We are excited to share that sparklyr 0.7 is now available on CRAN! Sparklyr provides an R interface to Apache Spark. It supports dplyr syntax for working with Spark DataFrames and exposes the full range of machine learning algorithms available in Spark. You can also learn more about Apache Spark .

Features in this release:

* Adds support for **ML Pipelines** which provide a uniform set of high-level APIs to help create, tune, and deploy machine learning pipelines at scale.
* Enhances **Machine Learning** capabilities by supporting the full range of ML algorithms and feature transformers.
* Improves **Data Serialization**, specifically by adding support for date columns.
* Adds support for YARN cluster mode connections.

In this blog post, we highlight Pipelines, new ML functions, and enhanced support for data serialization. To follow along in the examples below, you can upgrade to the latest stable version from CRAN with:

install.packages("sparklyr")

Installation

You can install the sparklyr package from CRAN as follows:

**install.packages**("sparklyr")

You should also install a local version of Spark for development purposes:

**library**(sparklyr)

**spark\_install**(version **=** "2.1.0")

Connecting to Spark

You can connect to both local instances of Spark as well as remote Spark clusters. Here we’ll connect to a local instance of Spark via the spark\_connect function:

**library**(sparklyr)

sc **<-** **spark\_connect**(master **=** "local")

The returned Spark connection (sc) provides a remote dplyr data source to the Spark cluster.

Using dplyr

We can now use all of the available dplyr verbs against the tables within the cluster.

We’ll start by copying some datasets from R into the Spark cluster (note that you may need to install the nycflights13 and Lahman packages in order to execute this code):

**install.packages**(**c**("nycflights13", "Lahman"))

**library**(dplyr)

iris\_tbl **<-** **copy\_to**(sc, iris)

flights\_tbl **<-** **copy\_to**(sc, nycflights13**::**flights, "flights")

batting\_tbl **<-** **copy\_to**(sc, Lahman**::**Batting, "batting")

dplyr**::src\_tbls**(sc)

## [1] "batting" "flights" "iris"

To start with here’s a simple filtering example:

*# filter by departure delay and print the first few records*

flights\_tbl **%>%** **filter**(dep\_delay **==** 2)

## # Source: lazy query [?? x 19]

## # Database: spark\_connection

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time

## <int> <int> <int> <int> <int> <dbl> <int>

## 1 2013 1 1 517 515 2 830

## 2 2013 1 1 542 540 2 923

## 3 2013 1 1 702 700 2 1058

## 4 2013 1 1 715 713 2 911

## 5 2013 1 1 752 750 2 1025

## 6 2013 1 1 917 915 2 1206

## 7 2013 1 1 932 930 2 1219

## 8 2013 1 1 1028 1026 2 1350

## 9 2013 1 1 1042 1040 2 1325

## 10 2013 1 1 1231 1229 2 1523

## # ... with more rows, and 12 more variables: sched\_arr\_time <int>,

## # arr\_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,

## # origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>, hour <dbl>,

## # minute <dbl>, time\_hour <dbl>

Introduction to dplyr provides additional dplyr examples you can try. For example, consider the last example from the tutorial which plots data on flight delays:

delay **<-** flights\_tbl **%>%**

**group\_by**(tailnum) **%>%**

**summarise**(count **=** **n**(), dist **=** **mean**(distance), delay **=** **mean**(arr\_delay)) **%>%**

**filter**(count **>** 20, dist **<** 2000, **!is.na**(delay)) **%>%**

collect

*# plot delays*

**library**(ggplot2)

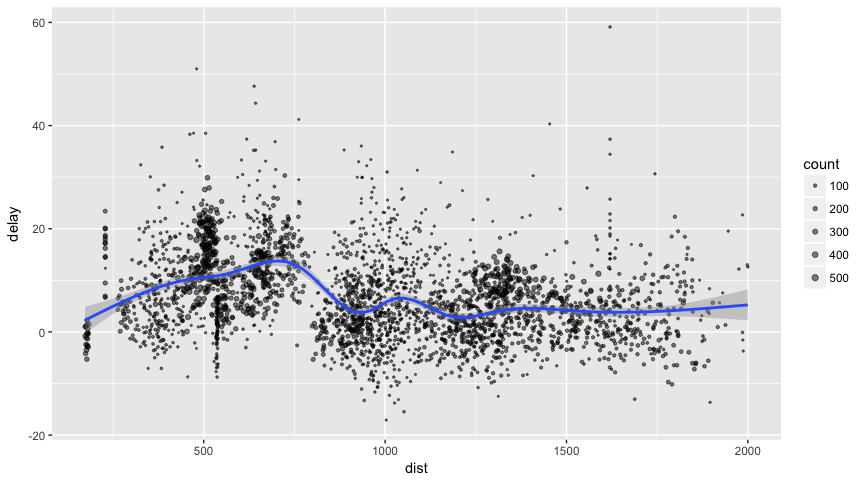
**ggplot**(delay, **aes**(dist, delay)) **+**

**geom\_point**(**aes**(size **=** count), alpha **=** 1**/**2) **+**

**geom\_smooth**() **+**

**scale\_size\_area**(max\_size **=** 2)

## `geom\_smooth()` using method = 'gam'



WINDOW FUNCTIONS

dplyr window functions are also supported, for example:

batting\_tbl **%>%**

**select**(playerID, yearID, teamID, G, AB**:**H) **%>%**

**arrange**(playerID, yearID, teamID) **%>%**

**group\_by**(playerID) **%>%**

**filter**(**min\_rank**(**desc**(H)) **<=** 2 **&** H **>** 0)

## # Source: lazy query [?? x 7]

## # Database: spark\_connection

## # Groups: playerID

## # Ordered by: playerID, yearID, teamID

## playerID yearID teamID G AB R H

## <chr> <int> <chr> <int> <int> <int> <int>

## 1 aaronha01 1959 ML1 154 629 116 223

## 2 aaronha01 1963 ML1 161 631 121 201

## 3 abbotji01 1999 MIL 20 21 0 2

## 4 abnersh01 1992 CHA 97 208 21 58

## 5 abnersh01 1990 SDN 91 184 17 45

## 6 acklefr01 1963 CHA 2 5 0 1

## 7 acklefr01 1964 CHA 3 1 0 1

## 8 adamecr01 2016 COL 121 225 25 49

## 9 adamecr01 2015 COL 26 53 4 13

## 10 adamsac01 1943 NY1 70 32 3 4

## # ... with more rows

For additional documentation on using dplyr with Spark see the dplyr section of the sparklyr website.

Using SQL

It’s also possible to execute SQL queries directly against tables within a Spark cluster. The spark\_connection object implements a DBI interface for Spark, so you can use dbGetQuery to execute SQL and return the result as an R data frame:

**library**(DBI)

iris\_preview **<-** **dbGetQuery**(sc, "SELECT \* FROM iris LIMIT 10")

iris\_preview

## Sepal\_Length Sepal\_Width Petal\_Length Petal\_Width Species

## 1 5.1 3.5 1.4 0.2 setosa

## 2 4.9 3.0 1.4 0.2 setosa

## 3 4.7 3.2 1.3 0.2 setosa

## 4 4.6 3.1 1.5 0.2 setosa

## 5 5.0 3.6 1.4 0.2 setosa

## 6 5.4 3.9 1.7 0.4 setosa

## 7 4.6 3.4 1.4 0.3 setosa

## 8 5.0 3.4 1.5 0.2 setosa

## 9 4.4 2.9 1.4 0.2 setosa

## 10 4.9 3.1 1.5 0.1 setosa

Machine Learning

You can orchestrate machine learning algorithms in a Spark cluster via the machine learning functions within sparklyr. These functions connect to a set of high-level APIs built on top of DataFrames that help you create and tune machine learning workflows.

Here’s an example where we use ml\_linear\_regression to fit a linear regression model. We’ll use the built-in mtcars dataset, and see if we can predict a car’s fuel consumption (mpg) based on its weight (wt), and the number of cylinders the engine contains (cyl). We’ll assume in each case that the relationship between mpg and each of our features is linear.

*# copy mtcars into spark*

mtcars\_tbl **<-** **copy\_to**(sc, mtcars)

*# transform our data set, and then partition into 'training', 'test'*

partitions **<-** mtcars\_tbl **%>%**

**filter**(hp **>=** 100) **%>%**

**mutate**(cyl8 **=** cyl **==** 8) **%>%**

**sdf\_partition**(training **=** 0.5, test **=** 0.5, seed **=** 1099)

*# fit a linear model to the training dataset*

fit **<-** partitions**$**training **%>%**

**ml\_linear\_regression**(response **=** "mpg", features **=** **c**("wt", "cyl"))

fit

## Call: ml\_linear\_regression.tbl\_spark(., response = "mpg", features = c("wt", "cyl"))

##

## Formula: mpg ~ wt + cyl

##

## Coefficients:

## (Intercept) wt cyl

## 33.499452 -2.818463 -0.923187

For linear regression models produced by Spark, we can use summary() to learn a bit more about the quality of our fit, and the statistical significance of each of our predictors.

**summary**(fit)

## Call: ml\_linear\_regression.tbl\_spark(., response = "mpg", features = c("wt", "cyl"))

##

## Deviance Residuals:

## Min 1Q Median 3Q Max

## -1.752 -1.134 -0.499 1.296 2.282

##

## Coefficients:

## (Intercept) wt cyl

## 33.499452 -2.818463 -0.923187

##

## R-Squared: 0.8274

## Root Mean Squared Error: 1.422

Spark machine learning supports a wide array of algorithms and feature transformations and as illustrated above it’s easy to chain these functions together with dplyr pipelines.

Reading and Writing Data

You can read and write data in CSV, JSON, and Parquet formats. Data can be stored in HDFS, S3, or on the local filesystem of cluster nodes.

temp\_csv **<-** **tempfile**(fileext **=** ".csv")

temp\_parquet **<-** **tempfile**(fileext **=** ".parquet")

temp\_json **<-** **tempfile**(fileext **=** ".json")

**spark\_write\_csv**(iris\_tbl, temp\_csv)

iris\_csv\_tbl **<-** **spark\_read\_csv**(sc, "iris\_csv", temp\_csv)

**spark\_write\_parquet**(iris\_tbl, temp\_parquet)

iris\_parquet\_tbl **<-** **spark\_read\_parquet**(sc, "iris\_parquet", temp\_parquet)

**spark\_write\_json**(iris\_tbl, temp\_json)

iris\_json\_tbl **<-** **spark\_read\_json**(sc, "iris\_json", temp\_json)

dplyr**::src\_tbls**(sc)

## [1] "batting" "flights" "iris" "iris\_csv"

## [5] "iris\_json" "iris\_parquet" "mtcars"

Distributed R

You can execute arbitrary r code across your cluster using spark\_apply. For example, we can apply rgamma over iris as follows:

**spark\_apply**(iris\_tbl, **function**(data) {

data[1**:**4] **+** **rgamma**(1,2)

})

## # Source: table<sparklyr\_tmp\_115c74acb6510> [?? x 4]

## # Database: spark\_connection

## Sepal\_Length Sepal\_Width Petal\_Length Petal\_Width

## <dbl> <dbl> <dbl> <dbl>

## 1 5.336757 3.736757 1.636757 0.4367573

## 2 5.136757 3.236757 1.636757 0.4367573

## 3 4.936757 3.436757 1.536757 0.4367573

## 4 4.836757 3.336757 1.736757 0.4367573

## 5 5.236757 3.836757 1.636757 0.4367573

## 6 5.636757 4.136757 1.936757 0.6367573

## 7 4.836757 3.636757 1.636757 0.5367573

## 8 5.236757 3.636757 1.736757 0.4367573

## 9 4.636757 3.136757 1.636757 0.4367573

## 10 5.136757 3.336757 1.736757 0.3367573

## # ... with more rows

You can also group by columns to perform an operation over each group of rows and make use of any package within the closure:

**spark\_apply**(

iris\_tbl,

**function**(e) broom**::tidy**(**lm**(Petal\_Width **~** Petal\_Length, e)),

names **=** **c**("term", "estimate", "std.error", "statistic", "p.value"),

group\_by **=** "Species"

)

## # Source: table<sparklyr\_tmp\_115c73965f30> [?? x 6]

## # Database: spark\_connection

## Species term estimate std.error statistic p.value

## <chr> <chr> <dbl> <dbl> <dbl> <dbl>

## 1 versicolor (Intercept) -0.08428835 0.16070140 -0.5245029 6.023428e-01

## 2 versicolor Petal\_Length 0.33105360 0.03750041 8.8279995 1.271916e-11

## 3 virginica (Intercept) 1.13603130 0.37936622 2.9945505 4.336312e-03

## 4 virginica Petal\_Length 0.16029696 0.06800119 2.3572668 2.253577e-02

## 5 setosa (Intercept) -0.04822033 0.12164115 -0.3964146 6.935561e-01

## 6 setosa Petal\_Length 0.20124509 0.08263253 2.4354220 1.863892e-02

Extensions

The facilities used internally by sparklyr for its dplyr and machine learning interfaces are available to extension packages. Since Spark is a general purpose cluster computing system there are many potential applications for extensions (e.g. interfaces to custom machine learning pipelines, interfaces to 3rd party Spark packages, etc.).

Here’s a simple example that wraps a Spark text file line counting function with an R function:

*# write a CSV*

tempfile **<-** **tempfile**(fileext **=** ".csv")

**write.csv**(nycflights13**::**flights, tempfile, row.names **=** FALSE, na **=** "")

*# define an R interface to Spark line counting*

count\_lines **<-** **function**(sc, path) {

**spark\_context**(sc) **%>%**

**invoke**("textFile", path, 1L) **%>%**

**invoke**("count")

}

*# call spark to count the lines of the CSV*

**count\_lines**(sc, tempfile)

## [1] 336777

To learn more about creating extensions see the Extensions section of the sparklyr website.

Table Utilities

You can cache a table into memory with:

**tbl\_cache**(sc, "batting")

and unload from memory using:

**tbl\_uncache**(sc, "batting")

Connection Utilities

You can view the Spark web console using the spark\_web function:

**spark\_web**(sc)

You can show the log using the spark\_log function:

**spark\_log**(sc, n **=** 10)

## 17/11/09 15:55:18 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 69 (/var/folders/fz/v6wfsg2x1fb1rw4f6r0x4jwm0000gn/T//RtmpyR8oP9/file115c74b94924.csv MapPartitionsRDD[258] at textFile at NativeMethodAccessorImpl.java:0) (first 15 tasks are for partitions Vector(0))

## 17/11/09 15:55:18 INFO TaskSchedulerImpl: Adding task set 69.0 with 1 tasks

## 17/11/09 15:55:18 INFO TaskSetManager: Starting task 0.0 in stage 69.0 (TID 140, localhost, executor driver, partition 0, PROCESS\_LOCAL, 4904 bytes)

## 17/11/09 15:55:18 INFO Executor: Running task 0.0 in stage 69.0 (TID 140)

## 17/11/09 15:55:18 INFO HadoopRDD: Input split: file:/var/folders/fz/v6wfsg2x1fb1rw4f6r0x4jwm0000gn/T/RtmpyR8oP9/file115c74b94924.csv:0+33313106

## 17/11/09 15:55:18 INFO Executor: Finished task 0.0 in stage 69.0 (TID 140). 832 bytes result sent to driver

## 17/11/09 15:55:18 INFO TaskSetManager: Finished task 0.0 in stage 69.0 (TID 140) in 126 ms on localhost (executor driver) (1/1)

## 17/11/09 15:55:18 INFO TaskSchedulerImpl: Removed TaskSet 69.0, whose tasks have all completed, from pool

## 17/11/09 15:55:18 INFO DAGScheduler: ResultStage 69 (count at NativeMethodAccessorImpl.java:0) finished in 0.126 s

## 17/11/09 15:55:18 INFO DAGScheduler: Job 47 finished: count at NativeMethodAccessorImpl.java:0, took 0.131380 s

Finally, we disconnect from Spark:

**spark\_disconnect**(sc)

RStudio IDE

The latest RStudio Preview Release of the RStudio IDE includes integrated support for Spark and the sparklyr package, including tools for:

* Creating and managing Spark connections
* Browsing the tables and columns of Spark DataFrames
* Previewing the first 1,000 rows of Spark DataFrames

Once you’ve installed the sparklyr package, you should find a new Spark pane within the IDE. This pane includes a New Connection dialog which can be used to make connections to local or remote Spark instances:

Once you’ve connected to Spark you’ll be able to browse the tables contained within the Spark cluster and preview Spark DataFrames using the standard RStudio data viewer:

Using H2O

rsparkling is a CRAN package from H2O that extends sparklyr to provide an interface into Sparkling Water. For instance, the following example installs, configures and runs h2o.glm:

**options**(rsparkling.sparklingwater.version **=** "2.1.14")

**library**(rsparkling)

**library**(sparklyr)

**library**(dplyr)

**library**(h2o)

sc **<-** **spark\_connect**(master **=** "local", version **=** "2.1.0")

mtcars\_tbl **<-** **copy\_to**(sc, mtcars, "mtcars")

mtcars\_h2o **<-** **as\_h2o\_frame**(sc, mtcars\_tbl, strict\_version\_check **=** FALSE)

mtcars\_glm **<-** **h2o.glm**(x **=** **c**("wt", "cyl"),

y **=** "mpg",

training\_frame **=** mtcars\_h2o,

lambda\_search **=** TRUE)

mtcars\_glm

## Model Details:

## ==============

##

## H2ORegressionModel: glm

## Model ID: GLM\_model\_R\_1510271749678\_1

## GLM Model: summary

## family link regularization

## 1 gaussian identity Elastic Net (alpha = 0.5, lambda = 0.1013 )

## lambda\_search

## 1 nlambda = 100, lambda.max = 10.132, lambda.min = 0.1013, lambda.1se = -1.0

## number\_of\_predictors\_total number\_of\_active\_predictors

## 1 2 2

## number\_of\_iterations training\_frame

## 1 100 frame\_rdd\_29\_b907d4915799eac74fb1ea60ad594bbf

##

## Coefficients: glm coefficients

## names coefficients standardized\_coefficients

## 1 Intercept 38.941654 20.090625

## 2 cyl -1.468783 -2.623132

## 3 wt -3.034558 -2.969186

##

## H2ORegressionMetrics: glm

## \*\* Reported on training data. \*\*

##

## MSE: 6.017684

## RMSE: 2.453097

## MAE: 1.940985

## RMSLE: 0.1114801

## Mean Residual Deviance : 6.017684

## R^2 : 0.8289895

## Null Deviance :1126.047

## Null D.o.F. :31

## Residual Deviance :192.5659

## Residual D.o.F. :29

## AIC :156.2425

**spark\_disconnect**(sc)

Connecting through Livy

Livy enables remote connections to Apache Spark clusters. Before connecting to Livy, you will need the connection information to an existing service running Livy. Otherwise, to test livy in your local environment, you can install it and run it locally as follows:

**livy\_install**()

**livy\_service\_start**()

To connect, use the Livy service address as master and method = "livy" in spark\_connect. Once connection completes, use sparklyr as usual, for instance:

sc **<-** **spark\_connect**(master **=** "http://localhost:8998", method **=** "livy")

**copy\_to**(sc, iris)

## # Source: table<iris> [?? x 5]

## # Database: spark\_connection

## Sepal\_Length Sepal\_Width Petal\_Length Petal\_Width Species

## <dbl> <dbl> <dbl> <dbl> <chr>

## 1 5.1 3.5 1.4 0.2 setosa

## 2 4.9 3.0 1.4 0.2 setosa

## 3 4.7 3.2 1.3 0.2 setosa

## 4 4.6 3.1 1.5 0.2 setosa

## 5 5.0 3.6 1.4 0.2 setosa

## 6 5.4 3.9 1.7 0.4 setosa

## 7 4.6 3.4 1.4 0.3 setosa

## 8 5.0 3.4 1.5 0.2 setosa

## 9 4.4 2.9 1.4 0.2 setosa

## 10 4.9 3.1 1.5 0.1 setosa

## # ... with more rows

**spark\_disconnect**(sc)

Once you are done using livy locally, you should stop this service with:

**livy\_service\_stop**()

To connect to remote livy clusters that support basic authentication connect as:

config **<-** **livy\_config**(username**=**"<username>", password**=**"<password"**>**)

sc **<-** **spark\_connect**(master **=** "<address>", method **=** "livy", config **=** config)

**spark\_disconnect**(sc)

YARN Cluster Mode

## Local mode

Local mode is an excellent way to learn and experiment with Spark. Local mode also provides a convenient development environment for analyses, reports, and applications that you plan to eventually deploy to a multi-node Spark cluster.

To work in local mode, you should **first install a version of Spark for local use**. You can do this using the spark\_install() function, for example:

### RECOMMENDED PROPERTIES

The following are the recommended Spark properties to set when connecting via R:

* **sparklyr.cores.local** - It defaults to using all of the available cores. Not a necessary property to set, unless there’s a reason to use less cores than available for a given Spark session.
* **sparklyr.shell.driver-memory** - The limit is the amount of RAM available in the computer minus what would be needed for OS operations.
* **spark.memory.fraction** - The default is set to 60% of the requested memory per executor.
* CONNECTION EXAMPLE

conf$`sparklyr.cores.local` <- 4

conf$`sparklyr.shell.driver-memory` <- "16G"

conf$spark.memory.fraction <- 0.9

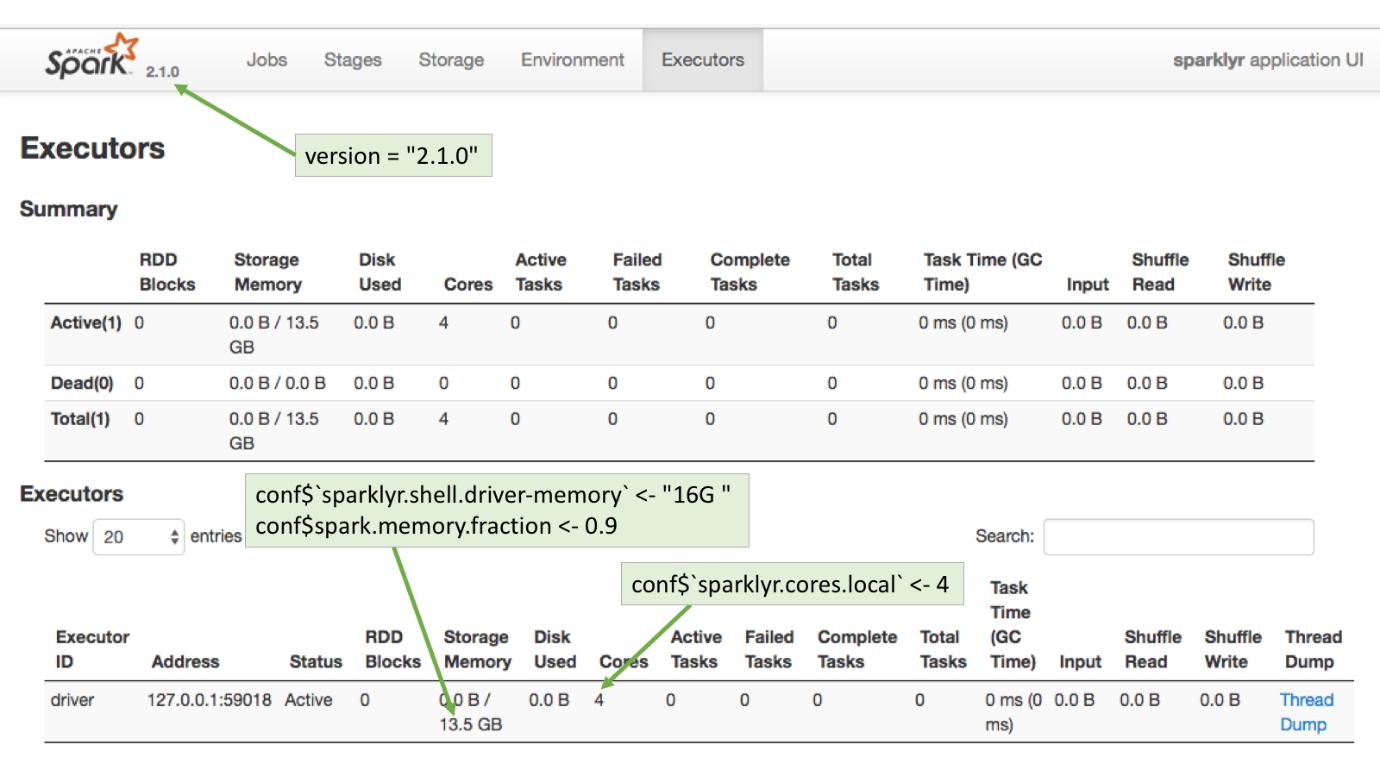
sc <- spark\_connect(master = "local",

version = "2.1.0",

config = conf)

#### **EXECUTORS PAGE**

To see how the requested configuration affected the Spark connection, go to the **Executors** page in the Spark Web UI available in http://localhost:4040/storage/



## Customizing connections

A connection to Spark can be customized by setting the values of certain Spark properties. In sparklyr, Spark properties can be set by using the config argument in the spark\_connect() function.

By default, spark\_connect() uses spark\_config() as the default configuration. But that can be customized as shown in the example code below. Because of the unending number of possible combinations, spark\_config() contains only a basic configuration, so it will be very likely that additional settings will be needed to properly connect to the cluster.

conf <- spark\_config() *# Load variable with spark\_config()*

conf$spark.executor.memory <- "16G" *# Use `$` to add or set values*

sc <- spark\_connect(master = "yarn-client",

config = conf) *# Pass the conf variable*

### SPARK DEFINITIONS

It may be useful to provide some simple definitions for the Spark nomenclature:

* **Node:** A server
* **Worker Node:** A server that is part of the cluster and are available to run Spark jobs
* **Master Node:** The server that coordinates the Worker nodes.
* **Executor:** A sort of virtual machine inside a node. **One Node can have multiple Executors**.
* **Driver Node:** The Node that initiates the Spark session. Typically, this will be the server where sparklyr is located.
* **Driver (Executor):** The Driver Node will also show up in the Executor list.

### USEFUL CONCEPTS

* **Spark configuration properties passed by R are just requests** - In most cases, the cluster has the final say regarding the resources apportioned to a given Spark session.
* **The cluster overrides ‘silently’** - Many times, no errors are returned when more resources than allowed are requested, or if an attempt is made to change a setting fixed by the cluster.

## YARN

### BACKGROUND

Using Spark and R inside a Hadoop based Data Lake is becoming a common practice at companies. Currently, there is no good way to manage user connections to the Spark service centrally. There are some caps and settings that can be applied, but in most cases there are configurations that the R user will need to customize.

The Running on YARN page in Spark’s official website is the best place to start for configuration settings reference, please bookmark it. Cluster administrators and users can benefit from this document. If Spark is new to the company, the YARN tunning article, courtesy of Cloudera, does a great job at explaining how the Spark/YARN architecture works.

### RECOMMENDED PROPERTIES

The following are the recommended Spark properties to set when connecting via R:

* **spark.executor.memory** - The maximum possible is managed by the YARN cluster.
* **spark.executor.cores** - Number of cores assigned per Executor.
* **spark.executor.instances** - Number of executors to start. This property is acknowledged by the cluster if spark.dynamicAllocation.enabled is set to “false”.
* **spark.dynamicAllocation.enabled** - Overrides the mechanism that Spark provides to dynamically adjust resources. Disabling it provides more control over the number of the Executors that can be started, which in turn impact the amount of storage available for the session.

### CLIENT MODE

Using yarn-client as the value for the master argument in spark\_connect() will make the server in which R is running to be the Spark’s session driver. Here is a sample connection:

conf <- spark\_config()

conf$spark.executor.memory <- "300M"

conf$spark.executor.cores <- 2

conf$spark.executor.instances <- 3

conf$spark.dynamicAllocation.enabled <- "false"

sc <- spark\_connect(master = "yarn-client",

spark\_home = "/usr/lib/spark/",

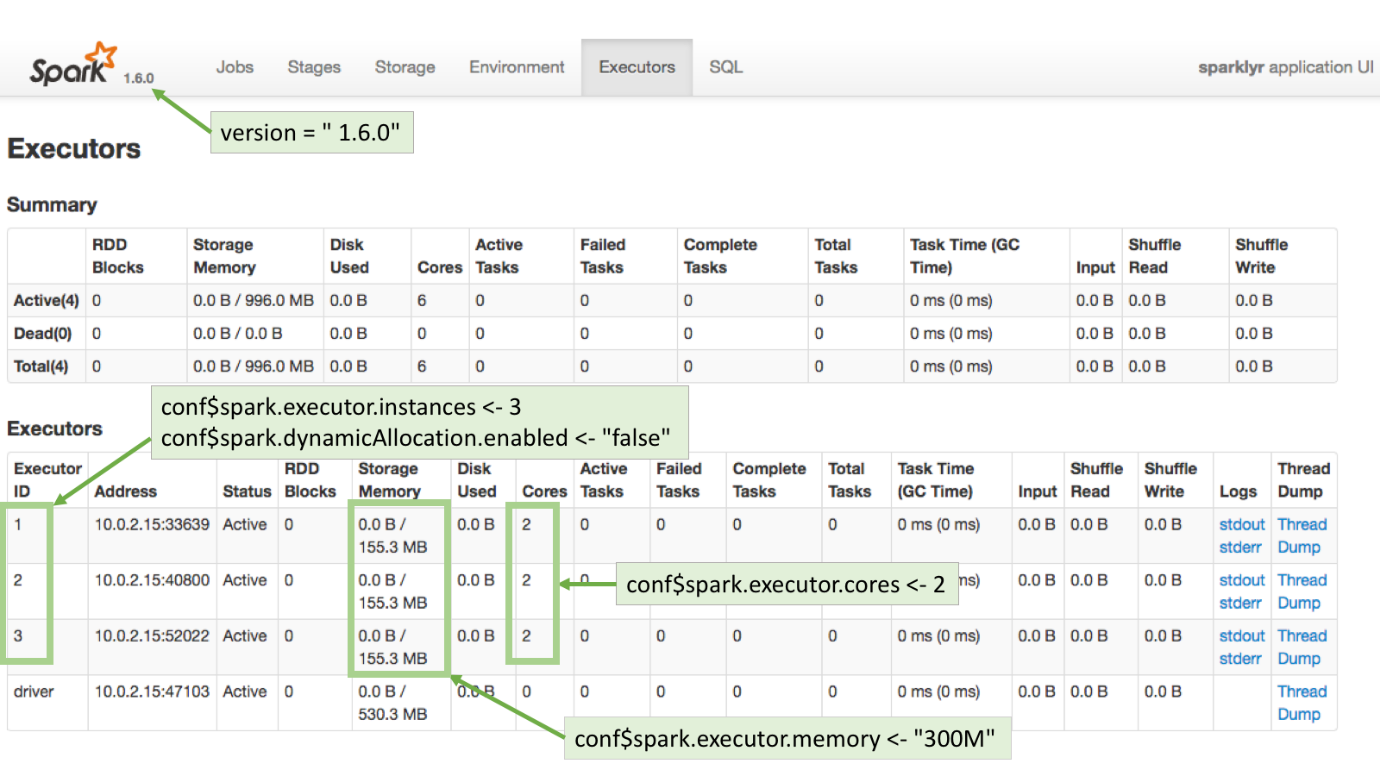
version = "1.6.0",

config = conf)

#### **EXECUTORS PAGE**

To see how the requested configuration affected the Spark connection, go to the **Executors** page in the Spark Web UI. Typically, the Spark Web UI can be found using the exact same URL used for RStudio but on port 4040.

Notice that 155.3MB per executor are assigned instead of the 300MB requested. This is because the spark.memory.fraction has been fixed by the cluster, plus, there is fixed amount of memory designated for overhead.



### CLUSTER MODE

Running in cluster mode means that YARN will choose where the driver of the Spark session will run. This means that the server where R is running may not necessarily be the driver for that session. Here is a good write-up explaining how running Spark applications work:

The server will need to have copies of at least two files: yarn-site.xml and hive-site.xml. There may be other files needed based on your cluster’s individual setup.

This is an example of connecting to a Cloudera cluster:

**library**(sparklyr)

Sys.setenv(JAVA\_HOME="/usr/lib/jvm/java-7-oracle-cloudera/")

Sys.setenv(SPARK\_HOME = '/opt/cloudera/parcels/CDH/lib/spark')

Sys.setenv(YARN\_CONF\_DIR = '/opt/cloudera/parcels/CDH/lib/spark/conf/yarn-conf')

conf$spark.executor.memory <- "300M"

conf$spark.executor.cores <- 2

conf$spark.executor.instances <- 3

conf$spark.dynamicAllocation.enabled <- "false"

conf <- spark\_config()

sc <- spark\_connect(master = "yarn-cluster",

config = conf)

### EXECUTOR MEMORY ERROR

Requesting more memory or CPUs for Executors than allowed will return an error. This is one of the exceptions to the cluster’s ‘silent’ overrides. It will return a message similar to this:

Failed during initialize\_connection: java.lang.IllegalArgumentException: Required executor memory (16384+1638 MB) is above the max threshold (8192 MB) of this cluster! Please check the values of 'yarn.scheduler.maximum-allocation-mb' and/or 'yarn.nodemanager.resource.memory-mb'

**A cluster’s administrator** is the only person who can make changes to the settings mentioned in the error. If the cluster is supported by a vendor, like Cloudera or Hortonworks, then the change can be made using the cluster’s web UI. Otherwise, changes to those settings are done directly in the yarn-default.xml file.

### KERBEROS

There are two options to access a “kerberized” data lake:

* Use kinit to get and cache the ticket. After kinit is installed and configured. After kinit is setup, it can used in R via a system() call prior to connecting to the cluster:

system("echo '<password>' | kinit <username>")

## Standalone mode

### RECOMMENDED PROPERTIES

The following are the recommended Spark properties to set when connecting via R:

The default behavior in Standalone mode is to create one executor per worker. So in a 3 worker node cluster, there will be 3 executors setup. The basic properties that can be set are:

* **spark.executor.memory** - The requested memory cannot exceed the actual RAM available.
* **spark.memory.fraction** - The default is set to 60% of the requested memory per executor. For more information.
* **spark.executor.cores** - The requested cores cannot be higher than the cores available in each worker.

#### **DYNAMIC ALLOCATION**

If dynamic allocation is disabled, then Spark will attempt to assign all of the available cores evenly across the cluster. The property used is **spark.dynamicAllocation.enabled**.

For example, the Standalone cluster used for this article has 3 worker nodes. Each node has 14.7GB in RAM and 4 cores. This means that there are a total of 12 cores (3 workers with 4 cores) and 44.1GB in RAM (3 workers with 14.7GB in RAM each).

If the spark.executor.cores property is set to 2, and dynamic allocation is disabled, then Spark will spawn 6 executors. The spark.executor.memory property should be set to a level that when the value is multiplied by 6 (number of executors) it will not be over total available RAM. In this case, the value can be safely set to 7GB so that the total memory requested will be 42GB, which is under the available 44.1GB.

### CONNECTION EXAMPLE

conf <- spark\_config()

conf$spark.executor.memory <- "7GB"

conf$spark.memory.fraction <- 0.9

conf$spark.executor.cores <- 2

conf$spark.dynamicAllocation.enabled <- "false"

sc <- spark\_connect(master="spark://master-url:7077",

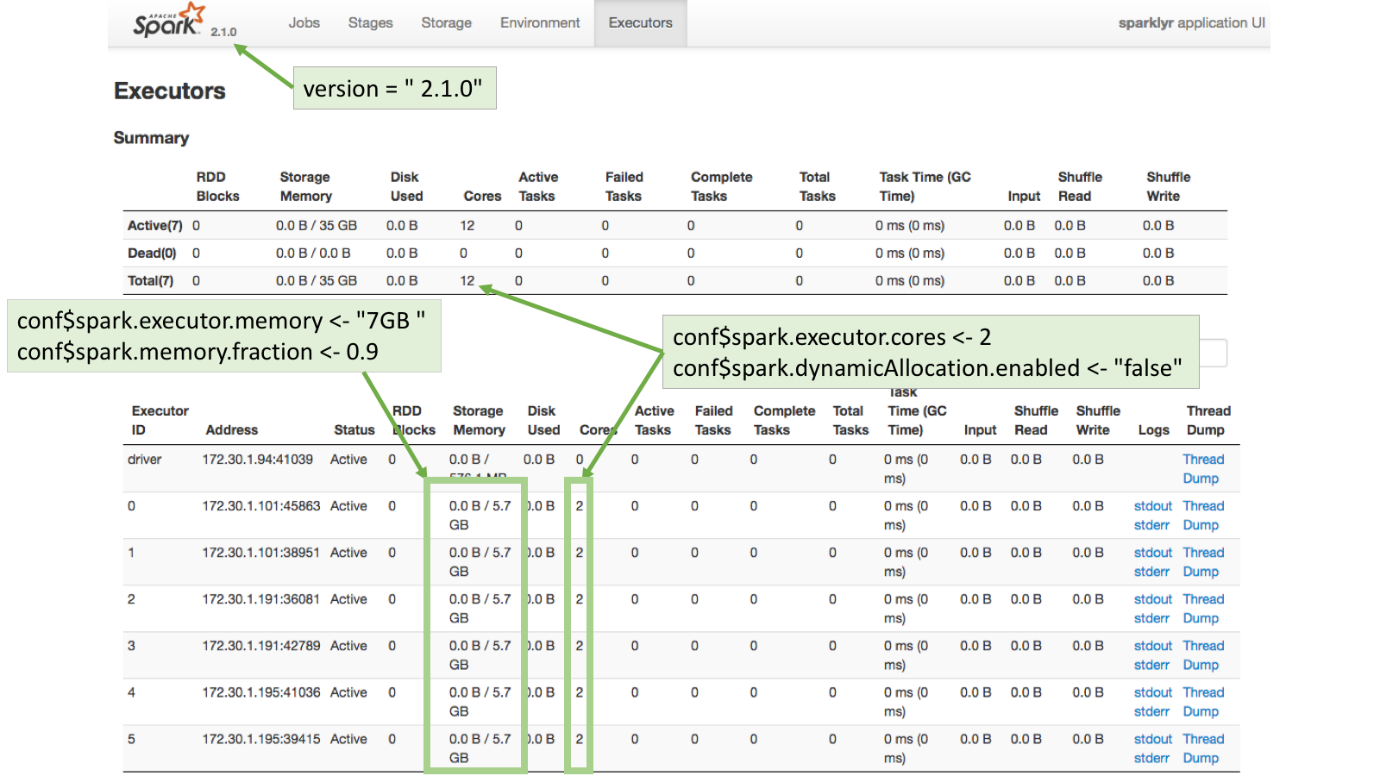
version = "2.1.0",

config = conf,

spark\_home = "/home/ubuntu/spark-2.1.0-bin-hadoop2.7/")

#### **EXECUTORS PAGE**

To see how the requested configuration affected the Spark connection, go to the **Executors** page in the Spark Web UI. Typically, the Spark Web UI can be found using the exact same URL used for RStudio but on port 4040:



Important Changes Of Sparklyr 0.7

# **Sparklyr 0.7.0**

* Added support for Spark 2.2.1.
* Switched copy\_to serializer to use Scala implementation, this change can be reverted by setting the sparklyr.copy.serializer option to csv\_file.
* Added support for spark\_web() for Livy and Databricks connections when using Spark 2.X.
* Fixed SIGPIPE error under spark\_connect() immediately after a spark\_disconnect() operation.
* spark\_web() is is more reliable under Spark 2.X by making use of a new API to programmatically find the right address.
* Added support in dbWriteTable() for temporary = FALSE to allow persisting table across connections. Changed default value for temporary to TRUE to match DBI specification, for compatibility, default value can be reverted back to FALSE using the sparklyr.dbwritetable.temp option.
* ncol() now returns the number of columns instead of NA, and nrow() now returns NA\_real\_.
* Added support to collect VectorUDT column types with nested arrays.
* Fixed issue in which connecting to Livy would fail due to long user names or long passwords.
* Fixed error in the Spark connection dialog for clusters using a proxy.
* Improved support for Spark 2.X under Cloudera clusters by prioritizing use of spark2-submit over spark-submit.
* Livy new connection dialog now prompts for password using rstudioapi::askForPassword().
* Added schema parameter to spark\_read\_parquet() that enables reading a subset of the schema to increase performance.
* Implemented sdf\_describe() to easily compute summary statistics for data frames.
* Fixed data frames with dates in spark\_apply() retrieved as Date instead of doubles.
* Added support to use invoke() with arrays of POSIXlt and POSIXct.
* Added support for context parameter in spark\_apply() to allow callers to pass additional contextual information to the f() closure.
* Implemented workaround to support in spark\_write\_table() for mode = 'append'.
* Various ML improvements, including support for pipelines, additional algorithms, hyper-parameter tuning, and better model persistence.
* Added spark\_read\_libsvm() for reading libsvm files.
* Added support for separating struct columns in sdf\_separate\_column().
* Fixed collection of short, float and byte to properly return NAs.
* Added sparklyr.collect.datechars option to enable collecting DateType and TimestampTime as characters to support compatibility with previos versions.
* Fixed collection of DateType and TimestampTime from character to proper Date and POSIXct types.

**ML Pipelines**

The ML Pipelines API is a high-level interface for building ML workflows in Spark. Pipelines provide a uniform approach to compose feature transformers and ML routines, and are interoperable across the different Spark APIs (R/sparklyr, Scala, and Python.)

**import** **org.apache.spark.ml.classification.LogisticRegression**

**import** **org.apache.spark.ml.linalg.**{**Vector**, **Vectors**}

**import** **org.apache.spark.ml.param.ParamMap**

**import** **org.apache.spark.sql.Row**

*// Prepare training data from a list of (label, features) tuples.*

**val** training **=** spark.createDataFrame(**Seq**(

(1.0, Vectors.dense(0.0, 1.1, 0.1)),

(0.0, Vectors.dense(2.0, 1.0, -1.0)),

(0.0, Vectors.dense(2.0, 1.3, 1.0)),

(1.0, Vectors.dense(0.0, 1.2, -0.5))

)).toDF("label", "features")

*// Create a LogisticRegression instance. This instance is an Estimator.*

**val** lr **=** **new** **LogisticRegression**()

*// Print out the parameters, documentation, and any default values.*

println(s"LogisticRegression parameters:\n ${lr.explainParams()}\n")

*// We may set parameters using setter methods.*

lr.setMaxIter(10)

.setRegParam(0.01)

*// Learn a LogisticRegression model. This uses the parameters stored in lr.*

**val** model1 **=** lr.fit(training)

*// Since model1 is a Model (i.e., a Transformer produced by an Estimator),*

*// we can view the parameters it used during fit().*

*// This prints the parameter (name: value) pairs, where names are unique IDs for this*

*// LogisticRegression instance.*

println(s"Model 1 was fit using parameters: ${model1.parent.extractParamMap}")

*// We may alternatively specify parameters using a ParamMap,*

*// which supports several methods for specifying parameters.*

**val** paramMap **=** **ParamMap**(lr.maxIter -> 20)

.put(lr.maxIter, 30) *// Specify 1 Param. This overwrites the original maxIter.*

.put(lr.regParam -> 0.1, lr.threshold -> 0.55) *// Specify multiple Params.*

*// One can also combine ParamMaps.*

**val** paramMap2 **=** **ParamMap**(lr.probabilityCol -> "myProbability") *// Change output column name.*

**val** paramMapCombined **=** paramMap ++ paramMap2

*// Now learn a new model using the paramMapCombined parameters.*

*// paramMapCombined overrides all parameters set earlier via lr.set\* methods.*

**val** model2 **=** lr.fit(training, paramMapCombined)

println(s"Model 2 was fit using parameters: ${model2.parent.extractParamMap}")

*// Prepare test data.*

**val** test **=** spark.createDataFrame(**Seq**(

(1.0, Vectors.dense(-1.0, 1.5, 1.3)),

(0.0, Vectors.dense(3.0, 2.0, -0.1)),

(1.0, Vectors.dense(0.0, 2.2, -1.5))

)).toDF("label", "features")

*// Make predictions on test data using the Transformer.transform() method.*

*// LogisticRegression.transform will only use the 'features' column.*

*// Note that model2.transform() outputs a 'myProbability' column instead of the usual*

*// 'probability' column since we renamed the lr.probabilityCol parameter previously.*

model2.transform(test)

.select("features", "label", "myProbability", "prediction")

.collect()

.foreach { **case** **Row**(features**:** Vector, label**:** Double, prob**:** Vector, prediction**:** Double) **=>**

println(s"($features, $label) -> prob=$prob, prediction=$prediction")

}

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/EstimatorTransformerParamExample.scala" in the Spark repo.

**Example: Pipeline**

This example follows the simple text document Pipeline illustrated in the figures above.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.**{**Pipeline**, **PipelineModel**}

**import** **org.apache.spark.ml.classification.LogisticRegression**

**import** **org.apache.spark.ml.feature.**{**HashingTF**, **Tokenizer**}

**import** **org.apache.spark.ml.linalg.Vector**

**import** **org.apache.spark.sql.Row**

*// Prepare training documents from a list of (id, text, label) tuples.*

**val** training **=** spark.createDataFrame(**Seq**(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0)

)).toDF("id", "text", "label")

*// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.*

**val** tokenizer **=** **new** **Tokenizer**()

.setInputCol("text")

.setOutputCol("words")

**val** hashingTF **=** **new** **HashingTF**()

.setNumFeatures(1000)

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

**val** lr **=** **new** **LogisticRegression**()

.setMaxIter(10)

.setRegParam(0.001)

**val** pipeline **=** **new** **Pipeline**()

.setStages(**Array**(tokenizer, hashingTF, lr))

*// Fit the pipeline to training documents.*

**val** model **=** pipeline.fit(training)

*// Now we can optionally save the fitted pipeline to disk*

model.write.overwrite().save("/tmp/spark-logistic-regression-model")

*// We can also save this unfit pipeline to disk*

pipeline.write.overwrite().save("/tmp/unfit-lr-model")

*// And load it back in during production*

**val** sameModel **=** PipelineModel.load("/tmp/spark-logistic-regression-model")

*// Prepare test documents, which are unlabeled (id, text) tuples.*

**val** test **=** spark.createDataFrame(**Seq**(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "spark hadoop spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

*// Make predictions on test documents.*

model.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { **case** **Row**(id**:** Long, text**:** String, prob**:** Vector, prediction**:** Double) **=>**

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

First, let’s go over a quick overview of terminology. A Pipeline consists of a sequence of stages—PipelineStages—that act on some data in order. A PipelineStage can be either a Transformer or an Estimator. A Transformer takes a data frame and returns a transformed data frame, whereas an Estimator take a data frame and returns a Transformer. You can think of an Estimator as an algorithm that can be fit to some data, e.g. the ordinary least squares (OLS) method, and a Transformer as the fitted model, e.g. the linear formula that results from OLS. A Pipeline is itself a PipelineStage and can be an element in another Pipeline. Lastly, a Pipeline is always an Estimator, and its fitted form is called PipelineModel which is a Transformer.

Let’s look at some examples of creating pipelines. We establish a connection and copy some data to Spark:

library(sparklyr)

library(dplyr)

# If needed, install Spark locally via `spark\_install()`

sc <- spark\_connect(master = "local")

iris\_tbl <- copy\_to(sc, iris)

# split the data into train and validation sets

iris\_data <- iris\_tbl %>%

sdf\_partition(train = 2/3, validation = 1/3, seed = 123)

Then, we can create a new Pipeline with ml\_pipeline() and add stages to it via the %>% operator. Here we also define a transformer using dplyr transformations using the newly available ft\_dplyr\_transformer().

pipeline <- ml\_pipeline(sc) %>%

ft\_dplyr\_transformer(

iris\_data$train %>%

mutate(Sepal\_Length = log(Sepal\_Length),

Sepal\_Width = Sepal\_Width ^ 2)

) %>%

ft\_string\_indexer("Species", "label")

pipeline

## Pipeline (Estimator) with 2 stages

##

## Stages

## |--1 SQLTransformer (Transformer)

## |

## | (Parameters -- Column Names)

## |--2 StringIndexer (Estimator)

## |

## | (Parameters -- Column Names)

## | input\_col: Species

## | output\_col: label

## | (Parameters)

## | handle\_invalid: error

Under the hood, ft\_dplyr\_transformer() extracts the SQL statements associated with the input and creates a Spark SQLTransformer, which can then be applied to new datasets with the appropriate columns. We now fit the Pipeline with ml\_fit() then transform some data using the resulting PipelineModel with ml\_transform().

pipeline\_model <- pipeline %>%

ml\_fit(iris\_data$train)

# pipeline\_model is a transformer

pipeline\_model %>%

ml\_transform(iris\_data$validation) %>%

glimpse()

## Observations: ??

## Variables: 6

## $ Petal\_Length 1.4, 1.3, 1.3, 1.0, 1.6, 1.9, 3.3, 4.5, 1.6, 1.5,...

## $ Petal\_Width 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 1.0, 1.7, 0.2, 0.2,...

## $ Species "setosa", "setosa", "setosa", "setosa", "setosa",...

## $ Sepal\_Length 1.482, 1.482, 1.482, 1.526, 1.548, 1.569, 1.589, ...

## $ Sepal\_Width 8.41, 9.00, 10.24, 12.96, 10.24, 11.56, 5.76, 6.2...

## $ label 1, 1, 1, 1, 1, 1, 0, 2, 1, 1, 1, 0, 1, 1, 1, 1, 1...

**A predictive modeling pipeline**

Now, let’s try to build a classification pipeline on the iris dataset.

Spark ML algorithms require that the label column be encoded as numeric and predictor columns be encoded as one vector column. We’ll build on the pipeline we created in the previous section, where we have already included a StringIndexer stage to encode the label column.

# define stages

# vector\_assember will concatenate the predictor columns into one vector column

vector\_assembler <- ft\_vector\_assembler(

sc,

input\_cols = setdiff(colnames(iris\_data$train), "Species"),

output\_col = "features"

)

logistic\_regression <- ml\_logistic\_regression(sc)

# obtain the labels from the fitted StringIndexerModel

labels <- pipeline\_model %>%

ml\_stage("string\_indexer") %>%

ml\_labels()

# IndexToString will convert the predicted numeric values back to class labels

index\_to\_string <- ft\_index\_to\_string(sc, "prediction", "predicted\_label",

labels = labels)

# construct a pipeline with these stages

prediction\_pipeline <- ml\_pipeline(

pipeline, # pipeline from previous section

vector\_assembler,

logistic\_regression,

index\_to\_string

)

# fit to data and make some predictions

prediction\_model <- prediction\_pipeline %>%

ml\_fit(iris\_data$train)

predictions <- prediction\_model %>%

ml\_transform(iris\_data$validation)

predictions %>%

select(Species, label:predicted\_label) %>%

glimpse()

## Observations: ??

## Variables: 7

## $ Species "setosa", "setosa", "setosa", "setosa", "setos...

## $ label 1, 1, 1, 1, 1, 1, 0, 2, 1, 1, 1, 0, 1, 1, 1, 1...

## $ features [<1.482, 8.410, 1.400, 0.200>, <1.482, 9.000,...

## $ rawPrediction [<-67.48, 2170.98, -2103.49>, <-124.4, 2365.8...

## $ probability [<0, 1, 0>, <0, 1, 0>, <0, 1, 0>, <0, 1, 0>, ...

## $ prediction 1, 1, 1, 1, 1, 1, 0, 2, 1, 1, 1, 0, 1, 1, 1, 1...

## $ predicted\_label "setosa", "setosa", "setosa", "setosa", "setos...

**Model persistence**

Another benefit of pipelines is reusability across programing languages and easy deployment to production. We can save a pipeline from R as follows:

ml\_save(prediction\_model, "path/to/prediction\_model")

When you call ml\_save() on a Pipeline or PipelineModel object, all of the information required to recreate it will be saved to disk. You can then load it in the future to, in the case of a PipelineModel, make predictions or, in the case of a Pipeline, retrain on new data.

**Machine learning**

Sparklyr 0.7 introduces more than 20 new feature transformation and machine learning functions to include the full set of Spark ML algorithms. We highlight just a couple here.

**Bisecting K-means**

Bisecting k-means is a variant of k-means that can sometimes be much faster to train. Here we show how to use ml\_bisecting\_kmeans() with iris data.

library(ggplot2)

model <- ml\_bisecting\_kmeans(iris\_tbl, Species ~ Petal\_Length + Petal\_Width, k = 3, seed = 123)

predictions <- ml\_predict(model, iris\_tbl) %>%

collect() %>%

mutate(cluster = as.factor(prediction))

ggplot(predictions, aes(

x = Petal\_Length,

y = Petal\_Width,

color = predictions$cluster)

) +

geom\_point()

**Frequent pattern mining**

ml\_fpgrowth() enables frequent pattern mining at scale using the FP-Growth algorithm.. Here we briefly showcase the sparklyr API.

# create an item purchase history dataset

items <- data.frame(items = c("1,2,5", "1,2,3,5", "1,2"),

stringsAsFactors = FALSE)

# parse into vector column

items\_tbl <- copy\_to(sc, items) %>%

mutate(items = split(items, ","))

# fit the model

fp\_model <- items\_tbl %>%

ml\_fpgrowth(min\_support = 0.5, min\_confidence = 0.6)

# use the model to predict related items based on

# learned association rules

fp\_model %>%

ml\_transform(items\_tbl) %>%

collect() %>%

mutate\_all(function(x) sapply(x, paste0, collapse = ","))

## # A tibble: 3 x 2

## items prediction

##

## 1 1,2,5 ""

## 2 1,2,3,5 ""

## 3 1,2 5

**import** **org.apache.spark.ml.fpm.FPGrowth**

**val** dataset **=** spark.createDataset(**Seq**(

"1 2 5",

"1 2 3 5",

"1 2")

).map(t **=>** t.split(" ")).toDF("items")

**val** fpgrowth **=** **new** **FPGrowth**().setItemsCol("items").setMinSupport(0.5).setMinConfidence(0.6)

**val** model **=** fpgrowth.fit(dataset)

*// Display frequent itemsets.*

model.freqItemsets.show()

*// Display generated association rules.*

model.associationRules.show()

*// transform examines the input items against all the association rules and summarize the*

*// consequents as prediction*

model.transform(dataset).show()

**Data serialization**

Various improvements were made to better support serialization and collection of data frames. Most notably, dates are now supported:

copy\_to(sc, nycflights13::flights) %>%

select(carrier, flight, time\_hour)

## # Source: lazy query [?? x 3]

## # Database: spark\_connection

## carrier flight time\_hour

##

## 1 UA 1545 2013-01-01 05:00:00

## 2 UA 1714 2013-01-01 05:00:00

## 3 AA 1141 2013-01-01 05:00:00

## 4 B6 725 2013-01-01 05:00:00

## 5 DL 461 2013-01-01 06:00:00

## 6 UA 1696 2013-01-01 05:00:00

## 7 B6 507 2013-01-01 06:00:00

## 8 EV 5708 2013-01-01 06:00:00

## 9 B6 79 2013-01-01 06:00:00

## 10 AA 301 2013-01-01 06:00:00

## # ... with more rows