Data and Statistics Packages

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 Contents

2 Overview

This lecture explores some of the key packages for working with data and doing statistics in Julia.

In particular, we will examine the `DataFrame` object in detail (i.e., construction, manipulation, querying, visualization, and nuances like missing data).

While Julia is not an ideal language for pure cookie-cutter statistical analysis, it has many useful packages to provide those tools as part of a more general solution.

This list is not exhaustive, and others can be found in organizations such as JuliaStats, JuliaData, and QueryVerse.

2.1 Setup

```
In [1]: using InstantiateFromURL
        # optionally add arguments to force installation: instantiate = true,
 ↪precompile = true
        github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```
In [2]: using LinearAlgebra, Statistics
        using DataFrames, RDatasets, DataFramesMeta, CategoricalArrays, Query,
 ↪VegaLite
        using GLM
```

3 DataFrames

A useful package for working with data is DataFrames.jl.

The most important data type provided is a `DataFrame`, a two dimensional array for storing heterogeneous data.

Although data can be heterogeneous within a `DataFrame`, the contents of the columns must be homogeneous (of the same type).

This is analogous to a `data.frame` in R, a `DataFrame` in Pandas (Python) or, more loosely, a spreadsheet in Excel.

There are a few different ways to create a DataFrame.

3.1 Constructing and Accessing a DataFrame

The first is to set up columns and construct a dataframe by assigning names

```
In [3]: using DataFrames, RDatasets  # RDatasets provides good standard data
  ↪examples from R

        # note use of missing
        commodities = ["crude", "gas", "gold", "silver"]
        last_price = [4.2, 11.3, 12.1, missing]
        df = DataFrame(commod = commodities, price = last_price)
```

```
Out[3]: \begin{tabular}{r|cc}
                & commod & price\\
                \hline
                & String & Float64?\\
                \hline
                1 & crude & 4.2 \\
                2 & gas & 11.3 \\
                3 & gold & 12.1 \\
                4 & silver & \emph{missing} \\
        \end{tabular}
```

Columns of the `DataFrame` can be accessed by name using `df.col`, as below

```
In [4]: df.price
```

```
Out[4]: 4-element Array{Union{Missing, Float64},1}:
          4.2
         11.3
         12.1
           missing
```

Note that the type of this array has values `Union{Missing, Float64}` since it was created with a `missing` value.

```
In [5]: df.commod
```

Out[5]:
```
4-element Array{String,1}:
 "crude"
 "gas"
 "gold"
 "silver"
```

The `DataFrames.jl` package provides a number of methods for acting on `DataFrame`'s, such as `describe`.

In [6]:
```
DataFrames.describe(df)
```

Out[6]:
```
\begin{tabular}{r|cccccccc}
	& variable & mean & min & median & max & nunique & nmissing & eltype\\
	\hline
	& Symbol & …Union & Any & …Union & Any & …Union & …Union & Type\\
	\hline
	1 & commod &  & crude &  & silver & 4 &  & String \\
	2 & price & 9.2 & 4.2 & 11.3 & 12.1 &  & 1 & Union\{Missing, Float64\} \\
\end{tabular}
```

While often data will be generated all at once, or read from a file, you can add to a `DataFrame` by providing the key parameters.

In [7]:
```
nt = (commod = "nickel", price= 5.1)
push!(df, nt)
```

Out[7]:
```
\begin{tabular}{r|cc}
	& commod & price\\
	\hline
	& String & Float64?\\
	\hline
	1 & crude & 4.2 \\
	2 & gas & 11.3 \\
	3 & gold & 12.1 \\
	4 & silver & \emph{missing} \\
	5 & nickel & 5.1 \\
\end{tabular}
```

Named tuples can also be used to construct a `DataFrame`, and have it properly deduce all types.

In [8]:
```
nt = (t = 1, col1 = 3.0)
df2 = DataFrame([nt])
push!(df2, (t=2, col1 = 4.0))
```

Out[8]:
```
\begin{tabular}{r|cc}
	& t & col1\\
	\hline
	& Int64 & Float64\\
	\hline
	1 & 1 & 3.0 \\
	2 & 2 & 4.0 \\
\end{tabular}
```

In order to modify a column, access the mutating version by the symbol `df[!, :col]`.

```
In [9]: df[!, :price]
```

```
Out[9]: 5-element Array{Union{Missing, Float64},1}:
          4.2
         11.3
         12.1
           missing
          5.1
```

Which allows modifications, like other mutating `!` functions in julia.

```
In [10]: df[!, :price] *= 2.0  # double prices
```

```
Out[10]: 5-element Array{Union{Missing, Float64},1}:
           8.4
          22.6
          24.2
            missing
          10.2
```

As discussed in the next section, note that the fundamental types, is propagated, i.e. `missing * 2 === missing`.

3.2 Working with Missing

As we discussed in fundamental types, the semantics of `missing` are that mathematical operations will not silently ignore it.

In order to allow `missing` in a column, you can create/load the `DataFrame` from a source with `missing`'s, or call `allowmissing!` on a column.

```
In [11]: allowmissing!(df2, :col1) # necessary to add in a for col1
         push!(df2, (t=3, col1 = missing))
         push!(df2, (t=4, col1 = 5.1))
```

```
Out[11]: \begin{tabular}{r|cc}
                & t & col1\\
                \hline
                & Int64 & Float64?\\
                \hline
                1 & 1 & 3.0 \\
                2 & 2 & 4.0 \\
                3 & 3 & \emph{missing} \\
                4 & 4 & 5.1 \\
         \end{tabular}
```

We can see the propagation of `missing` to caller functions, as well as a way to efficiently calculate with non-missing data.

```
In [12]: @show mean(df2.col1)
         @show mean(skipmissing(df2.col1))
```

```
    mean(df2.col1) = missing
mean(skipmissing(df2.col1)) = 4.033333333333333
```

```
Out[12]: 4.033333333333333
```

And to replace the `missing`

```
In [13]: df2.col1  .= coalesce.(df2.col1, 0.0) # replace all missing with 0.0
```

```
Out[13]: 4-element Array{Union{Missing, Float64},1}:
          3.0
          4.0
          0.0
          5.1
```

3.3 Manipulating and Transforming DataFrames

One way to do an additional calculation with a `DataFrame` is to tuse the `@transform` macro from `DataFramesMeta.jl`.

```
In [14]: using DataFramesMeta
         f(x) = x^2
         df2 = @transform(df2, col2 = f.(:col1))
```

```
Out[14]: \begin{tabular}{r|ccc}
                 & t & col1 & col2\\
                 \hline
                 & Int64 & Float64? & Float64\\
                 \hline
                 1 & 1 & 3.0 & 9.0 \\
                 2 & 2 & 4.0 & 16.0 \\
                 3 & 3 & 0.0 & 0.0 \\
                 4 & 4 & 5.1 & 26.01 \\
         \end{tabular}
```

3.4 Categorical Data

For data that is categorical

```
In [15]: using CategoricalArrays
         id = [1, 2, 3, 4]
         y = ["old", "young", "young", "old"]
         y = CategoricalArray(y)
         df = DataFrame(id=id, y=y)
```

```
Out[15]: \begin{tabular}{r|cc}
                 & id & y\\
                 \hline
                 & Int64 & …Cat\\
                 \hline
```

```
        1 & 1 & old \\
        2 & 2 & young \\
        3 & 3 & young \\
        4 & 4 & old \\
\end{tabular}
```

In [16]:
```julia
levels(df.y)
```

Out[16]:
```
2-element Array{String,1}:
 "old"
 "young"
```

3.5 Visualization, Querying, and Plots

The `DataFrame` (and similar types that fulfill a standard generic interface) can fit into a variety of packages.

One set of them is the QueryVerse.

Note: The QueryVerse, in the same spirit as R's tidyverse, makes heavy use of the pipeline syntax `|>`.

In [17]:
```julia
x = 3.0
f(x) = x^2
g(x) = log(x)

@show g(f(x))
@show x |> f |> g; # pipes nest function calls
```

```
   g(f(x)) = 2.1972245773362196
(x |> f) |> g = 2.1972245773362196
```

To give an example directly from the source of the LINQ inspired Query.jl

In [18]:
```julia
using Query

df = DataFrame(name=["John", "Sally", "Kirk"], age=[23., 42., 59.],
↪children=[3,5,2])

x = @from i in df begin
    @where i.age>50
    @select {i.name, i.children}
    @collect DataFrame
end
```

Out[18]:
```
\begin{tabular}{r|cc}
        & name & children\\
        \hline
        & String & Int64\\
        \hline
        1 & Kirk & 2 \\
\end{tabular}
```

While it is possible to just use the `Plots.jl` library, there may be better options for displaying tabular data – such as VegaLite.jl.

In [19]:
```
using RDatasets, VegaLite
iris = dataset("datasets", "iris")

iris |> @vlplot(
    :point,
    x=:PetalLength,
    y=:PetalWidth,
    color=:Species
)
```

```
   WARN Missing type for channel "color", using "nominal" instead.
WARN Missing type for channel "color", using "nominal" instead.
```

Out[19]:

4 Statistics and Econometrics

While Julia is not intended as a replacement for R, Stata, and similar specialty languages, it has a growing number of packages aimed at statistics and econometrics.

Many of the packages live in the JuliaStats organization.

A few to point out

- StatsBase has basic statistical functions such as geometric and harmonic means, autocorrelations, robust statistics, etc.
- StatsFuns has a variety of mathematical functions and constants such as pdf and cdf of many distributions, softmax, etc.

4.1 General Linear Models

To run linear regressions and similar statistics, use the GLM package.

In [20]:
```
using GLM

x = randn(100)
y = 0.9 .* x + 0.5 * rand(100)
df = DataFrame(x=x, y=y)
ols = lm(@formula(y ~ x), df) # R-style notation
```

Out[20]:
```
StatsModels.TableRegressionModel{LinearModel{GLM.
↪LmResp{Array{Float64,1}},GLM.DensePredC
hol{Float64,Cholesky{Float64,Array{Float64,2}}}},Array{Float64,2}}

y ~ 1 + x

Coefficients:
────────────────────────────────────────────────────────────────────────────
             Estimate  Std. Error  t value  Pr(>|t|)  Lower 95%  Upper 95%
```

(Intercept)	0.228452	0.01451	15.7445	<1e-27	0.199657	0.257247
x	0.910643	0.0130831	69.6044	<1e-84	0.88468	0.936606

To display the results in a useful tables for LaTeX and the REPL, use RegressionTables for output similar to the Stata package esttab and the R package stargazer.

```
In [21]: using RegressionTables
         regtable(ols)
         # regtable(ols,  renderSettings = latexOutput()) # for LaTex output
```

```
----------------------
                     y
              --------
                   (1)
----------------------
(Intercept)   0.228***
               (0.015)
x             0.911***
               (0.013)
----------------------
Estimator          OLS
----------------------
N                  100
R2               0.980
----------------------
```

4.2 Fixed Effects

While Julia may be overkill for estimating a simple linear regression, fixed-effects estimation with dummies for multiple variables are much more computationally intensive.

For a 2-way fixed-effect, taking the example directly from the documentation using cigarette consumption data

```
In [22]: using FixedEffectModels
         cigar = dataset("plm", "Cigar")
         cigar.StateCategorical =  categorical(cigar.State)
         cigar.YearCategorical =  categorical(cigar.Year)
         fixedeffectresults = reg(cigar, @formula(Sales ~ NDI +
 ↪fe(StateCategorical) +
         fe(YearCategorical)),
                                     weights = :Pop, Vcov.cluster(:State))
         regtable(fixedeffectresults)
```

```
---------------------------
                      Sales
                  ---------
                        (1)
---------------------------
```

NDI	-0.005***
	(0.001)
StateCategorical	Yes
YearCategorical	Yes
Estimator	OLS
N	1,380
R2	0.803