**Introduction**

Pipeline concept is definitely not new for software world, Unix pipe operator (|) links two tasks putting the output of one as the input of the other. In machine learning solutions it is pretty much usual to apply several transformation and manipulation to datasets, or to different portions or sample of the same dataset (from classic test/train slices to samples taken for cross-validation procedure). In these cases, pipelines are definitely useful to define a sequence of tasks chained together to define a complete process, which in turn simplifies the operation of the ml solution. In addition, in BigData environment, it is possible to apply the “laziness” of execution to the entire process in order to make it more scalable and robust, therefore no surprise to see pipeline implemented in Spark machine learning library and R API available, by SparklyR package, to leverage the construct. Pipeline component in Spark are basically of two types :

* **transformer:** since dataframe usually need to undergo several kinds of changes column-wide, row-wide or even single value-wide, transformers are the component meant to deliver these transformations. Typically a transformer has a table or dataframe as input and delivers a table/dataframe as output. Sparks, through MLlib, provide a [set of feature’s transformers](http://spark.apache.org/docs/latest/ml-features.html) meant to address most common transformations needed;
* **estimator:** estimator is the component which delivers a model, fitting an algorithm to train data. Indeed *fit()* is the key method for an estimator and produces, as said a *model* which is a transformer. Leveraging the parallel processing which is provided by Spark, it is possible to run several estimators in parallel on different training dataset in order to find the best solution (or even to avoid overfitting issue). *ML algorithms* are basically a set of Estimators, they build a rich set of machine learning (ML) common algorithms, available from MLlib. This is a library of algorithms meant to be scalable and run in a parallel environment. Starting from the 2.0 release of Spark, the RDD-based library is in maintenance mode (the RDD-based APIs are expected to be removed in 3.0 release) whereas the mainstream development is focused on supporting dataframes. In MLlib features are to be expressed with *labeledpoints*, which means numeric vectors for features and predictors.Pipelines of transformers are, even for this reason, extremely useful to operationalize an ML solutions Spark-based. For additional details on MLlib refer to [Apache Spark documentation](https://spark.apache.org/docs/latest/ml-guide.html)

In this post we’ll see a simple example of pipeline usage and we’ll see two version of the same example: the first one using Scala (which is a kind of “native” language for Spark environment), afterward we’ll see how to implement the same example in R, leveraging SaprklyR package in order to show how powerful and complete it is.

**The dataset**

For this example, the dataset comes from [UCI – Machine Learning Repository Irvine, CA: University of California, School of Information and Computer Science](http://archive.ics.uci.edu/ml/index.php). “Adults Income” dataset reports individual’s annual income results from various factors. The annual income will be our label (it is divided into two classes: <=50K and >50K) and there are 14 features, for each individual, we can leverage to explore the possibility in predicting income level. For additional detail on “adults dataset” refer to the UCI machine learning repository <http://www.cs.toronto.edu/~delve/data/adult/desc.html>.

**Scala code**

As said, we’ll show how we can use scala API to access pipeline in MLlib, therefore we need to include references to classes we’re planning to use in our example and to start a spark session :

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.\_

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer}

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

import org.apache.spark.sql.SparkSession

then we’ll read dataset and will start to manipulate data in order to prepare for the pipeline. In our example, we’ll get data out of local repository (instead of referring to an eg. HDFS or Datalake repository, there are API – for both scala and R – which allows the access to these repositories as well). We’ll leverage this upload activity also to rename some columns, in particular, we’ll rename the “income” column as “label” since we’ll use this a label column in our logistic regression algorithm.

//load data source from local repository

val csv = spark.read.option("inferSchema","true")

.option("header", "true").csv("...\yyyy\xxxx\adult.csv")

val data\_start = csv.select(($"workclass"),($"gender"),($"education"),($"age"),

($"marital-status").alias("marital"), ($"occupation"),($"relationship"),

($"race"),($"hours-per-week").alias("hr\_per\_week"), ($"native-country").alias("country"),

($"income").alias("label"),($"capital-gain").alias("capitalgain"),

($"capital-loss").alias("capitalloss"),($"educational-num").alias("eduyears")).toDF

We’ll do some data clean up basically recoding the “working class” and “marital” columns, in order to reduce the number of codes and we’ll get rid of rows for which “occupation”, “working class”” (even recoded) and “capital gain” are not available. For first two column the dataset has the “?” value instead of “NA”, for capital gain there’s the 99999 value which will be filtered out. To recode “working class” and “marital” columns we’ll use UDF functions which in turn are wrappers of the actual recoding functions. To add a new column to the (new) dataframe we’ll use the “withColumn” method which will add “new\_marital” and “new\_workclass” to the *startingdata* dataframe. Afterwards, we’ll filter out all missing values and we’ll be ready to build the pipeline.

// recoding marital status and working class, adding a new column

def newcol\_marital(str1:String): String = {

var nw\_str = "noVal"

if ((str1 == "Married-spouse-absent") | (str1 =="Married-AF-spouse")

| (str1 == "Married-civ-spouse")) {nw\_str = "Married" }

else if ((str1 == "Divorced") | (str1 == "Never-married" )

| (str1 == "Separated" ) | (str1 == "Widowed"))

{nw\_str = "Nonmarried" }

else { nw\_str = str1}

return nw\_str

}

val udfnewcol = udf(newcol\_marital \_)

val recodeddata = data\_start.withColumn("new\_marital", udfnewcol('marital'))

def newcol\_wkclass(str1:String): String = {

var nw\_str = "noVal"

if ((str1 == "Local-gov") | (str1 =="Federal-gov") | (str1 == "State-gov"))

{nw\_str = "Gov" }

else if ((str1 == "Self-emp-not-inc") | (str1 == "Self-emp-inc" ))

{nw\_str = "Selfemployed" }

else if ((str1 == "Never-worked") | (str1 == "Without-pay"))

{nw\_str = "Notemployed" }

else { nw\_str = str1}

return nw\_str

}

val udfnewcol = udf(newcol\_wkclass \_)

val startingdata = recodeddata.withColumn("new\_workclass", udfnewcol('workclass'))

// remove missing data

val df\_work01 = startingdata.na.drop("any")

val df\_work = startingdata.filter("occupation <> '?'

and capitalgain < 99999

and new\_workclass <> '?'

and country <> '?' ")

In our example, we’re going to use 12 features, 7 are categorical variables and 5 are numeric variables. The feature’s array we’ll use to fit the model will be the results of merging two arrays, one for categorical variables and the second one for numeric variables. Before building the categorical variables array, we need to transform categories to indexes using transformers provided by spark.ml, even the *label* must be transformed to an index. Our pipeline then will include 7 pipeline *stages* to transform categorical variables, 1 *stage* to transform the label, 2 *stages* to build categorical and numeric arrays, and the final *stage* to fit the logistic model. Finally, we’ll build an 11-stages pipeline.  
To transform categorical variables into index we’ll use “Stringindexer” *Transformer*. StringIndexer encodes a vector of string to a column of non-negative indices, ranging from 0 to the number of values. The indices ordered by label frequencies, so the most frequent value gets index 0. For each variable, we need to define the *input column* and the *output column* which we’ll use as input for other transformer or evaluators. Finally it is possible to define the strategy to handle unseen labels (possible when you use the S*tringindexer* to fit a model and run the fitted model against a test dataframe) through *setHandleInvalid* method , in our case we simply put “skip” to tell *Stringindexer* we want to skip unseen labels (additional details are available in MLlib documentation).

// define stages

val new\_workclassIdx = new StringIndexer().setInputCol("new\_workclass")

.setOutputCol("new\_workclassIdx").setHandleInvalid("skip")

val genderIdx = new StringIndexer().setInputCol("gender")

.setOutputCol("genderIdx").setHandleInvalid("skip")

val maritalIdx = new StringIndexer().setInputCol("new\_marital")

.setOutputCol("maritalIdx").setHandleInvalid("skip")

val occupationIdx = new StringIndexer().setInputCol("occupation")

.setOutputCol("occupationIdx").setHandleInvalid("skip")

val relationshipIdx = new StringIndexer().setInputCol("relationship")

.setOutputCol("relationshipIdx").setHandleInvalid("skip")

val raceIdx = new StringIndexer().setInputCol("race")

.setOutputCol("raceIdx").setHandleInvalid("skip")

val countryIdx = new StringIndexer().setInputCol("country")

.setOutputCol("countryIdx").setHandleInvalid("skip")

val labelIdx = new StringIndexer().setInputCol("label")

.setOutputCol("labelIdx").setHandleInvalid("skip")

In addition to *Transfomer* and *Estimator* provided by spark.ml package, it is possible to define custom Estimator and Transformers. As an example we’ll see how to define a custom transformer aimed at recoding “marital status” in our dataset (basically we’ll do the same task we have already seen, implementing it with a custom transformer; for additional details on implementing customer estimator and transformer see the [nice article by H.Karau](https://www.oreilly.com/learning/extend-spark-ml-for-your-own-modeltransformer-types). To define a custom transformer, we’ll define a new scala class, *columnRecoder*, which extends the Transformer class, we’ll override the *transformSchema*method to map the correct type of the new column we’re going to add with the transformation and we’ll implement the *transform* method which actually does the recoding in our dataset. A possible implementation is :

import org.apache.spark.ml.Transformer

class columnRecoder(override val uid: String) extends Transformer {

final val inputCol= new Param[String](this, "inputCol", "input column")

final val outputCol = new Param[String](this, "outputCol", "output column")

def setInputCol(value: String): this.type = set(inputCol, value)

def setOutputCol(value: String): this.type = set(outputCol, value)

def this() = this(Identifiable.randomUID("columnRecoder"))

def copy(existingParam: ParamMap): columnRecoder = {defaultCopy(existingParam)}

override def transformSchema(schema: StructType): StructType = {

// Check inputCol type

val idx = schema.fieldIndex($(inputCol))

val field = schema.fields(idx)

if (field.dataType != StringType) {

throw new Exception(s"Input type ${field.dataType} type mismatch: String expected")

}

// The return field

schema.add(StructField($(outputCol),StringType, false))

}

val newcol\_recode = new marital\_code()

private def newcol\_recode(str1: String): String = {

var nw\_str = "noVal"

if ((str1 == "Married-spouse-absent") | (str1 =="Married-AF-spouse")

| (str1 == "Married-civ-spouse"))

{nw\_str = "Married" }

else if ((str1 == "Divorced") | (str1 == "Never-married" )

| (str1 == "Separated" ) | (str1 == "Widowed"))

{nw\_str = "Nonmarried" }

else { nw\_str = str1}

nw\_str

}

private def udfnewcol = udf(newcol\_recode.recode(\_))

def transform(df: Dataset[\_]): DataFrame = {

df.withColumn($(outputCol), udfnewcol(df($(inputCol))))

}

}

Once defined as a transformer, we can use it in our pipeline as the first stage.

// define stages

val new\_marital = new columnRecoder().setInputCol("marital")

.setOutputCol("new\_marital")

val new\_workclassIdx = new StringIndexer().setInputCol("new\_workclass")

.setOutputCol("new\_workclassIdx").setHandleInvalid("skip")

val genderIdx = new StringIndexer().setInputCol("gender")

.setOutputCol("genderIdx").setHandleInvalid("skip")

val maritalIdx = new StringIndexer().setInputCol("new\_marital")

.setOutputCol("maritalIdx").setHandleInvalid("skip")

.......

A second step in building our pipeline is to assemble categorical indexes in a single vector, therefore many categorical indexes are put all together in a single vector through the *VectorAssembler*transformer. This *VectorAssembler* will deliver a single column feature which will be, in turn, transformed to indexes by *VectorIndexer* transformer to deliver indexes within the *“catFeatures”* column:

// cat vector for categorical variables

val catVect = new VectorAssembler()

.setInputCols(Array("new\_workclassIdx", "genderIdx", "catVect","maritalIdx", "occupationIdx","relationshipIdx","raceIdx","countryIdx"))

.setOutputCol("cat01Features")

val catIdx = new VectorIndexer()

.setInputCol(catVect.getOutputCol)

.setOutputCol("catFeatures")

For numeric variables we need just to assemble columns with VectorAssembler, then we’re ready to put these two vectors (one for categorical variables, the other for numeric variables) together in a single vector.

// numeric vector for numeric variable

val numVect = new VectorAssembler()

.setInputCols(Array("age","hr\_per\_week","capitalgain","capitalloss","eduyears"))

.setOutputCol("numFeatures")

val featVect = new VectorAssembler()

.setInputCols(Array("catFeatures", "numFeatures"))

.setOutputCol("features")

We have now label and features ready to build the logistic regression model which is the final component of our pipeline. We can also set some parameters for the model, in particular, we can define the threshold (by default set to 0.5) to make the decision between label values, as well as the max number of iterations for this algorithm and a parameter to tune regularization.  
When all stages of the pipeline are ready, we just need to define the *pipeline* component itself, passing as an input parameter an array with all defined stages:

val lr = new LogisticRegression().setLabelCol("labelIdx").setFeaturesCol("features")

.setThreshold(0.33).setMaxIter(10).setRegParam(0.2)

val pipeline = new Pipeline().setStages(Array(new\_marital,new\_workclassIdx, labelIdx,maritalIdx,occupationIdx, relationshipIdx,raceIdx,genderIdx, countryIdx,catVect, catIdx, numVect,featVect,lr))

Now the pipeline component, which encompasses a number of transformations as well as the classification algorithm, is ready; to actually use it we supply a *train* dataset to fit the model and then a *test* dataset to evaluate our fitted model. Since we have defined a pipeline, we’ll be sure that both, train and test datasets, will undergo the same transformations, therefore, we don’t have to replicate the process twice.  
We need now to define train and test datasets.In our dataset, label values are unbalanced being the “more than 50k USD per year” value around the 25% of the total, in order to preserve the same proportion between label values we’ll subset the original dataset based on label value, obtaining a *low-income* dataset and an *high-income* dataset. We’ll split both dataset for train (70%) and test (30%), then we’ll merge back the two “train”” and the two “test” datasets and we’ll use resulting “train” dataset as input for our pipeline:

// split betwen train and test

val df\_low\_income = df\_work.filter("label == '<=50K'")

val df\_high\_income = df\_work.filter("label == '>50K'")

val splits\_LI = df\_low\_income.randomSplit(Array(0.7, 0.3), seed=123)

val splits\_HI = df\_high\_income.randomSplit(Array(0.7, 0.3), seed=123)

val df\_work\_train = splits\_LI(0).unionAll(splits\_HI(0))

val df\_work\_test = splits\_LI(1).unionAll(splits\_HI(1))

// fitting the pipeline

val data\_model = pipeline.fit(df\_work\_train)

Once the pipeline is trained, we can use the data\_model for testing against the test dataset, calculate the confusion matrix and evaluate the classifier metrics :

// generate prediction

val data\_prediction = data\_model.transform(df\_work\_test)

val data\_predicted = data\_prediction.select("features", "prediction", "label","labelIdx")

// confusion matrix

val tp = data\_predicted.filter("prediction == 1 AND labelIdx == 1").count().toFloat

val fp = data\_predicted.filter("prediction == 1 AND labelIdx == 0").count().toFloat

val tn = data\_predicted.filter("prediction == 0 AND labelIdx == 0").count().toFloat

val fn = data\_predicted.filter("prediction == 0 AND labelIdx == 1").count().toFloat

val metrics = spark.createDataFrame(Seq(

("TP", tp),

("FP", fp),

("TN", tn),

("FN", fn),

("Precision", tp / (tp + fp)),

("Recall", tp / (tp + fn)))).toDF("metric", "value")

metrics.show()

**R code and SparklyR**

Now we’ll try to replicate the same example we just saw in R, more precisely, working with the SparklyR package.  
We’ll use the developer version of SparklyR (as you possibly know, there’s an interesting debate on the best API to access Apache Spark resources from R. Tutorial link :

, <http://blog.revolutionanalytics.com/2016/10/tutorial-scalable-r-on-spark.html>).  
We need to install it from github before connecting with Spark environment.  
In our case Spark is a standalone instance running version 2.2.0, as reported in the official documentation for SparklyR, configuration parameters can be set through spark\_config() function, in particular, spark\_config() provides the basic configuration used by default for spark connection. To change parameters it’s possible to get default configuration via spark\_connection() then change parameters as needed ( [here’s the link for additional details to run sparklyR on Yarn cluster](https://spark.rstudio.com/guides/connections/)).

library(sparklyr)

library(dplyr)

sc <- spark\_connect(master = "local",spark\_home = "...\Local\spark",version="2.2.0")

After connecting with Spark, we’ll read the dataset into a Spark dataframe and select fields (with column renaming, where needed) we’re going to use for our classification example. It is worthwhile to remember that *dplyr* (and therefore sparklyR) uses lazy evaluation when accessing and manipulating data, which means that *‘the data is only fetched at the last moment when it’s needed’* (Efficient R programming, *C.Gillespie R.Lovelace*).For this reason, later on we’ll that we’ll force the statement execution calling *action* functions (like *collect()*).  
As we did in scala script, we’ll recode *“marital status”* and *“workclass”* in order to simplify the analysis. In renaming dataframe columns, we’ll use the “select” function available from dplyr package, indeed one of the SparklyR aims is to allow the manipulation of Spark dataframes/tables through dplyr functions. This is an example of how function like “select” or “filter” can be used also for spark dataframes.

income\_table <- spark\_read\_csv(sc,"income\_table","...\adultincome\adult.csv")

income\_table <- select(income\_table,"workclass","gender","eduyears"="educationalnum",

"age","marital"="maritalstatus","occupation","relationship",

"race","hr\_per\_week"="hoursperweek","country"="nativecountry",

"label"="income","capitalgain","capitalloss")

# recoding marital status and workingclass

income\_table <- income\_table %>% mutate(marital = ifelse(marital == "Married-spouse-absent" | marital == "Married-AF-spouse" |

marital == "Married-civ-spouse","married","nonMarried"))

income\_table <- income\_table %>% mutate(workclass = ifelse(workclass == "Local-gov"| workclass == "Federal-gov" | workclass == "State\_gov",

"Gov",ifelse(workclass == "Self-emp-inc" | workclass == "Self-emp-not-inc","Selfemployed",ifelse(workclass=="Never-worked" |

workclass == "Without-pay","Notemployed",workclass))))

SparklyR provides functions which are bound to Spark spark.ml package, therefore it is possible to build Machine Learning solutions putting together the power of dplyr grammar with Spark ML algorithms. To simply link package function to Spark.ml, SparklyR provides functions which use specific prefixes to identify functions group:

* functions prefixed with **sdf\_** generally access the Scala Spark DataFrame API directly (as opposed to the dplyr interface which uses Spark SQL) to manipulate dataframes;
* functions prefixed with **ft\_** are functions to manipulate and transform features. Pipeline transformers and estimators belong to this group of functions;
* functions prefixed with **ml\_** implement algorithms to build machine learning workflow. Even pipeline instance is provided by *ml\_pipeline()* which belongs to these functions.

We can then proceed with pipeline, stages and feature array definition. Ft-prefixed functions recall the spark.ml functions these are bound to:

income\_pipe <- ml\_pipeline(sc,uid="income\_pipe")

income\_pipe <-ft\_string\_indexer(income\_pipe,input\_col="workclass",output\_col="workclassIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="gender",output\_col="genderIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="marital",output\_col="maritalIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="occupation",output\_col="occupationIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="race",output\_col="raceIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="country",output\_col="countryIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="label",output\_col="labelIdx")

array\_features <- c("workclassIdx","genderIdx","maritalIdx",

"occupationIdx","raceIdx","countryIdx","eduyears",

"age","hr\_per\_week","capitalgain","capitalloss")

income\_pipe <- ft\_vector\_assembler(income\_pipe, input\_col=array\_features, output\_col="features")

In our example we’ll use *ml\_logistic\_regression()* to implement the classification solution aimed at predicting the income level. Being bound to the *LogisticRegression()* function in spark.ml package, (as expected) all parameters are available for specific setting, therefore we can can “mirror” the same call we made in scala code: e.g. we can manage the “decision” threshold for our binary classifier (setting the value to 0.33).

# putting in pipe the logistic regression evaluator

income\_pipe <- ml\_logistic\_regression(income\_pipe,

features\_col = "features", label\_col = "labelIdx",

family= "binomial",threshold = 0.33, reg\_param=0.2, max\_iter=10L)

Final steps in our example are to split data between train and test subset, fit the pipeline and evaluate the classifier. As we’ve had already done in scala code, we’ll manage the unbalance in label values, splitting the dataframe in a way which secures the relative percentage of label values. Afterwards, we’ll fit the pipeline and get the predictions relative to test dataframe (as we did already in scala code).

# data split

# dealing with label inbalance

df\_low\_income = filter(income\_table,income\_table$label == "<=50K")

df\_high\_income = filter(income\_table,income\_table$label == ">50K")

splits\_LI <- sdf\_partition(df\_low\_income, test=0.3, train=0.7, seed = 7711)

splits\_HI <- sdf\_partition(df\_high\_income,test=0.3,train=0.7,seed=7711)

df\_test <- sdf\_bind\_rows(splits\_LI[[1]],splits\_HI[[1]])

df\_train <- sdf\_bind\_rows(splits\_LI[[2]],splits\_HI[[2]])

df\_model <- ml\_fit(income\_pipe,df\_train)

Once fitted, the model exposes, for each pipeline stage, all parameters and logistic regression (which is the last element in the stages list) coefficients.

df\_model$stages

df\_model$stages[[9]]$coefficients

We can then process the test dataset putting it in the pipeline to get predictions and evaluate the fitted model

df\_prediction <- ml\_predict(df\_model,df\_test)

df\_results <- select(df\_prediction,"prediction","labelIdx","probability")

We can then proceed to evaluate the confusion matrix and get main metrics to evaluate the model (from precision to AUC).

# calculating confusion matrix

df\_tp <- filter(df\_results,(prediction == 1 && labelIdx == 1))

df\_fp <- filter(df\_results,(prediction ==1 && labelIdx == 0))

df\_tn <- filter(df\_results,(prediction == 1 && labelIdx == 0))

df\_fn <- filter(df\_results,(prediction == 1 && labelIdx == 1))

tp <- df\_tp %>% tally() %>% collect() %>% as.integer()

fp <- df\_fp %>% tally() %>% collect() %>% as.integer()

tn <- df\_tn %>% tally() %>% collect() %>% as.integer()

fn <- df\_fn %>% tally() %>% collect() %>% as.integer()

df\_precision <- (tp/(tp+fp))

df\_recall <- (tp/(tp+fn))

df\_accuracy = (tp+tn)/(tp+tn+fp+fn)

c\_AUC <- ml\_binary\_classification\_eval(df\_prediction, label\_col = "labelIdx",

prediction\_col = "prediction", metric\_name = "areaUnderROC")

**Conclusion**

As we have seen, pipelines are a useful mechanism to assemble and serialize transformation in order to make it repeatable for different sets of data. Are then a simple way for fitting and evaluating models through train/test datasets, but also suitable to run the same sequence of transformer/estimator in parallel over different nodes of the Spark cluster (i.e. to find the best parameter set). Moreover, a powerful API to deal with pipelines in R is available by SparklyR package, which provides in addition a comprehensive set of functions to leverage the ML spark package ([here’s a link for a guide to deploy SparlyR in different cluster environment](https://spark.rstudio.com/examples/)). Finally a support to run R code distributed across a Spark cluster has been to SparklyR with the *spark\_apply()* function (<https://spark.rstudio.com/guides/distributed-r/>) which makes evenmore interesting the possibility to leverage pipelines in R in ditributed environment for analytical solutions.