val sparkDF = hc.asSparkFrame(frame).withColumn("
    CAPSULE", $"CAPSULE" cast "string")
val Array(trainingDF, testingDF) = sparkDF.
    randomSplit(Array(0.8, 0.2))
```

- **Python**

```
import h2o
frame = h2o.import_file("https://raw.
    githubusercontent.com/h2oai/sparkling-water/
    master/examples/smalldata/prostate/prostate.
    csv")
sparkDF = hc.asSparkFrame(frame)
```

```
sparkDF = sparkDF.withColumn("CAPSULE", sparkDF.
    CAPSULE.cast("string"))
[trainingDF, testingDF] = sparkDF.randomSplit
    ([0.8, 0.2])
```

Train the model. You can configure all the available arguments using provided setters, such as the label column:

- **Scala**

```
import ai.h2o.sparkling.ml.algos.H2OGBM
val estimator = new H2OGBM()
        .setLabelCol("CAPSULE")
val model = estimator.fit(trainingDF)
```

- **Python**

```
from pysparkling.ml import H2OGBM
estimator = H2OGBM(labelCol = "CAPSULE")
model = estimator.fit(trainingDF)
```

Instead of calling the generic wrapper, we can do regression explicitly as:

- **Scala**

```
import ai.h2o.sparkling.ml.algos.H2OGBMRegressor
val estimator = new H2OGBMRegressor()
        .setLabelCol("CAPSULE")
val model = estimator.fit(trainingDF)
```

- **Python**

```
from pysparkling.ml import H2OGBMRegressor
estimator = H2OGBMRegressor(labelCol = "CAPSULE")
model = estimator.fit(trainingDF)
```

or classification explicitly as:

- **Scala**

```
import ai.h2o.sparkling.ml.algos.H2OGBMClassifier
val estimator = new H2OGBMClassifier()
        .setLabelCol("CAPSULE")
val model = estimator.fit(trainingDF)
```

- **Python**

```
from pysparkling.ml import H2OGBMClassifier
estimator = H2OGBMClassifier(labelCol = "CAPSULE")
model = estimator.fit(trainingDF)
```

Run Predictions:

- **Scala**

```
model.transform(testingDF).show(false)
```

- **Python**

```
model.transform(testingDF).show(truncate = False)
```

The code is identical to the rest of the exposed algorithms.

In the case of AutoML, after you have fit the model, you can obtain the leaderboard frame using the `estimator.getLeaderBoard()` method.

In case of Grid, after you have fit the model, you can obtain the grid models, their parameters and metrics using `estimator.getGridModels()`, `estimator.getGridModelsParams()` and `estimator.getGridModelsMetrics()`.

Productionizing MOJOs from H2O-3

Loading the H2O-3 MOJOs

When training algorithm using Sparkling Water API, Sparkling Water always produces `H2OMOJOModel`. It is however also possible to import existing MOJO into Sparkling Water ecosystem from H2O-3. After importing the H2O-3 MOJO the API is unified for the loaded MOJO and the one created in Sparkling Water, for example, using `H2OXGBoost`.

H2O MOJOs can be imported to Sparkling Water from all data sources supported by Apache Spark such as local file, S3 or HDFS and the semantics of the import is the same as in the Spark API.

When creating a MOJO specified by a relative path and HDFS is enabled, the method attempts to load the MOJO from the HDFS home directory of the current user. In case we are not running on HDFS-enabled system, we create the mojo from a current working directory.

- **Scala**

```
import ai.h2o.sparkling.ml.models._
val model = H2OMOJOModel.createFromMojo("
    prostate_mojo.zip")
```

- **Python**

```
from pysparkling.ml import *
model = H2OMOJOModel.createFromMojo("prostate_mojo
    .zip")
```

- **R**

```
library(rsparkling)
sc <- spark_connect(master = "local")
model <- H2OMOJOModel.createFromMojo("prostate_
    mojo.zip")
```

Absolute local path can also be used. To create a MOJO model from a locally available MOJO, call:

- **Scala**

```
import ai.h2o.sparkling.ml.models._
```

```
2 val model = H2OMOJOModel.createFromMojo("/Users/
    peter/prostate_mojo.zip")
```

- **Python**

```
1 from pysparkling.ml import *
2 model = H2OMOJOModel.createFromMojo("/Users/peter/
    prostate_mojo.zip")
```

- **R**

```
1 library(rsparkling)
2 sc <- spark_connect(master = "local")
3 model <- H2OMOJOModel.createFromMojo("/Users/peter
    /prostate_mojo.zip")
```

Absolute paths on Hadoop can also be used. To create a MOJO model from a MOJO stored on HDFS, call:

- **Scala**

```
1 import ai.h2o.sparkling.ml.models._
2 val model = H2OMOJOModel.createFromMojo("/user/
    peter/prostate_mojo.zip")
```

- **Python**

```
1 from pysparkling.ml import *
2 model = H2OMOJOModel.createFromMojo("/user/peter/
    prostate_mojo.zip")
```

- **R**

```
1 library(rsparkling)
2 sc <- spark_connect(master = "local")
3 model <- H2OMOJOModel.createFromMojo("/user/peter/
    prostate_mojo.zip")
```

The call loads the mojo file from the following location `hdfs://server:port/user/peter/prostate_mojo.zip`, where `server` and `port` are automatically filled in by Spark.

We can also manually specify the type of data source we need to use, in that case, we need to provide the schema:

- **Scala**

```scala
import ai.h2o.sparkling.ml.models._
// HDFS
val modelHDFS = H2OMOJOModel.createFromMojo("hdfs:///user/peter/prostate_mojo.zip")
// Local file
val modelLocal = H2OMOJOModel.createFromMojo("file:///Users/peter/prostate_mojo.zip")
```

- **Python**

```python
from pysparkling.ml import *
# HDFS
modelHDFS = H2OMOJOModel.createFromMojo("hdfs:///user/peter/prostate_mojo.zip")
# Local file
modelLocal = H2OMOJOModel.createFromMojo("file:///Users/peter/prostate_mojo.zip")
```

- **R**

```r
library(rsparkling)
sc <- spark_connect(master = "local")
# HDFS
modelHDFS <- H2OMOJOModel.createFromMojo("hdfs:///user/peter/prostate_mojo.zip")
# Local file
modelLocal <- H2OMOJOModel.createFromMojo("file:///Users/peter/prostate_mojo.zip")
```

The loaded model is an immutable instance, so it's not possible to change the configuration of the model during its existence. On the other hand, the model can be configured during its creation via `H2OMOJOSettings`:

- **Scala**

```scala
import ai.h2o.sparkling.ml.models._
val settings = H2OMOJOSettings(
    convertUnknownCategoricalLevelsToNa = true,
    convertInvalidNumbersToNa = true)
val model = H2OMOJOModel.createFromMojo("
    prostate_mojo.zip", settings)
```

- **Python**

```
from pysparkling.ml import *
settings = H2OMOJOSettings(
    convertUnknownCategoricalLevelsToNa = True,
    convertInvalidNumbersToNa = True)
model = H2OMOJOModel.createFromMojo("prostate_mojo
    .zip", settings)
```

- **R**

```
library(rsparkling)
sc <- spark_connect(master = "local")
settings <- H2OMOJOSettings(
    convertUnknownCategoricalLevelsToNa = TRUE,
    convertInvalidNumbersToNa = TRUE)
model <- H2OMOJOModel.createFromMojo("prostate_
    mojo.zip", settings)
```

To score the dataset using the loaded mojo, call:

- **Scala**

```
model.transform(dataset)
```

- **Python**

```
model.transform(dataset)
```

- **R**

```
model$transform(dataset)
```

In Scala, the `createFromMojo` method returns a mojo model instance casted as a base class `H2OMOJOModel`. This class holds only properties that are shared accross all MOJO model types from the following type hierarchy:

- `H2OMOJOModel`
- `H2OUnsupervisedMOJOModel`
- `H2OSupervisedMOJOModel`
- `H2OTreeBasedSupervisedMOJOModel`

If a Scala user wants to get a property specific for a given MOJO model type, they must utilize casting or call the `createFromMojo` method on the specific MOJO model type.

```
import ai.h2o.sparkling.ml.models._
val specificModel = H2OTreeBasedSupervisedMOJOModel.
    createFromMojo("prostate_mojo.zip")
println(s"Ntrees: ${specificModel.getNTrees()}") //
    Relevant only to GBM, DRF and XGBoost
```

Exporting the loaded MOJO model using Sparkling Water

To export the MOJO model, call `model.write.save(path)`. In case of Hadoop enabled system, the command by default uses HDFS.

Importing the previously exported MOJO model from Sparkling Water

To import the Sparkling Water MOJO model, call `H2OMOJOModel.read.load(path)`. In case of Hadoop enabled system, the command by default uses HDFS.

Accessing additional prediction details

After computing predictions, the `prediction` column contains in case of classification problem the predicted label and in case regression problem the predicted number. If we need to access more details for each prediction, , see the content of a detailed prediction column. By default, the column is named named `detailed_prediction` . It could contain, for example, predicted probabilities for each predicted label in case of classification problem, shapley values and other information.

Customizing the MOJO Settings

We can configure the output and format of predictions via the `H2OMOJOSettings`. The available options are:

- `predictionCol` - Specifies the name of the generated prediction column. Default value is `prediction`.

- `detailedPredictionCol` - Specifies the name of the generated detailed prediction column. The detailed prediction column, if enabled, contains additional details, such as probabilities, shapley values etc. The default value is `detailed_prediction`.
- `convertUnknownCategoricalLevelsToNa` - Enables or disables conversion of unseen categoricals to NAs. By default, it is disabled.
- `convertInvalidNumbersToNa` - Enables or disables conversion of invalid numbers to NAs. By default, it is disabled.
- `withContributions` - Enables or disables computing Shapley values. Shapley values are generated as a sub-column for the detailed prediction column.
- `withLeafNodeAssignments` - When enabled, a user can obtain the leaf node assignments after the model training has finished. By default, it is disabled.
- `withStageResults` - When enabled, a user can obtain the stage results for tree-based models. By default, it is disabled and also it's not supported by XGBoost although it's a tree-based algorithm.
- `dataFrameSerializer` - A full name of a serializer used for serialization and deserialization of Spark DataFrames to a JSON value within `NullableDataFrameParam`

Methods available on MOJO Model

Obtaining Domain Values

To obtain domain values of the trained model, we can run `getDomainValues()` on the model. This call returns a mapping from a column name to its domain in a form of array.

Obtaining Model Category

The method `getModelCategory` can be used to get the model category (such as `binomial`, `multinomial` etc).

Obtaining Feature Types

The method `getFeatureTypes` returns a map/dictionary from a feature name to a corresponding feature type [`enum` (categorical), `numeric`, `string`, etc.]. These pieces helps to understand how individual columns of the training dataset were treated during the model training.

Obtaining Feature Importances

The method `getFeatureImportances` returns a data frame describing importance of each feature. The importance is expressed by several numbers (Relative Importance, Scaled Importance and Percentage).

Obtaining Scoring History

The method `getScoringHistory` returns a data frame describing how the model evolved during the training process according to a certain training and validation metrics.

Obtaining Training Params

The method `getTrainingParams` can be used to get map containing all training parameters used in the H2O. It is a map from parameter name to the value. The parameters name use the H2O's naming structure. An alternative approach for Scala and Python API is to use a getter method on the MOJO model instance for a given training parameter. The getter methods utilize Sparkling Water naming conventions (E.g. H2O name: `max_depth`, getter method name: `getMaxDepth`).

Obtaining Metrics

There are several methods to obtain metrics from the MOJO model. All return a map from metric name to its double value.

- `getTrainingMetrics` - obtain training metrics
- `getValidationMetrics` - obtain validation metrics
- `getCrossValidationMetrics` - obtain cross validation metrics

We also have method `getCurrentMetrics` which gets one of the metrics above based on the following algorithm:

If cross validation was used, ie, `setNfolds` was called and value was higher than zero, this method returns cross validation metrics. If cross validation was not used, but validation frame was used, the method returns validation metrics. Validation frame is used if `setSplitRatio` was called with value lower than one. If neither cross validation or validation frame was used, this method returns the training metrics.

Obtaining Leaf Node Assignments

To obtain the leaf node assignments, please first make sure to set `withLeafNodeAssignments` on your MOJO settings object. The leaf node assignments are now stored in the `$detailedPredictionCol.leafNodeAssignm` column on the dataset obtained from the prediction. Please replace `$detailedPredictionCol` with the actual value of your detailed prediction col. By default, it is `detailed_prediction`.

Obtaining Stage Probabilities

To obtain the stage results, please first make sure to set `withStageResults` to true on your MOJO settings object. The stage results for regression and anomaly detection problems are stored in the `$detailedPredictionCol.stageRe` on the dataset obtained from the prediction. The stage results for classification(binomial, multinomial) problems are stored under `$detailedPredictionCol.stageProbabilities`. Please replace `$detailedPredictionCol` with the actual value of your detailed prediction col. By default, it is `detailed_prediction`.

The stage results are an array of values, where a value at the position *t* is the prediction/probability combined from contributions of trees *T1, T2, ..., Tt*. For *t* equal to a number of model trees, the value is the same as the final prediction/probability. The stage results (probabilities) for the classification problem are represented by a list of columns, where one column contains stage probabilities for a given prediction class.

Productionizing MOJOs from Driverless AI

MOJO scoring pipeline artifacts, created in Driverless AI, can be used in Spark to carry out predictions in parallel using the Sparkling Water API. This section shows how to load and run predictions on the MOJO scoring pipeline in Sparkling Water.

Note: Sparkling Water is backward compatible with MOJO versions produced by different Driverless AI versions.

One advantage of scoring the MOJO artifacts is that `H2OContext` does not have to be created if we only want to run predictions on MOJOs using Spark. This is because the scoring is independent of the H2O run-time. It is also important to mention that the format of prediction on MOJOs from Driverless AI differs from predictions on H2O-3 MOJOs. The format of Driverless AI prediction is explained bellow.

Requirements

In order to use the MOJO scoring pipeline, Driverless AI license has to be passed to Spark. This can be achieved via `--jars` argument of the Spark launcher scripts.

Note: In Local Spark mode, please use `--driver-class-path` to specify the path to the license file.

We also need Sparkling Water distribution which can be obtained from `https://www.h2o.ai/download/`. After we downloaded the Sparkling Water distribution, extract it, and go to the extracted directory.

Loading and Score the MOJO

First, start the environment for the desired language with Driverless AI license. There are two variants. We can use Sparkling Water prepared scripts which put required dependencies on the Spark classpath or we can use Spark directly and add the dependencies manually.

In case of Scala, run:

```
1 ./bin/spark-shell --jars license.sig,jars/sparkling-
    water-assembly_2.12-3.30.1.1-1-3.0-all.jar
```

or

```
./bin/sparkling-shell --jars license.sig
```

In case of Python, run:

```
./bin/pyspark --jars license.sig --py-files py/
   h2o_pysparkling_3.0-3.30.1.1-1-3.0.zip
```

or

```
./bin/pysparkling --jars license.sig
```

In case of R, run in R console or RStudio:

```
library(sparklyr)
library(rsparkling)
config <- spark_config()
config$sparklyr.jars.default <- "license.sig"
spark_connect(master = "local", version = "3.0.0",
   config = config)
```

At this point, we should have Spark interactive terminal where we can carry out predictions. For productionalizing the scoring process, we can use the same configuration, except instead of using Spark shell, we would submit the application using `./bin/spark-submit`.

Now Load the MOJO as:

- **Scala**

```
import ai.h2o.sparkling.ml.models.
   H2OMOJOPipelineModel
val settings = H2OMOJOSettings(predictionCol = "
   fruit_type",
   convertUnknownCategoricalLevelsToNa = true)
val mojo = H2OMOJOPipelineModel.createFromMojo("
   file:///path/to/the/pipeline_mojo.zip",
   settings)
```

- **Python**

```
from pysparkling.ml import H2OMOJOPipelineModel
```

```
settings = H2OMOJOSettings(predictionCol = "
    fruit_type",
    convertUnknownCategoricalLevelsToNa = True)
mojo = H2OMOJOPipelineModel.createFromMojo("file
    :///path/to/the/pipeline_mojo.zip", settings)
```

- **R**

```
library(rsparkling)
settings <- H2OMOJOSettings(predictionCol = "fruit
    _type", convertUnknownCategoricalLevelsToNa =
    TRUE)
mojo <- H2OMOJOPipelineModel.createFromMojo("file
    :///path/to/the/pipeline_mojo.zip", settings)
```

In the examples above `settings` is an optional argument. If it's not specified, the default values are used.

Prepare the dataset to score on:

- **Scala**

```
val dataFrame = spark.read.option("header", "true"
    ).option("inferSchema", "true").csv("file:///
    path/to/the/data.csv")
```

- **Python**

```
dataFrame = spark.read.option("header", "true").
    option("inferSchema", "true").csv("file:///
    path/to/the/data.csv")
```

- **R**

```
dataFrame <- spark_read_csv(sc, name = "table_name
    ", path = "file:///path/to/the/data.csv",
    header = TRUE)
```

And finally, score the mojo on the loaded dataset:

- **Scala**

```
val predictions = mojo.transform(dataFrame)
```

- **Python**

```
predictions = mojo.transform(dataFrame)
```

- **R**

```
predictions <- mojo$$transform(dataFrame)
```

We can select the predictions as:

- **Scala**

```
predictions.select("prediction")
```

- **Python**

```
predictions.select("prediction")
```

- **R**

```
predictions <- select(dataFrame, "prediction")
```

The output data frame contains all the original columns plus the prediction column which is by default named `prediction`. The prediction column contains all the prediction detail. Its name can be modified via the `H2OMOJOSettings` object.

Predictions Format

The `predictionCol` contains sub-columns with names corresponding to the columns Driverless AI identified as output columns. For example, if Driverless API MOJO pipeline contains one output column `AGE` (for example regression problem), the prediction column contains another sub-column named `AGE`. If The MOJO pipeline contains multiple output columns, such as `VALID.0` and `VALID.1` (for example classification problems), the prediction column contains two sub-columns with the aforementioned names.

If this option is disabled, the `predictionCol` contains the array of predictions without the column names. For example, if Driverless API MOJO pipeline contains one output column `AGE` (for example regression problem), the prediction column contains array of size 1 with the predicted value. If The MOJO pipeline contains multiple output columns, such as `VALID.0` and `VALID.1` (for example classification problems), the prediction column contains array of size 2 containing predicted probabilities for each class.

By default, this option is enabled.

Customizing the MOJO Settings

We can configure the output and format of predictions via the H2OMOJOSettings. The available options are

- `predictionCol` - Specifies the name of the generated prediction column. The default value is `prediction`.
- `convertUnknownCategoricalLevelsToNa` - Enables or disables conversion of unseen categoricals to NAs. By default, it is disabled.
- `convertInvalidNumbersToNa` - Enables or disables conversion of invalid numbers to NAs. By default, it is disabled.

Troubleshooting

If you see the following exception during loading the MOJO pipeline: java.io.IOException: MOJO doesn't contain `resource mojo/pipeline.pb`, then it means you are adding incompatible mojo-runtime.jar on your classpath. It is not required and also not suggested to put the JAR on the classpath as Sparkling Water already bundles the correct dependencies.

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]). Deployment on top of Kubernetes is described in the next section. 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 `ai.h2o.sparkling.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

Referencing Sparkling Water

Using Assembly Jar

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

An application submission with Sparkling Water assembly jar is using the `--jars` option which references included jar

[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 /sparkling-water-distribution/jars/sparkling-
     water-assembly_2.12-3.30.1.1-1-3.0-all.jar \
  --class ai.h2o.sparkling.SparklingWaterDriver
```

Using PySparkling Zip

An application submission for PySparkling is using the `--py-files` option which references the PySparkling zip package

```
$SPARK_HOME/bin/spark-submit \
  --py-files /sparkling-water-distribution/py/
     h2o_pysparkling_3.0-3.30.1.1-1-3.0.zip \
  app.py
```

Using the 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.

```
$SPARK_HOME/bin/spark-submit \
  --packages ai.h2o:sparkling-water-package_2
     .12:3.30.0.7-1-3.0 \
  --class ai.h2o.sparkling.SparklingWaterDriver
```

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-package_2
    .12:3.30.0.7-1-3.0
```

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

Target Deployment Environments

Sparkling Water supports deployments to the following Spark cluster types:

- Local cluster
- Standalone cluster
- YARN cluster

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-package_2
       .12:3.30.0.7-1-3.0 \
6   --class ai.h2o.sparkling.SparklingWaterDriver
```

On a 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 `SparklingWaterDriver` on a standalone cluster where the master node is exposed on IP `machine-foo.bar.com` and port `7077`:

```
1 $SPARK_HOME/bin/spark-submit \
2   --conf spark.executor.memory=5g \
3   --conf spark.driver.memory=5g \
```

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

```
4   --master spark://machine-foo.bar.com:7077 \
5   --packages ai.h2o:sparkling-water-package_2
        .12:3.30.0.7-1-3.0 \
6   --class ai.h2o.sparkling.SparklingWaterDriver
```

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

On a 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 points to Hadoop configuration directory (e.g., `/etc/hadoop/conf`).

```
1  $SPARK_HOME/bin/spark-submit \
2    --conf spark.executor.memory=5g \
3    --conf spark.driver.memory=5g \
4    --num-executors 5 \
5    --master yarn \
6    --deploy-mode client
7    --packages ai.h2o:sparkling-water-package_2
         .12:3.30.0.7-1-3.0 \
8    --class ai.h2o.sparkling.SparklingWaterDriver
```

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

DataBricks Cloud

This section describes how to use Sparkling Water and PySparkling with DataBricks. The first part describes how to create a cluster for Sparkling Water/PySparkling and then discusses how to use Sparkling Water and PySparkling in Databricks.

DataBricks cloud is Integrated with Sparkling Water and Pysparkling. Only internal Sparkling Water backend may be used.

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

Creating a Cluster

Requirements:

- Databricks Account
- AWS Account

Steps:

1. In Databricks, click **Create Cluster** in the Clusters dashboard.
2. Select your Databricks Runtime Version.
3. Select 0 on-demand workers. On demand workers are currently not supported with Sparkling Water.
4. In the SSH tab, upload your public key. You can create a public key by running the below command in a terminal session:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

5. Click **Create Cluster**
6. Once the cluster has started, run the following command in a terminal session:

```
ssh ubuntu@<ec-2 driver host>.compute.amazonaws.com -p 2200 -i <path to your public/private key> -L 54321:localhost:54321
```

   This will allow you to use the Flow UI.

   (You can find the 'ec-2 driver host' information in the SSH tab of the cluster.)

Running Sparkling Water

Requirements:

- Sparkling Water Jar

Steps:

1. Create a new library containing the Sparkling Water jar.

2. Download the selected Sparkling Water version from `https://www.h2o.ai/download/`.
3. The jar file is located in the sparkling water zip file at the following location: 'jars/sparkling-water-assembly_*-all.jar'
4. Attach the Sparkling Water library to the cluster.
5. Create a new Scala notebook.
6. Create an H2O cluster inside the Spark cluster:

```
import ai.h2o.sparkling._
val conf = new H2OConf(spark)
val h2oContext = H2OContext.getOrCreate(conf)
```

You can access Flow by going to localhost:54321.

Running PySparkling

Requirements:

- PySparkling zip file
- Python Module: request
- Python Module: tabulate

Steps:

1. Create a new Python library containing the PySparkling zip file.
2. Download the selected Sparkling Water version from `https://www.h2o.ai/download/`.
3. The PySparkling zip file is located in the sparkling water zip file at the following location: 'py/h2o_pysparkling_*.zip.'
4. Create libraries for the following python modules: request, tabulate.
5. Attach the PySparkling library and python modules to the cluster.
6. Create a new python notebook.
7. Create an H2O cluster inside the Spark cluster:

```
from pysparkling import *
conf = H2OConf(spark)
hc = H2OContext.getOrCreate(conf)
```

Running RSparkling

Steps:

1. Create a new R notebook
2. Create an H2O cluster inside the Spark cluster:

```
install.packages("sparklyr")
# Install H2O
install.packages("h2o", type = "source", repos = "
    http://h2o-release.s3.amazonaws.com/h2o/rel-
    zahradnik/7/R")
# Install RSparkling
install.packages("rsparkling", type = "source",
    repos = "http://h2o-release.s3.amazonaws.com/
    sparkling-water/spark-2.4/3.30.0.7-1-2.4/R")
# Connect to Spark
sc <- spark_connect(method = "databricks")
# Create H2OContext
hc <- H2OContext.getOrCreate()
```

Running Sparkling Water in Kubernetes

Sparkling Water can be executed inside the Kubernetes cluster. Sparkling Water supports Kubernetes since Spark version 2.4.

Before we start, please check the following:

1. Please make sure we are familiar with how to run Spark on Kubernetes at `https://spark.apache.org/docs/SUBST_SPARK_VERSION/running-on-kubernetes.html`.
2. Ensure that we have a working Kubernetes Cluster and `kubectl` installed.
3. Ensure we have `SPARK_HOME` set up to a home directory of our Spark distribution, for example of version 3.0.0.
4. Run `kubectl cluster-info` to obtain Kubernetes master URL.
5. Have internet connection so Kubernetes can download Sparkling Water docker images.
6. If we have some non-default network policies applied to the namespace where Sparkling Water is supposed to run, make sure that the following ports are exposed: all Spark ports and ports 54321 and 54322 as these are also necessary by H2O to be able to communicate.

The examples below are using the default Kubernetes namespace which we enable for Spark as:

```
kubectl create clusterrolebinding default --
    clusterrole=edit --serviceaccount=default:default
    --namespace=default
```

We can also use different namespace setup for Spark. In that case please don't forget to pass the following option to your Spark start command: `spark.kubernetes.authenticate.driver.serviceAccountName` with a value equal to the serviceName.

Internal Backend

In the internal backend of Sparkling Water, we need to pass the option `spark.scheduler.minRegisteredResourcesRatio=1` to our Spark job invocation. This ensures that Spark waits for all resources and therefore Sparkling Water will start H2O on all requested executors.

Dynamic allocation must be disabled in Spark.

Scala

Both cluster and client deployment modes of Kubernetes are supported.

To submit Scala job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
   sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=3 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
   water-app \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
name: sparkling-water-app
spec:
clusterIP: "None"
selector:
spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-
   app --restart=Never --labels spark-driver-
   selector=sparkling-water-app --image=h2oai/
   sparkling-water-scala:SUBST_SW_VERSION -- /bin
   /bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/spark-shell \
 --master "k8s://KUBERNETES_ENDPOINT" \
 --deploy-mode client \
 --conf spark.scheduler.
    minRegisteredResourcesRatio=1 \
 --conf spark.kubernetes.container.image=h2oai/
    sparkling-water-scala:SUBST_SW_VERSION \
 --conf spark.executor.instances=3 \
 --conf spark.driver.host=sparkling-water-app \
 --conf spark.kubernetes.driver.pod.name=sparkling
    -water-app
```

4. Inside the shell, run:

```
import ai.h2o.sparkling._
val hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod:

```
kubectl port-forward sparkling-water-app
   54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --
   restart=Never --labels spark-driver-selector=
   sparkling-water-app --image=h2oai/sparkling-water-
   scala:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
   sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=3 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
   water-app \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

Python

Both cluster and client deployment modes of Kubernetes are supported.

To submit Python job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=3 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
local:///opt/sparkling-water/tests/initTest.py
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
name: sparkling-water-app
spec:
clusterIP: "None"
selector:
spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-
    app --restart=Never --labels spark-driver-
    selector=sparkling-water-app --image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION -- /
    bin/bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/pyspark \
```

```
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=3 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app
```

4. Inside the shell, run:

```
from pysparkling import *
hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod as:

```
kubectl port-forward sparkling-water-app 54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --restart=Never --labels spark-driver-selector=sparkling-water-app --image=h2oai/sparkling-water-python:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=3 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
local:///opt/sparkling-water/tests/initTest.py
```

R

First, make sure that RSparkling is installed on the node we want to run RSparkling from. You can install RSparkling as:

```
# Download, install, and initialize the H2O package for R.
# In this case we are using rel-SUBST_H2O_RELEASE_NAME SUBST_H2O_BUILD_NUMBER (SUBST_H2O_VERSION)
install.packages("h2o", type = "source", repos = "http://h2o-release.s3.amazonaws.com/h2o/rel-SUBST_H2O_RELEASE_NAME/SUBST_H2O_BUILD_NUMBER/R")

# Download, install, and initialize the RSparkling
install.packages("rsparkling", type = "source", repos = "http://h2o-release.s3.amazonaws.com/sparkling-water/spark-SUBST_SPARK_MAJOR_VERSION/SUBST_SW_VERSION/R")
```

To start `2OContext` in an interactive shell, run the following code in R or RStudio:

```
library(sparklyr)
library(rsparkling)
config <- spark_config_kubernetes("k8s://KUBERNETES_ENDPOINT",
    image = "h2oai/sparkling-water-r:SUBST_SW_VERSION",
    account = "default",
    executors = 3,
    conf = list("spark.kubernetes.file.upload.path"="file:///tmp"),
    version = "SUBST_SPARK_VERSION",
    ports = c(8880, 8881, 4040, 54321))
config["spark.home"] <- Sys.getenv("SPARK_HOME")
sc <- spark_connect(config = config, spark_home = Sys.getenv("SPARK_HOME"))
hc <- H2OContext.getOrCreate()
spark_disconnect(sc)
```

You can also submit RSparkling batch job. In that case, create a file called `batch.R` with the content from the code box above and run:

```
Rscript --default-packages=methods,utils batch.R
```

Note: In the case of RSparkling, SparklyR automatically sets the Spark deployment mode and it is not possible to specify it.

Manual Mode of External Backend

Sparkling Water External backend can be also used in Kubernetes. First, we need to start an external H2O backend on Kubernetes. To achieve this, please follow the steps on the H2O on Kubernetes Documentation available at `https://h2o-release.s3.amazonaws.com/h2o/rel-zahradnik/7/docs-website/h2o-docs/welcome.html#kubernetes-integration/>` with **one important exception**. The image to be used needs to be **h2oai/sparkling-water-external-backend:3.30.0.7-1-3.0** for Sparkling Water 3.30.0.7-1 and not the base H2O image as mentioned in H2O documentation as Sparkling Water enhances the H2O image with additional dependencies.

In order for Sparkling Water to be able to connect to the H2O cluster, we need to get the address of the leader node of the H2O cluster. If we followed the H2O documentation on how to start H2O cluster on Kubernetes, the address is `h2o-service.default.svc.cluster.local:54321` where the first part is the H2O headless service name and the second part is the name of the namespace.

After we created the external H2O backend, we can connect to it from Sparkling Water clients as:

Scala

Both cluster and client deployment modes of Kubernetes are supported.

To submit Scala job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
   sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
```

```
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.cloud.representative=h2o-service.default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
name: sparkling-water-app
spec:
clusterIP: "None"
selector:
spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-app --restart=Never --labels spark-driver-selector=sparkling-water-app --image=h2oai/sparkling-water-scala:SUBST_SW_VERSION -- /bin/bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/spark-shell \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-scala:SUBST_SW_VERSION \
```

```
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.cloud.representative=h2o-service.default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root
```

4. Inside the shell, run:

```
import ai.h2o.sparkling._
val hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod:

```
kubectl port-forward sparkling-water-app 54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --restart=Never --labels spark-driver-selector=sparkling-water-app --image=h2oai/sparkling-water-scala:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
```

```
--conf spark.ext.h2o.cloud.representative=h2o-service.default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

Python

Both cluster and client deployment modes of Kubernetes are supported.

To submit Python job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.cloud.representative=h2o-service.default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root \
local:///opt/sparkling-water/tests/initTest.py
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
name: sparkling-water-app
spec:
clusterIP: "None"
selector:
```

```
spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-
    app --restart=Never --labels spark-driver-
    selector=sparkling-water-app --image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION -- /
    bin/bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/pyspark \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio
    =1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external
     \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.cloud.representative=h2o-
    service.default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root
```

4. Inside the shell, run:

```
from pysparkling import *
hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod as:

```
kubectl port-forward sparkling-water-app
    54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --
    restart=Never --labels spark-driver-selector=
    sparkling-water-app --image=h2oai/sparkling-water-
    python:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=manual \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.cloud.representative=h2o-service.
    default.svc.cluster.local:54321 \
--conf spark.ext.h2o.cloud.name=root \
local:///opt/sparkling-water/tests/initTest.py
```

R

First, make sure that RSparkling is installed on the node we want to run RSparkling from. You can install RSparkling as:

```
# Download, install, and initialize the H2O package
    for R.
# In this case we are using rel-SUBST_H2O_RELEASE_NAME
    SUBST_H2O_BUILD_NUMBER (SUBST_H2O_VERSION)
install.packages("h2o", type = "source", repos = "http
    ://h2o-release.s3.amazonaws.com/h2o/rel-SUBST_H2O_
    RELEASE_NAME/SUBST_H2O_BUILD_NUMBER/R")

# Download, install, and initialize the RSparkling
install.packages("rsparkling", type = "source", repos
    = "http://h2o-release.s3.amazonaws.com/sparkling-
```

```
    water/spark-SUBST_SPARK_MAJOR_VERSION/SUBST_SW_
    VERSION/R")
```

To start `H2OContext` in an interactive shell, run the following code in R or RStudio:

```
library(sparklyr)
library(rsparkling)
config <- spark_config_kubernetes("k8s://KUBERNETES_
    ENDPOINT",
    image = "h2oai/sparkling-water-r:SUBST_SW_VERSION"
        ,
    account = "default",
    executors = 2,
    version = "SUBST_SPARK_VERSION",
    conf = list(
        "spark.ext.h2o.backend.cluster.mode"="external
            ",
        "spark.ext.h2o.external.start.mode"="manual",
        "spark.ext.h2o.external.memory"="2G",
        "spark.ext.h2o.cloud.representative"="h2o-
            service.default.svc.cluster.local:54321",
        "spark.ext.h2o.cloud.name"="root",
        "spark.kubernetes.file.upload.path"="file:///
            tmp"),
    ports = c(8880, 8881, 4040, 54321))
config["spark.home"] <- Sys.getenv("SPARK_HOME")
sc <- spark_connect(config = config, spark_home = Sys.
    getenv("SPARK_HOME"))
hc <- H2OContext.getOrCreate()
spark_disconnect(sc)
```

You can also submit RSparkling batch job. In that case, create a file called `batch.R` with the content from the code box above and run:

```
Rscript --default-packages=methods,utils batch.R
```

Note: In the case of RSparkling, SparklyR automatically sets the Spark deployment mode and it is not possible to specify it.

Automatic Mode of External Backend

In the automatic mode, Sparkling Water starts external H2O on Kubernetes automatically. The requirement is that the driver node is configured to communicate with the Kubernetes cluster. Docker image for the external H2O backend is specified using the "spark.ext.h2o.external.k8s.docker.image" option.

Scala

Both cluster and client deployment modes of Kubernetes are supported.

To submit Scala job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=
    kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/
    sparkling-water-external-backend:SUBST_SW_VERSION
    \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
```

```
  name: sparkling-water-app
spec:
  clusterIP: "None"
  selector:
    spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-app --restart=Never --labels spark-driver-selector=sparkling-water-app --image=h2oai/sparkling-water-scala:SUBST_SW_VERSION -- /bin/bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/spark-shell \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/sparkling-water-external-backend:SUBST_SW_VERSION
```

4. Inside the shell, run:

```
import ai.h2o.sparkling._
val hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod:

```
kubectl port-forward sparkling-water-app
    54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --
    restart=Never --labels spark-driver-selector=
    sparkling-water-app --image=h2oai/sparkling-water-
    scala:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-scala:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=
    kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/
    sparkling-water-external-backend:SUBST_SW_VERSION
    \
--class ai.h2o.sparkling.KubernetesTest \
local:///opt/sparkling-water/tests/kubernetesTest.jar
```

Python

Both cluster and client deployment modes of Kubernetes are supported.

To submit Python job in a cluster mode, run:

```
$SPARK_HOME/bin/spark-submit \
```

```
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode cluster \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=
    kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/
    sparkling-water-external-backend:SUBST_SW_VERSION
    \
local:///opt/sparkling-water/tests/initTest.py
```

To start an interactive shell in a client mode:

1. Create Headless service so Spark executors can reach the driver node:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: sparkling-water-app
spec:
  clusterIP: "None"
  selector:
    spark-driver-selector: sparkling-water-app
EOF
```

2. Start pod from where we run the shell:

```
kubectl run -n default -i --tty sparkling-water-
    app --restart=Never --labels spark-driver-
    selector=sparkling-water-app --image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION -- /
    bin/bash
```

3. Inside the container, start the shell:

```
$SPARK_HOME/bin/pyspark \
--master "k8s://KUBERNETES_ENDPOINT" \
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/sparkling-water-external-backend:SUBST_SW_VERSION
```

4. Inside the shell, run:

```
from pysparkling import *
hc = H2OContext.getOrCreate()
```

5. To access flow, we need to enable port-forwarding from the driver pod as:

```
kubectl port-forward sparkling-water-app 54321:54321
```

To submit a batch job using client mode:

First, create the headless service as mentioned in the step 1 above and run:

```
kubectl run -n default -i --tty sparkling-water-app --restart=Never --labels spark-driver-selector=sparkling-water-app --image=h2oai/sparkling-water-python:SUBST_SW_VERSION -- \
$SPARK_HOME/bin/spark-submit \
--master "k8s://KUBERNETES_ENDPOINT" \
```

```
--deploy-mode client \
--conf spark.scheduler.minRegisteredResourcesRatio=1 \
--conf spark.kubernetes.container.image=h2oai/
    sparkling-water-python:SUBST_SW_VERSION \
--conf spark.executor.instances=2 \
--conf spark.driver.host=sparkling-water-app \
--conf spark.kubernetes.driver.pod.name=sparkling-
    water-app \
--conf spark.ext.h2o.backend.cluster.mode=external \
--conf spark.ext.h2o.external.start.mode=auto \
--conf spark.ext.h2o.external.auto.start.backend=
    kubernetes \
--conf spark.ext.h2o.external.cluster.size=2 \
--conf spark.ext.h2o.external.memory=2G \
--conf spark.ext.h2o.external.k8s.docker.image=h2oai/
    sparkling-water-external-backend:SUBST_SW_VERSION
    \
local:///opt/sparkling-water/tests/initTest.py
```

R

First, make sure that RSparkling is installed on the node we want to run RSparkling from. You can install RSparkling as:

```
# Download, install, and initialize the H2O package
    for R.
# In this case we are using rel-SUBST_H2O_RELEASE_NAME
     SUBST_H2O_BUILD_NUMBER (SUBST_H2O_VERSION)
install.packages("h2o", type = "source", repos = "http
    ://h2o-release.s3.amazonaws.com/h2o/rel-SUBST_H2O_
    RELEASE_NAME/SUBST_H2O_BUILD_NUMBER/R")

# Download, install, and initialize the RSparkling
install.packages("rsparkling", type = "source", repos
    = "http://h2o-release.s3.amazonaws.com/sparkling-
    water/spark-SUBST_SPARK_MAJOR_VERSION/SUBST_SW_
    VERSION/R")
```

To start `H2OContext` in an interactive shell, run the following code in R or RStudio:

```
library(sparklyr)
library(rsparkling)
config <- spark_config_kubernetes("k8s://KUBERNETES_ENDPOINT",
    image = "h2oai/sparkling-water-r:SUBST_SW_VERSION",
    account = "default",
    executors = 2,
    version = "SUBST_SPARK_VERSION",
    conf = list(
        "spark.ext.h2o.backend.cluster.mode"="external",
        "spark.ext.h2o.external.start.mode"="auto",
        "spark.ext.h2o.external.auto.start.backend"="kubernetes",
        "spark.ext.h2o.external.memory"="2G",
        "spark.ext.h2o.external.cluster.size"="2",
        "spark.ext.h2o.external.k8s.docker.image"="h2oai/sparkling-water-external-backend:SUBST_SW_VERSION",
        "spark.kubernetes.file.upload.path"="file:///tmp"),
    ports = c(8880, 8881, 4040, 54321))
config["spark.home"] <- Sys.getenv("SPARK_HOME")
sc <- spark_connect(config = config, spark_home = Sys.getenv("SPARK_HOME"))
hc <- H2OContext.getOrCreate()
spark_disconnect(sc)
```

You can also submit RSparkling batch job. In that case, create a file called 'batch.R' with the content from the code box above and run:

```
Rscript --default-packages=methods,utils batch.R
```

Note: In the case of RSparkling, SparklyR automatically sets the Spark deployment mode and it is not possible to specify it.

Sparkling Water Configuration Properties

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

Configuration Properties Independent of Selected Backend

Property name	Default	Description
spark.ext.h2o.backend.cluster.mode	internal	This option can be set either to "internal" or "external". When set to "external", "H2O Context" is created by connecting to existing H2O cluster, otherwise H2O cluster located inside Spark is created. That means that each Spark executor will have one H2O instance running in it. The "internal" mode is not recommended for big clusters and clusters where Spark executors are not stable.
spark.ext.h2o.cloud.name	None	Name of H2O cluster. If this option is not set, the name is automatically generated

spark.ext.h2o.nthreads	-1	Limit for number of threads used by H2O. Default "-1" using internal backend means: Use the value of "spark.executor.cores" if the property is set, otherwise use H2O's default value Runtime.getRuntime().availablePro Default "-1" using automatically started external backend on Hadoop means: Use H2O's default value Runtime.getRuntime().availablePro Default "-1" using automatically started external backend on Kubernetes means: Use just one cpu.
spark.ext.h2o.progressbar.enabled	true	Decides whether to display progress bar related to H2O jobs on stdout or not.
spark.ext.h2o.model.print.after.training.enabled	true	Decides whether to display model info on stdout after training or not.
spark.ext.h2o.repl.enabled	true	Decides whether H2O REPL is initiated or not.
spark.ext.scala.int.default.num	1	Number of parallel REPL sessions started at the start of Sparkling Water.
spark.ext.h2o.topology.change.listener.enabled	true	Decides whether listener which kills H2O cluster on the change of the underlying cluster's topology is enabled or not. This configuration has effect only in non-local mode.
spark.ext.h2o.spark.version.check.enabled	true	Enables check if runtime Spark version matches build time Spark version.
spark.ext.h2o.fail.on.unsupported.spark.param	true	If unsupported Spark parameter is detected, then application is forced to shutdown.

spark.ext.h2o.jks	None	Path to a Java keystore file with certificates securing H2O Flow UI and internal REST connections between instances (driver + executors) and H2O nodes. When configuring this property, you must consider that a Spark executor can communicate to any of H2O nodes and verifies H2O node according to the hostname specified in the keystore certificate. You can consider usage of a wildcard certificate or you can disable the hostname verification completely with the "spark.ext.h2o.verify_ssl_hostna property.
spark.ext.h2o.jks.pass	None	Password for the Java keystore file.
spark.ext.h2o.jks.alias	None	Alias to certificate in the to the Java keystore file to secure H2O Flow UI and internal REST connections between Spark instances (driver + executors) and H2O nodes.
spark.ext.h2o.ssl.ca.cert	None	A path to a CA bundle file or a directory with certificates of trusted CAs. This path is used by RSparkling or PySparking for connecting to a Sparkling Water backend.
spark.ext.h2o.hash.login	false	Enable hash login.
spark.ext.h2o.ldap.login	false	Enable LDAP login.
spark.ext.h2o.proxy.login.only	false	Enable proxy only login for the chosen login method.
spark.ext.h2o.kerberos.login	false	Enable Kerberos login.

spark.ext.h2o.pam.login	false	Enable PAM login. PAM has to be configured on the system where Spark driver is running.
spark.ext.h2o.login.conf	None	Login configuration file.
spark.ext.h2o.user.name	None	Username used for the backend H2O cluster and to authenticate the client against the backend.
spark.ext.h2o.password	None	Password used to authenticate the client against the backend.
spark.ext.h2o.internal_security_conf	None	Path to a file containing H2O or Sparkling Water internal security configuration.
spark.ext.h2o.auto.flow.ssl	false	Automatically generate the required key store and password to secure secure H2O Flow UI and internal REST connections between Spark executors and H2O nodes. Hostname verification is disabled when creating SSL connections to H2O nodes.
spark.ext.h2o.log.level	INFO	H2O log level.
spark.ext.h2o.log.dir	None	Location of H2O logs. When not specified, it uses user.dir/h2ologs/AppId or YARN container dir
spark.ext.h2o.backend.heartbeat.interval	10000	Interval (in msec) for getting heartbeat from the H2O backend.
spark.ext.h2o.cloud.timeout	60000	Timeout (in msec) for cluster formation.
spark.ext.h2o.node.network.mask	None	Subnet selector for H2O running inside park executors. This disables using IP reported by Spark but tries to find IP based on the specified mask.

spark.ext.h2o.stacktrace.collector.interval	-1	Interval specifying how often stack traces are taken on each H2O node. -1 means that no stack traces will be taken
spark.ext.h2o.context.path	None	Context path to expose H2O web server.
spark.ext.h2o.flow.scala.cell.async	false	Decide whether the Scala cells in H2O Flow will run synchronously or Asynchronously. Default is synchronously.
spark.ext.h2o.flow.scala.cell.max.parallel	-1	Number of max parallel Scala cell jobs. The value -1 means not limited.
spark.ext.h2o.internal.port.offset	1	Offset between the API(=web) port and the internal communication port on the client node; "api_port + port_offset = h2o_port"
spark.ext.h2o.base.port	54321	Base port used for individual H2O nodes
spark.ext.h2o.mojo.destroy.timeout	600000	If a scoring MOJO instance is not used within a Spark executor JVM for a given timeout in milliseconds, it's evicted from executor's cache. Default timeout value is 10 minutes.
spark.ext.h2o.extra.properties	None	A string containing extra parameters passed to H2O nodes during startup. This parameter should be configured only if H2O parameters do not have any corresponding parameters in Sparkling Water.
spark.ext.h2o.flow.dir	None	Directory where flows from H2O Flow are saved.

spark.ext.h2o.flow.extra.http.headers	None	Extra HTTP headers that will be used in communication between the front-end and back-end part of Flow UI. The headers should be delimited by a new line. Don't forget to escape special characters when passing the parameter from a command line. Example: ""spark.ext.h2o.flow.extra.http.Transport-Security:max-age=31536000" "
spark.ext.h2o.flow.proxy.request.maxSize	32768	The maximum size of a request coming to flow UI proxy running on the Spark driver. The request is forwarded to Flow UI on H2O leader node.
spark.ext.h2o.flow.proxy.response.maxSize	32768	The maximum size of a response coming from flow UI proxy running on the Spark driver. The content for the response comes from Flow UI H2O leader node.
spark.ext.h2o.internal_secure_connections	false	Enables secure communications among H2O nodes. The security is based on automatically generated keystore and truststore. This is equivalent for "-internal_secure_conections" option in H2O Hadoop. More information is available in the H2O documentation.
spark.ext.h2o.allow_insecure_xgboost	false	If the property set to true, insecure communication among H2O nodes is allowed for the XGBoost algorithm even if the other security options are enabled
spark.ext.h2o.client.ip	None	IP of H2O client node.

spark.ext.h2o.client.web.port	-1	Exact client port to access web UI. The value "-1" means automatic search for a free port starting at "spark.ext.h2o.base.port".
spark.ext.h2o.client.verbose	false	The client outputs verbose log output directly into console. Enabling the flag increases the client log level to "INFO".
spark.ext.h2o.client.network.mask	None	Subnet selector for H2O client, this disables using IP reported by Spark but tries to find IP based on the specified mask.
spark.ext.h2o.client.flow.baseurl.override	None	Allows to override the base URL address of Flow UI, including the scheme, which is showed to the user.
spark.ext.h2o.cluster.client.retry.timeout	60000	Timeout in milliseconds specifying how often we check whether the the client is still connected.
spark.ext.h2o.verify_ssl_certificates	true	If the property is enabled, Pysparkling or RSparkling client will verify certificates when connecting Sparkling Water Flow UI.
spark.ext.h2o.internal.rest.verify_ssl_certificates	true	If the property is enabled, Sparkling Water will verify ssl certificates during establishing secured http connections to one of H2O nodes. Such connections are utilized for delegation of Flow UI calls to H2O leader node or during data exchange between Spark executors and H2O nodes. If the property is disabled, hostname verification is disabled as well.

spark.ext.h2o.internal.rest.verify_ssl_hostnames	true	If the property is enabled, Sparkling Water will verify a hostname during establishing of secured http connections to one of H2O nodes. Such connections are utilized for delegation of Flow UI calls to H2O leader node or during data exchange between Spark executors and H2O nodes.
spark.ext.h2o.kerberized.hive.enabled	false	If enabled, H2O instances will create JDBC connections to a Kerberized Hive so that all clients can read data from HiveServer2. Don't forget to put a jar with Hive driver on Spark classpath if the internal backend is used.
spark.ext.h2o.hive.host	None	The full address of HiveServer2, for example hostname:10000.
spark.ext.h2o.hive.principal	None	Hiveserver2 Kerberos principal, for example hive/hostname@DOMAIN.COM
spark.ext.h2o.hive.jdbc_url_pattern	None	A pattern of JDBC URL used for connecting to Hiveserver2. Example: "jdbc:hive2://host/;auth"
spark.ext.h2o.hive.token	None	An authorization token to Hive.
spark.ext.h2o.iced.dir	None	Location of iced directory for H2O nodes.
spark.ext.h2o.rest.api.timeout	300000	Timeout in milliseconds for Rest API requests.

Internal Backend Configuration Properties

Property name	Default	Description

spark.ext.h2o.cluster.size	None	Expected number of workers of H2O cluster. Value None means automatic detection of cluster size. This number must be equal to number of Spark executors. If Spark property "spark.executor.instances" is specified, this Sparkling Water property is set to its value.
spark.ext.h2o.extra.cluster.nodes	false	If the property is set true and the Sparkling Water internal backend identifies more executors than specified in the Spark property "spark.executor.instances" or in the Sparkling Water property "spark.ext.h2o.cluster.size", Sparkling Water deploys H2O nodes to all discovered Spark executors. Otherwise, Sparkling Water deploys just a number of executors specified in "spark.ext.h2o.cluster.size" (or "spark.executor.instances").
spark.ext.h2o.dummy.rdd.mul.factor	10	Multiplication factor for dummy RDD generation. Size of dummy RDD is "spark.ext.h2o.cluster.size" multiplied by this option.
spark.ext.h2o.spreadrdd.retries	10	Number of retries for creation of an RDD spread across all existing Spark executors
spark.ext.h2o.default.cluster.size	20	Starting size of cluster in case that size is not explicitly configured.
spark.ext.h2o.subseq.tries	5	Subsequent successful tries to figure out size of Spark cluster, which are producing the same number of nodes.

spark.ext.h2o.hdfs_conf	None	Either a string with the Path to a file with Hadoop HDFS configuration or the hadoop.conf.Configuration object in the org.apache package. Useful for HDFS credentials settings and other HDFS-related configurations. Default value None means use 'sc.hadoopConfig'.
spark.ext.h2o.spreadrdd.retries.timeout	0	Specifies how long the discovering of Spark executors should last. This option has precedence over other options influencing the discovery mechanism. That means that as long as the timeout hasn't expired, we keep trying to discover new executors. This option might be useful in environments where Spark executors might join the cloud with some delays.
spark.ext.h2o.direct.configuration.ip	true	If the property is disabled, Spark executor doesn't assign its IP address to H2O node directly. The IP address is suggested to H2O node and its bootstrap logic performs additional network interface availability checks before the IP is assigned to the node.
spark.ext.h2o.jetty.aes.login.module.key	None	Specific to water.webserver.jetty9.LdapAesEncrypte AES CBC Key
spark.ext.h2o.jetty.aes.login.module.iv	None	Specific to water.webserver.jetty9.LdapAesEncrypte AES CBC IV. When no IV is provided an all zero IV is used.

External Backend Configuration Properties

Property name	Default	Description
spark.ext.h2o.external.driver.if	None	Ip address or network of mapper-⟩driver callback interface. Default value means automatic detection.
spark.ext.h2o.external.driver.port	None	Port of mapper-⟩driver callback interface. Default value means automatic detection.
spark.ext.h2o.external.driver.port.range	None	Range portX-portY of mapper-⟩driver callback interface; eg: 50000-55000.
spark.ext.h2o.external.extra.memory.percent	10	This option is a percentage of external memory option and specifies memory for internal JVM use outside of Java heap.
spark.ext.h2o.cloud.representative	None	ip:port of a H2O cluster leader node to identify external H2O cluster.
spark.ext.h2o.external.cluster.size	None	Number of H2O nodes to start when "auto" mode of the external backend is set.
spark.ext.h2o.cluster.start.timeout	120	Timeout in seconds for starting H2O external cluster
spark.ext.h2o.cluster.info.name	None	Full path to a file which is used as the notification file for the startup of external H2O cluster.
spark.ext.h2o.external.memory	6G	Amount of memory assigned to each external H2O node
spark.ext.h2o.external.hdfs.dir	None	Path to the directory on HDFS used for storing temporary files.

spark.ext.h2o.external.start.mode	manual	If this option is set to "auto" then H2O external cluster is automatically started using the provided H2O driver JAR on YARN, otherwise it is expected that the cluster is started by the user manually
spark.ext.h2o.external.h2o.driver	None	Path to H2O driver used during "auto" start mode.
spark.ext.h2o.external.yarn.queue	None	Yarn queue on which external H2O cluster is started.
spark.ext.h2o.external.kill.on.unhealthy	true	If true, the client will try to kill the cluster and then itself in case some nodes in the cluster report unhealthy status.
spark.ext.h2o.external.kerberos.principal	None	Kerberos Principal
spark.ext.h2o.external.kerberos.keytab	None	Kerberos Keytab
spark.ext.h2o.external.run.as.user	None	Impersonated Hadoop user
spark.ext.h2o.external.backend.stop.timeout	10000	Timeout for confirmation from worker nodes when stopping the external backend. It is also possible to pass "-1" to ensure the indefinite timeout. The unit is milliseconds.
spark.ext.h2o.external.hadoop.executable	hadoop	Name or path to path to a hadoop executable binary which is used to start external H2O backend on YARN.
spark.ext.h2o.external.extra.jars	None	Comma-separated paths to jars that will be placed onto classpath of each H2O node.
spark.ext.h2o.external.communication.compression	SNAPPY	The type of compression used for data transfer between Spark and H2O node. Possible values are "NONE", "DEFLATE", "GZIP", "SNAPPY".

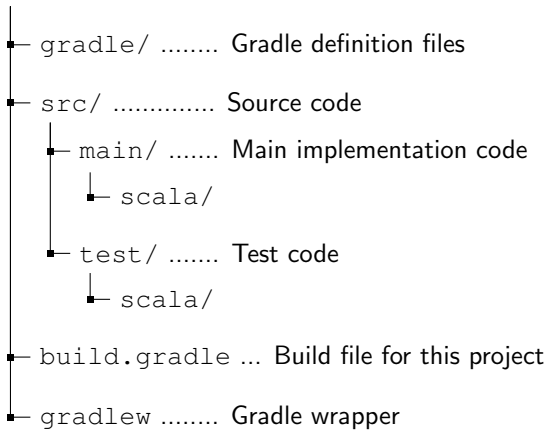
spark.ext.h2o.external.auto.start.backend	yarn	The backend on which the external H2O backend will be started in auto start mode. Possible values are "YARN" and "KUBERNETES".
spark.ext.h2o.external.k8s.h2o.service.name	h2o-service	Name of H2O service required to start H2O on K8s.
spark.ext.h2o.external.k8s.h2o.statefulset.name	h2o-statefulset	Name of H2O stateful set required to start H2O on K8s.
spark.ext.h2o.external.k8s.h2o.label	app=h2o	Label used to select node for H2O cluster formation.
spark.ext.h2o.external.k8s.h2o.api.port	8081	Kubernetes API port.
spark.ext.h2o.external.k8s.namespace	default	Kubernetes namespace where external H2O is started.
spark.ext.h2o.external.k8s.docker.image	See doc	Docker image containing Sparkling Water external H2O backend. Default value is h2oai/sparkling-water-external-backend:3.46.0.1-1-3.0
spark.ext.h2o.external.k8s.domain	cluster.local	Domain of the Kubernetes cluster.
spark.ext.h2o.external.k8s.svc.timeout	300	[Deprecated] Timeout in seconds used as a limit for K8s service creation.

Building a Standalone Application

Sparkling Water Example Project

This is a structure of a simple example project to start coding with Sparkling Water. The source code is available at `https://github.com/h2oai/h2o-droplets/tree/master/sparkling-water-droplet`

Project structure



Project building

For building, please, use provided gradlew command:

```
./gradlew build
```

Run demo

For running a simple application:

```
./gradlew run
```

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.jar` 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[*]"
$SPARK_HOME/bin/spark-submit --class water.droplets.
    SparklingWaterDroplet build/libs/sparkling-water-
    droplet-all.jar
```

A Use Case Example

Predicting Arrival Delay in Minutes - Regression

What is the task?

As a chief air traffic controller, your job is to 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.

SQL queries from Spark

One of the many cool features about the Spark is the ability to initiate a Spark Session 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 that contains the weather data at the arriving city. Joining the two tables will require a Spark Session such that we can write an INNER JOIN against the two independent `DataFrame`s.

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

Let's get started!

First, create Spark Session and H2OContext:

```
1 import org.apache.spark.SparkConf
2 import ai.h2o.sparkling._
3 import org.apache.spark.sql.SparkSession
4 val conf = new SparkConf().setAppName("Sparkling Water
    : Join of Airlines with Weather Data")
5 val spark = SparkSession.builder().config(conf).
    getOrCreate()
6 import spark.implicits._
7 val h2oContext = H2OContext.getOrCreate()
```

Read the weather data:

```
val weatherDataFile = "examples/smalldata/chicago/
    Chicago_Ohare_International_Airport.csv"
val weatherTable = spark.read.option("header", "true")
  .option("inferSchema", "true")
  .csv(weatherDataFile)
  .withColumn("Date", to_date(regexp_replace('Date,
      "(\\d+)/(\\d+)/(\\d+)", "$3-$2-$1")))
  .withColumn("Year", year('Date))
  .withColumn("Month", month('Date))
  .withColumn("DayofMonth", dayofmonth('Date))
```

Also read the airlines data:

```
val airlinesDataFile = "examples/smalldata/airlines/
    allyears2k_headers.csv"
val airlinesTable = spark.read.option("header", "true"
    )
  .option("inferSchema", "true")
  .option("nullValue", "NA")
  .csv(airlinesDataFile)
```

We load the data tables using Spark and also use Spark to do some data scrubbing.

Select flights destined for Chicago (ORD):

```
val flightsToORD = airlinesTable.filter('Dest === "ORD
    ")
println(s"\nFlights to ORD: ${flightsToORD.count}\n")
```

At this point, we are ready to join these two tables which are currently Spark DataFrames:

```
val joined = flightsToORD.join(weatherTable, Seq("Year
    ", "Month", "DayofMonth"))
```

Run deep learning to produce a model estimating arrival delay:

```
import ai.h2o.sparkling.ml.algos.H2ODeepLearning
val dl = new H2ODeepLearning()
    .setLabelCol("ArrDelay")
    .setColumnsToCategorical(Array("Year", "Month", "
       DayofMonth"))
    .setEpochs(5)
    .setActivation("RectifierWithDropout")
    .setHidden(Array(100, 100))
val model = dl.fit(joined)
```

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 predictions = model.transform(joined)
```

FAQ

Where do I find the Spark logs?

- **Standalone mode**: Spark executor logs are located in the directory `$SPARK_HOME/work/app-<AppName>` (where `<AppName>` is the name of your application). The location contains also stdout/stderr from H2O.
- **YARN mode**: YARN mode: The executor logs are available via the `$yarn logs -applicationId <appId>` command. Driver logs are by default printed to console, however, H2O also writes logs into `current_dir/h2ologs`.

The location of H2O driver logs can be controlled via the Spark property `spark.ext.h2o.client.log.dir` (pass via `--conf`) option.

How can I display Sparkling Water information in the Spark History Server?

Sparkling Water reports the information already, you just need to add the sparkling-water classes on the classpath of the Spark history server. To see how to configure the spark application for logging into the History Server, please see Spark Monitoring Configuration at `http://spark.apache.org/docs/latest/monitoring.html`.

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?

H2O accepts the `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 section 11.

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 add Apache Spark classes to Python path?

Configure the Python path variable PYTHONPATH:

```
export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/build:$PYTHONPATH
export PYTHONPATH=$SPARK_HOME/python/lib/py4j-*-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
```

Please use the following syntax to import a class from the hex package:

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

Trying to run Sparkling Water on HDP Yarn cluster, but getting error:

```
java.lang.NoClassDefFoundError: com/sun/jersey/api/client/config/ClientConfig
```

The YARN time service is not compatible with libraries provided by Spark. Please disable time service via setting `spark.hadoop.yarn.timeline-service.enabled=false`. For more details, please visit `https://issues.apache.org/jira/browse/SPARK-15343`.

How can I configure the Hive metastore location?

Spark SQL context (in fact Hive) requires the use of metastore, which stores metadata about Hive tables. In order to ensure this works correctly, the `${SPARK_HOME}/conf/hive-site.xml` needs to contain the following configuration. We provide two examples, how to use MySQL and Derby as the metastore.

For MySQL, the following configuration needs to be located in the `${SPARK_HOME}/conf/hive-site.xml` configuration file:

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://{mysql_host}:${mysql_port}/{
        metastore_db}?createDatabaseIfNotExist=true</
        value>
    <description>JDBC connect string for a JDBC
        metastore</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.jdbc.Driver</value>
    <description>Driver class name for a JDBC
        metastore</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>{username}</value>
    <description>username to use against metastore
        database</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>{password}</value>
    <description>password to use against metastore
        database</description>
</property>
```

where:

- {mysql_host} and {mysql_port} are the host and port of the MySQL database.
- {metastore_db} is the name of the MySQL database holding all the metastore tables.
- {username} and {password} are the username and password to MySQL database with read and write access to the {metastore_db} database.

For Derby, the following configuration needs to be located in the `${SPARK_HOME}/conf/hive-site.xml` configuration file:

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby://{file_location}/metastore_db;
       create=true</value>
    <description>JDBC connect string for a JDBC
       metastore</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>org.apache.derby.jdbc.ClientDriver</value>
    <description>Driver class name for a JDBC
       metastore</description>
</property>
```

where:

- {file_location} is the location to the metastore_db database file.

After conversion of Spark Data Frame to H2O Frame, I see only 100 columns on the console?

If your Spark Data Frame has more than 100 columns, we don't treat it any different. We always fully convert the Spark Data Frame to H2O Frame. We just limit the number of columns we send to the client as it's hard to read that many columns in the console plus it optimizes the amount of data we transfer betweeen the client and backend. If you wish to configure how many columns are sent to the client, you can specify it as part of the conversion method as:

```
h2o_context.asH2OFrame(dataframe, "Frame_Name", 200):
```

The last parameter specifies the number of columns to sent for the preview.

Getting exception about given invalid locale:

When getting `java.lang.reflect.InvocationTargetException` via `java.lang.IllegalArgumentException` saying that *YOUR_SPARK_ML_STAGE* parameter locale given invalid value *YOUR_LOCALE*. when using a Spark stage in my ML pipeline, set the default locale for JVM of Spark driver to a valid combination of a language and country:

```
--conf spark.driver.extraJavaOptions="-Duser.language=
   en -Duser.country=US"
```

References

H2O.ai Team. **H2O website**, 2024. URL http://h2o.ai H2O.ai Team. **H2O documentation**, 2024. URL http://docs.h2o.ai H2O.ai Team. **H2O GitHub Repository**, 2024. URL https://github.com/h2oai H2O.ai Team. **H2O Datasets**, 2024. URL http://data.h2o.ai H2O.ai Team. **H2O JIRA**, 2024. URL https://jira.h2o.ai H2O.ai Team. **H2Ostream**, 2024. URL https://groups.google.com/d/forum/h2ostream H2O.ai Team. **H2O R Package Documentation**, 2024. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html