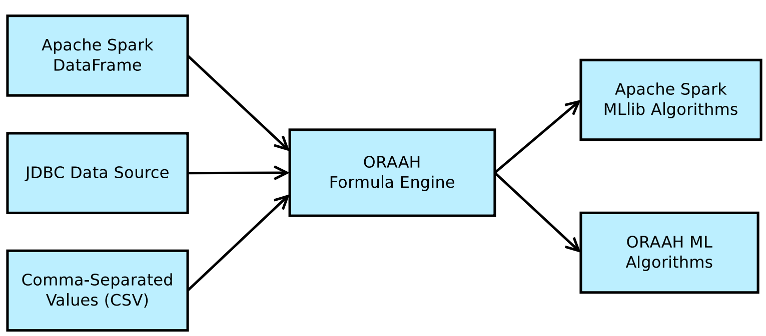
ORAAH 2.8.0 Formula and Data Preprocessing

May 14, 2018

1 Introduction

ORAAH 2.8.0 introduces new scalable data preprocessing algorithms to facilitate machine learning model building and deployment. ORAAH includes its own high performance Java implementation of the R formula to create parallel distributed model matrices, which serve as input for machine learning and statistical algorithms.

In this document we mainly use R to illustrate functionality. Please keep in mind, that both ORAAH Data Preprocessing and Formula engines are readily accessible from Java, Python, C++, R, Scala and other programming languages.



R formula is a remarkably powerful mechanism to define and create statistical and machine learning models. It is used to specify response and explanatory variables, nonlinear transformations, and interactions.

Here is a simple example

```
Y ~ X1 + X2
```

where Y is a dependent variable (target), and X1, X2 are predictors (features). The tilde operator ~ separates response (left-hand side) from explanatory terms (right-hand side of the formula).

Consider a table `tab` with three columns `Y`, `X1`, and `X2`. `Y` is a binary variable (only takes two values). We can fit (train) a logistic regression model as follows

```
fit <- orch.glm2(formula = Y ~ X1 + X2, data = tab)
```

Omitting the response (empty left-hand side) is used to define and create unsupervised models.

```
~ X1 + X2
```

With this specification, we can train a k-means model:

```
fit <- orch.ml.kmeans(formula = ~ X1 + X2, data = tab)
```

Operations allowed in a formula object fall into two main categories:

1. Set-theoretic. In our earlier example `X1 + X2`, the plus operator `+` means to *include* variables as predictors to the algorithm for model building; there is no arithmetic summation here of any kind; the operation is precisely equivalent to how one adds elements (variables) to a set (model).

2. Arithmetic (e.g., add a number to each element of the column), where plus `+`, minus `-`, and multiply `*` operators are used in their classical arithmetic sense. The divide operator `/` can be used only in an arithmetic context. The places where these operators are understood in their arithmetic sense are *function arguments* (the most common case), response, and boolean expressions.

2 R Formula Syntax

`+` plus operator when used in set-theoretic sense means to include the variable as a predictor to the algorithm for model building (into the set). To reiterate, there is no arithmetic summation here of any kind. For instance,

```
Y ~ A + B + C
```

here `Y` is our target (response); and we also include three predictors (data columns) `A`, `B`, and `C` into the model.

`-` minus operator removes the corresponding variable from the model.

`()` parenthesis are used to group variables.

```
(X + Y + Z)
```

creates a subset, which comprises three variables X, Y, and Z.

`.` dot-character stands for all variables in the training data, except the response variable. For instance

```
Y ~ . - (X20 + X50)
```

will include all variables (notice the dot-character), except X20 and X50. Alternatively, you can use `Y ~ .  - X20 - X50`.

`:` colon operator generates and includes the interaction between the two variables.

```
A : B
```

`*` asterisk operator includes both the main effects, and the interactions between them; `A * B` is equivalent to

```
A + B + A : B
```

It is perfectly fine to generate interactions between two groups of variables, for instance `(A + B) * (X + Y)`, is equivalent to `A * X + A * Y + B * X + B * Y`. Both colon and asterisk operators are of higher precedence than add (plus) and remove (minus).

`(A1 + A2 + ... + Ak)^n` include the variables and generate all interactions up to n-way. Example: `Y` $\sim$ `(. - A)^3`. One more example

```
Y ~ (log(A) + B : Z)^2
```

is equivalent to

```
Y ~ log(A) + B : Z + log(A) : B : Z
```

`I()` identity function. Its argument will be treated in the arithmetic sense. A handy function whenever you need to scale, or carry out any nonlinear transformation. Everything between the parentheses will be treated in the arithmetic sense. For instance

```
I(log(A / 10) * B + C)
```

will create a new column, whose row elements will be

```
log(A[row] / 10 ) * B[row] + C[row]
```

Here all arithmetic operators are understood in their traditional arithmetic sense. `*` means simple element-wise multiplication.

`sin(), cos(), log(), exp(), ...` math functions (see next sections). Example:

```
log(A) ~ exp(B)
```

relational operators traditional operators, `&` is a synonym for `&&`; similarly, | is a synonym for ||.

```
A >= B
A <= B
A >  B
A <  B
A == B
A != B
A && B
A & B
A || B
A | B
```

`+1` adds intercept.

`-1` removes intercept.

`-0` adds intercept, synonym for `+1`.

`+0` removes intercept, synonym for `-1`.

`as.factor(X)` allows treating a numerical column X (often integer) as a factor. For instance, consider the classical Airline on-time performance dataset, which has among others the following columns

- Cancelled, whether the flight was cancelled, binary integer column.
- Year, integer column.
- DayOfWeek, integer column.
- DepDelay departure delay in minutes, integer column.

We can train a logistic regression model as follows:

```
Cancelled ~ as.factor(Year) + as.factor(DayOfWeek) +
 DepDelay
```

The above formula will create three predictors: Year as a factor variable, DayOfWeek as a factor variable, and DepDelay as numeric. In ORAAH you can also use a shortcut `F(x)`, which means exactly the same thing as `as.factor(x)`, it is just a little less to type.

3 Arithmetic Functions in ORAAH Formula

Recall, functions treat their arguments in an arithmetic sense, similar to the identity function `I(x)`.

The argument `x` can be a number, an arithmetic expression that evaluates to number, or a column expression that is evaluated in the arithmetic sense. For instance

```
log(Y) ~ I(X * Z * 0.1) + tan(Z)
```

in the `I(X * Z * 0.1)` expression above, the formula creates a new column whose row elements are the result of multiplication `X[row] * Z[row] * 0.1`.

Table 1: Arithmetic functions in ORAAH formula

`abs(x)`	absolute value
`acos(x)`	arc cosine
`asin(x)`	arc sine
`atan(x)`	arc tangent
`cbrt(x)`	cube root
`ceil(x)`	ceiling function. The smallest integer value (returned as double) that is greater than or equal to the argument. Returns the argument, if is already a whole number (integer).
`cos(x)`	trigonometric cosine
`cosh(x)`	hyperbolic cosine
`exp(x)`	exponent e^x
`expm1(x)`	$e^x - 1$
`floor(x)`	ceiling function, the largest integer value (returned as double) that is less than or equal to the argument. Returns the argument, if is already a whole number (integer).
`log(x)`	natural logarithm, $\ln(x)$
`log10(x)`	the base 10 logarithm
`log1p(x)`	$\ln(1 + x)$
`log2(x)`	base 2 logarithm
`rint(x)`	integer, closest to the argument
`round(x)`	integer, closest to the argument

`signum(x)`	signum function $\begin{cases} +1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x > 0 \end{cases}$
`sin(x)`	trigonometric sine
`sinh(x)`	hyperbolic sine
`sqrt(x)`	square root
`tan(x)`	trigonometric tangent
`tanh(x)`	hyperbolic tangent
`toDegrees(x)`	converts an angle measured in radians to an approximately equivalent angle measured in degrees
`toRadians(x)`	converts an angle measured in degrees to an approximately equivalent angle measured in radians.

4 Statistical Functions in ORAAH Formula

Statistical and special functions (described in the next section) can be very helpful to create new features, add noise to a dataset, etc.

Arguments `x`, `p`, `q` below must be column expressions, evaluated in an arithmetic sense. The remaining parameters (`alpha`, `beta`) must be numbers (or expressions which evaluate to numbers).

Table 2: Statistical functions in ORAAH formula

	Beta distribution
	`alpha > 0, beta > 0`
density	`dbeta(x, alpha, beta)`
cumulative density	`pbeta(q, alpha, beta)`
quantile	`qbeta(p, alpha, beta)`
random deviates	`rbeta(alpha, beta)`
	Binomial distribution
	`nTrials > 0 integer, 0 <= probability <= 1`
density	`dbinom(x, nTrials, probability)`
cumulative density	`pbinom(q, nTrials, probability)`
quantile	`qbinom(p, nTrials, probability)`

random deviates	`rbinom(nTrials, probability)`

Cauchy distribution

`scale > 0`

density	`dcauchy(x, median, scale)`
cumulative density	`pcauchy(q, median, scale)`
quantile	`qcauchy(p, median, scale)`
random deviates	`rcauchy(median, scale)`

Chi-squared distribution

`degreesOfFreedom > 0`

density	`dchisq(x, degreesOfFreedom)`
cumulative density	`pchisq(q, degreesOfFreedom)`
quantile	`qchisq(p, degreesOfFreedom)`
random deviates	`rchisq(degreesOfFreedom)`

Exponential distribution

`rate > 0`

density	`dexp(x, rate)`
cumulative density	`pexp(q, rate)`
quantile	`qexp(p, rate)`
random deviates	`rexp(rate)`

F-distribution

`numeratorDF > 0, denominatorDF > 0`

density	`df(x, numeratorDF, denominatorDF)`
cumulative density	`pf(q, numeratorDF, denominatorDF)`
quantile	`qf(p, numeratorDF, denominatorDF)`
random deviates	`rf(numeratorDF, denominatorDF)`

Gamma distribution

`shape > 0, scale > 0`

density	`dgamma(x, shape, scale)`
cumulative density	`pgamma(q, shape, scale)`
quantile	`qgamma(p, shape, scale)`
random deviates	`rgamma(shape, scale)`

Geometric distribution

	`0 < probability <= 1`
density	`dgeom(x, probability)`
cumulative density	`pgeom(q, probability)`
quantile	`qgeom(p, probability)`
random deviates	`rgeom(probability)`

Hypergeometric distribution

	`populationSize > 0`
	`0 <= nSuccesses <= populationSize`
	`0 < sampleSize <= populationSize`
density	`dhyper(x, populationSize, nSuccesses, sampleSize)`
cumulative density	`phyper(q, populationSize, nSuccesses, sampleSize)`
quantile	`qhyper(p, populationSize, nSuccesses, sampleSize)`
random deviates	`rhyper(populationSize, nSuccesses, sampleSize)`

Log-normal distribution

	`shape > 0`
density	`dlnorm(x, scale, shape)`
cumulative density	`plnorm(q, scale, shape)`
quantile	`qlnorm(p, scale, shape)`
random deviates	`rlnorm(scale, shape)`

Normal distribution

	`sd > 0`
density	`dnorm(x, mean, sd)`
cumulative density	`pnorm(q, mean, sd)`
quantile	`qnorm(p, mean, sd)`
random deviates	`rnorm(mean, sd)`

Poisson distribution

	`mean > 0`
density	`dpois(x, mean)`
cumulative density	`ppois(q, mean)`
quantile	`qpois(p, mean)`
random deviates	`rpois(mean)`

Student t-distribution

	`degreesOfFreedom > 0`

density	`dt(x, degreesOfFreedom)`
cumulative density	`pt(q, degreesOfFreedom)`
quantile	`qt(p, degreesOfFreedom)`
random deviates	`rt(degreesOfFreedom)`

Triangular distribution

`lower <= mode <= upper`

density	`dtriangular(x, lower, mode, upper)`
cumulative density	`ptriangular(q, lower, mode, upper)`
quantile	`qtriangular(p, lower, mode, upper)`
random deviates	`rtriangular(lower, mode, upper)`

Uniform distribution

`lower < upper`

density	`dunif(x, lower, upper)`
cumulative density	`punif(q, lower, upper)`
quantile	`qunif(p, lower, upper)`
random deviates	`runif(lower, upper)`

Weibull distribution

`alpha > 0, beta > 0`

density	`dweibull(x, alpha, beta)`
cumulative density	`pweibull(q, alpha, beta)`
quantile	`qweibull(p, alpha, beta)`
random deviates	`rweibull(alpha, beta)`

Pareto distribution

`scale > 0, shape > 0`

density	`dpareto(x, scale, shape)`
cumulative density	`ppareto(q, scale, shape)`
quantile	`qpareto(p, scale, shape)`
random deviates	`rpareto(scale, shape)`

5 Special Functions

Arguments of the special functions are treated in a numerical sense, and they can be numbers, expressions that evaluate to numbers, or column expressions.

Table 3: Special functions in ORAAH formula

`gamma(x)`	gamma function $\Gamma(x)$
`lgamma(x)`	natural logarithm of the gamma function $\ln(\Gamma(x))$
`digamma(x)`	digamma function, $\frac{d}{dx}\ln(\Gamma(x))$
`trigamma(x)`	trigamma function, $\frac{d^2}{dx^2}\ln(\Gamma(x))$
`lanczos(x)`	Lanczos approximation of the gamma function
`factorial(x)`	factorial $n!$
`lfactorial(x)`	natural logarithm of the factorial function $\ln(n!)$
`lbeta(a, b)`	natural logarithm of the Beta function $\ln\left(\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}\right)$
`lchoose(n, k)`	natural logarithm of the binomial coefficient $\ln\left(\frac{n!}{k!(n-k)!}\right)$

6 Scaling and Aggregate Functions

Normalization can often be of paramount importance for successful creation and deployment of machine learning models in practice. In this section we describe new functions and techniques introduced in ORAAH 2.8.0

Notation:

x data column

m number of observations (e.g. number of rows in a table)

$$\begin{aligned}
\text{mean}(x) &= \frac{1}{m}\sum_{j=1}^{m} x_j \\
\text{sd}(x) &= \sqrt{\frac{\sum_{j=1}^{m}(x_j - \text{mean}(x))^2}{m-1}} \\
\text{range}(x) &= \max(x) - \min(x) \\
\text{midrange}(x) &= \frac{\min(x) + \max(x)}{2}
\end{aligned}$$

Table 4 lists normalization techniques for the function

`orch.df.scale(data, method).`

Table 4: Scalable (Parallel-Distributed) Normalization Techniques

`method`	what it is
`"standardization"`	$\frac{x - \text{mean}(x)}{\text{sd}(x)}$
`"unitization"`	$\frac{x - \text{mean}(x)}{\text{range}(x)}$
`"unitization_zero_minimum"`	$\frac{x - \min(x)}{\text{range}(x)}$
`"normalization"`	$\frac{x - \text{midrange}(x)}{\text{range}(x)/2}$
`"normalization_2"`	$\frac{x - \text{mean}(x)}{\max(\|x - \text{mean}(x)\|)}$
`"normalization_3"`	$\frac{x - \text{mean}(x)}{\sqrt{\sum (x_j - \text{mean}(x))^2}}$
`"quotient_sd"`	$\frac{x}{\text{sd}(x)}$
`"quotient_range"`	$\frac{x}{\text{range}(x)}$
`"quotient_max"`	$\frac{x}{\max(x)}$
`"quotient_mean"`	$\frac{x}{\text{mean}(x)}$
`"quotient_sum"`	$\frac{x}{\sum x_j}$
`"quotient_sqrt_ssq"`	$\frac{x}{\sqrt{\sum x_j^2}}$

Consider an example where we create a Spark data frame from Comma-Separated Values (CSV) files; then create a summary (which would show us some basic statistics on the data frame columns); and then scale the columns.

```
# Create a CSV file; use a few columns ("GNP", "Unemployed", and "Employed")
# from the Longley's Economic Regression Data.
require(stats)
head(longley)
data <- subset(longley, select=c("GNP", "Unemployed", "Employed"))
write.csv(data, file="longley.csv", row.names=FALSE)
```

Now, our CSV dataset is ready. First, create a Spark data frame using the ORCH function:

```
  orch.df.fromCSV(path, minPartitions= -1L,
    headerPresent=TRUE, fieldSeparator=",",
    quote="\"", na="NA", verbose=TRUE)
```

`orch.df.fromCSV()` can work in a fully automated "disovery" mode, where both CSV column names, and columns types will be automatically discovered. The CSV data can be read from HDFS, a local file system (in which case the file must exist on every compute node), or any other Hadoop-compliant file system.

```
dataFrame <- orch.df.fromCSV("longley.csv")
```

produces the following data frame (recall, we saved just a few columns from the longley dataset):

```
+--------+----------+--------+
|     GNP|Unemployed|Employed|
+--------+----------+--------+
|234.289|     235.6|  60.323|
|259.426|     232.5|  61.122|
|258.054|     368.2|  60.171|
|284.599|     335.1|  61.187|
|328.975|     209.9|  63.221|
|346.999|     193.2|  63.639|
|365.385|     187.0|  64.989|
|363.112|     357.8|  63.761|
|397.469|     290.4|  66.019|
| 419.18|     282.2|  67.857|
|442.769|     293.6|  68.169|
```

```
|444.546|     468.1|  66.513|
|482.704|     381.3|  68.655|
|502.601|     393.1|  69.564|
|518.173|     480.6|  69.331|
|554.894|     400.7|  70.551|
+-------+----------+--------+
```

Its summary, calculated by `orch.df.summary(dataFrame)`, is as follows (some digits were omitted for compactness):

```
+----------+----------+-----------+-------+-------+--------------+----------+-----------------+
|  col_name|  col_type|num_missing|    min|    max|          mean|        sd|num_factor_levels|
+----------+----------+-----------+-------+-------+--------------+----------+-----------------+
|       GNP|DoubleType|          0|234.289|554.894|      387.6...| 99.394...|             null|
|Unemployed|DoubleType|          0|  187.0|  480.6|319.33124999...| 93.446...|             null|
|  Employed|DoubleType|          0| 60.171| 70.551| 65.31700000...|3.51196...|             null|
+----------+----------+-----------+-------+-------+--------------+----------+-----------------+
```

Now, let us scale the `dataFrame` via technique `"normalization_2"` (normalization in range [-1,1]):

```
scaledFrame <- orch.df.scale(dataFrame, "normalization_2")
```

This produces:

```
+--------------------+--------------------+--------------------+
|                 GNP|          Unemployed|            Employed|
+--------------------+--------------------+--------------------+
| -0.9175449109183148| -0.5192031934271204| -0.9541459686664142|
| -0.7672000116629891| -0.5384257644459941| -0.8014902560183439|
| -0.7754059710765353|  0.3030267798318026| -0.9831868551776869|
| -0.6166397956883576| 0.09777932798511839| -0.7890714558654975|
|-0.35122605302398496| -0.6785645080029448|-0.40045854031333833|
|-0.24342414889151134| -0.7821183583304262| -0.3205961024073378|
|-0.13345711552601774| -0.8205635003681736|-0.06266717615590434|
|-0.14705197394219113|  0.2385381544781617|-0.29728696981276387|
| 0.05843792953536062|-0.17939774444831974| 0.13412304165074493|
|  0.1882918543367441|-0.23024454520792126|  0.4852884982804728|
| 0.32937813466191723|  -0.159555090493353|  0.5448987390141369|
| 0.34000640716765423|  0.9224896329884122| 0.22850592281238044|
|   0.568230167591918|  0.3842576444599467|  0.6377531524646537|
|  0.6872345221482776| 0.45742743091888566|  0.8114252961406171|
|  0.7803709652880291|                 1.0|  0.7669086740542606|
|                 1.0|  0.5045537340619308|                 1.0|
+--------------------+--------------------+--------------------+
```

Its summary is as follows (some digits were omitted for compactness):

```
+----------+----------+-----------+---------+---+------+--------+-----------------+
|col_name  |col_type  |num_missing|min      |max|mean  |sd      |num_factor_levels|
+----------+----------+-----------+---------+---+------+--------+-----------------+
|GNP       |DoubleType|0          |-0.917...|1.0|-5e-17|0.594...|null             |
|Unemployed|DoubleType|0          |-0.820...|1.0| 3e-16|0.579...|null             |
|Employed  |DoubleType|0          |-0.983...|1.0|-1e-15|0.670...|null             |
+----------+----------+-----------+---------+---+------+--------+-----------------+
```

Note: `orch.df.scale()` preserves the original column names, but it changes integral column types (byte, short, integer, long), and 32-bit floating point into 64-bit double. In other words, the result of scaling is a 64-bit double column. Boolean, date, calendar, and timestamp columns are treated as strings (factor), and remain unchanged.

In addition to the `orch.df.scale()` users can use the following aggregate functions, which return a single double floating point value:

Table 5: Formula aggregate functions

function name	what it is
(`colExpr` below stands for column expression)	
`avg(colExpr)`	mean (average) value
`mean(colExpr)`	mean value (synonym for avg)
`max(colExpr)`	maximum value
`min(colExpr)`	minimum value
`sd(colExpr)`	standard deviation
`stddev(colExpr)`	standard deviation (synonym for sd)
`sum(colExpr)`	sum
`variance(colExpr)`	variance
`var(colExpr)`	variance (synonym for variance)
`kurtosis(colExpr)`	kurtosis
`skewness(colExpr)`	skewness

Each aggregate function returns a single double floating point value; and therefore, can only be used inside an arithmetic context. In fact, this is exactly how formula preprocessor works: it will compute each aggregate function value, and then literally replace aggregates with their values. For instance

```
Kyphosis ~ I( Start / max(Start) )
```

In the above example, `I()` creates an arithmetic context, where we use `max(Start)` as a denominator to scale the `Start` variable. Formula preprocessor will compute the `max(Start)` (18.0), and the final formula will become

```
Kyphosis ~ I( Start / 18.0 )
```

This value 18.0 will remain fixed (will be used for prediction / scoring).