We are excited to share that [sparklyr 0.7](https://cran.r-project.org/web/packages/sparklyr/index.html) 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 and sparklyr in [spark.rstudio.com](http://spark.rstudio.com/) and our new [webinar series on Apache Spark](https://www.rstudio.com/resources/webinars/introducing-an-r-interface-for-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](https://spark.rstudio.com/guides/connections/#cluster-mode) connections.
* Adds various other improvements as listed in the [NEWS](http://spark.rstudio.com/news/) file.

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")

**ML Pipelines**

The [ML Pipelines](https://spark.apache.org/docs/latest/ml-pipeline.html) 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.)

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](http://spark.rstudio.com/reference/#section-spark-machine-learning) 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](https://en.wikipedia.org/wiki/Association_rule_learning) at scale using the FP-Growth algorithm. See the [Spark ML documentation](https://spark.apache.org/docs/2.2.0/ml-frequent-pattern-mining.html) for more details. 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

**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