Working with Data

July 6, 2019

1 Working with Data

The purpose of this chapter is to familiarize the reader with some of the basic of working with data in Julia. As would be expected, much of the focus of this chapter is on or around dataframes, including dataframe functions. Other topic covered include categorical data, input-output (IO), and the split-apply-combine strategy.

1.1 Dataframes

A dataframe is a tabular representation of data, similar to a spreadsheet or a data matrix. As with a matrix, the observations are rows and the variables are columns. Each row is a single (vectorvalued) observation. For a single row, i.e., observation, each column represents a single realization of a variable. At this stage, it may be helppful to explicitly draw the analogy between a dataframe and the more formal notation often used in statistics and data science.

Suppose we observe n realizations $\mathbf{x}_1, \cdots, \mathbf{x}_n$ of p-dimensional random variables $\mathbf{X}_1, \cdots, \mathbf{X}_n$, where $\mathbf{X}_i = (X_{i1}, X_{i2}, \cdots, X_{ip})'$ for $i = 1, \cdots, n$. In matrix form, this can be written

$$\mathscr{X} = (\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)' = \begin{pmatrix} \mathbf{X}_1' \\ \mathbf{X}_2' \\ \vdots \\ \mathbf{X}_n' \end{pmatrix} = \begin{pmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{pmatrix}. \tag{1}$$

Now, $\mathbf{X}_i$ is called a random vector and $\mathscr{X}$ is called an $n \times p$ random matrix. A realization of $\mathscr{X}$ can be considered a data matrix. For completeness, note that a matrix $\mathbf{A}$ with all entries constant is called a constant matrix.

Consider, for example, data on the weight and height of 500 people. Let $\mathbf{x}_i = (x_{i1}, x_{i2})'$ be the associated observation fro the ith person, $i = 1, 2, \cdots, 500$, where x_{i1} represents their weight and x_{i2} represents their height. The associated data matrix is then

$$\mathscr{X} = (\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_{500})' = \begin{pmatrix} \mathrm{x}_1' \\ \mathrm{x}_2' \\ \vdots \\ \mathrm{x}_{500}' \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{500,1} & x_{500,2} \end{pmatrix} \tag{2}$$

A dataframe is a computer representation of a data matrix. In Julia, the `DataFrame` type is available through the `DataFrames.jl` package. There are several convenient features of a `DataFrame`, including:

- columns can be different Julia types;
- table cell entries can be missing;
- metadata can be associated with a `DataFrame`;
- columns can be names;
- tables can be subsetted by row, column or both.

The columns of a `DataFrame` are most ogten integers, floats or strings, an dthey are specified by Julia symbols.

```
In [1]: ## symbol versus string
        fruit = "apple"
        println("eval(:fruit): ", eval(:fruit))
```

```
eval(:fruit): apple
```

```
In [2]: println("""eval("apple"): """, eval("apple"))
```

```
eval("apple"): apple
```

In Julia, a symbol is how a vaariable name is represented as data; on the other hand, a string represents itself. Note that `df[:symbol]` is how a column is accessed with a symbol; specifically, the data in the column represented by `symbol` contained in the `DataFrame df` is being accessed. In Julia, a `DataFrame` can be built all once in multiple phases.

```
In [6]: ## Some examples with DataFrames

        using DataFrames, Distributions, StatsBase, Random

        Random.seed!(825)

        N=50

        ## Create a sample dataframe
        ## Initially the DataFrame has N rows and 3 collumns
        df1 = DataFrame(
            x1 = rand(Normal(2,1),N),
            x2 = [sample(["High","Medium","Low"],
                        pweights([0.25,0.45,0.30])) for i =1:N],
            x3 = rand(Pareto(2,1),N)
        )

        ## Add a 4th column, y, which is dependent on x3 and the level of x2
        df1[:y] = [df1[i,:x2] == "Medium" ? *(2, df1[i, :x3]) :
                        df1[i,:x2] == "High" ? *(4, df1[i, :x3]) :
                            *(0.5, df1[i,:x3]) for i=1:N]
```

```
Out[6]: 50-element Array{Float64,1}:
         2.393325358524223
         2.4310412249328515
         5.0422956234853125
         3.7345079211580465
         0.8588031634040387
         0.5231696808320803
         2.031479078828272
         2.4806651435260076
         9.576090582790384
         6.124545507841852
         0.7720685108626731
         6.192654315744594
         6.699890875731672

         3.691184458092902
         9.562195756696816
         2.433614543200589
         2.412383239462625
         1.222717094180467
         3.6746066646508866
         2.5425484862899155
         0.6933323954399253
         7.280430172887931
         3.2074781385911297
         4.770360925053668
         2.39414636559205
```

A `DataFrame` can be sliced the same a two-dimensional `Array` is sliced, i.e., `df[row_range, column_range]`. These ranges can be specified in a number of ways:

- Using `Int` indices individually or as arrays, e.g., `1` or `[4,6,9]`.
- Using `:` to select indics in a dimension, e.g., `x:y` selects the range from `x` to `y` and `:` selects all indices in that dimension.
- Via arrays of Boolean values, where `true` selects the elements at that index.

Note that columns can be selected by their symbols, either individually or in an array `[:x1, :x2]`.

```
In [7]: ## Slicing DataFrames
        println("df1[1:2,3:4]: ",df1[1:2, 3:4])
```

```
df1[1:2,3:4]: 2Œ2 DataFrame
 Row  x3       y
      Float64  Float64

 1    1.19666  2.39333
 2    1.21552  2.43104
```

```
In [8]: println("\ndf1[1:2, [:y,:x1]]: ",df1[1:2, [:y,:x1]])
```

```
df1[1:2, [:y,:x1]]: 2Œ2 DataFrame
 Row  y        x1
      Float64  Float64

 1    2.39333  1.81025
 2    2.43104  3.1707
```

```
In [9]: ## Now, exclude columns x1 and x2
        keep = setdiff(names(df1), [:x1, :x2])
        println("\nColums to keep", keep)
```

```
Colums to keepSymbol[:x3, :y]
```

```
In [10]: println("df1[1:2,keep]: ",df1[1:2, keep])
```

```
df1[1:2,keep]: 2Œ2 DataFrame
 Row  x3       y
      Float64  Float64

 1    1.19666  2.39333
 2    1.21552  2.43104
```

In practical applications, missing data is common. In `DataFrames.jl`, the `Missing` type is used to represent missing values. In Julia, a singlton occurence of `Missing`, `missing` is used to represent missing data. Specifically, `missing` is used to represent the value of a measurement when a valid value could have been observed but was not. Note that `missing` in Julia is analogous to `NA` in *R*.

In the following code block, the array `v2` has type `Union{Float64,Missings.Missing}`. In Julia, `Union` types are an abstract type that contain objects of types included in its arguments. In this example, `v2` can contain values of `missing` or `Float64` numbers. Note that `missings()` can be used togenerate arrays that will support missing values; specifically, it will generate vectors of type `Union` if another type is specified in the first argument of the function call. Also, `ismissing(x)` is used to test whether `x` is missing, where `x` is usually an element of a data structure, e.g., `ismissing(v2[1])`.

```
In [13]: v1 = missings(2)
         println("v1: ",v1)
```

```
v1: Missing[missing, missing]
```

```
In [14]: v2 = missings(Float64,1,3)
         v2[2] = pi
         println("v2: $(v2)")
```

```
v2: Union{Missing, Float64}[missing 3.14159 missing]
```

```
In [15]: ## test for missing
         m1 = map(ismissing,v2)
         println("m1: $(m1)")
```

```
m1: Bool[true false true]
```

```
In [16]: println("Percent missing v2: ", *(mean([ismissing(i) for i in v2]), 100))
```

```
Percent missing v2: 66.66666666666666
```

Note that most functions in Julia do not accept data of type `Missings.Missing` as input. Therefore, users are often required to remove them before they can use specific functions. Using `skipmissing()` returns an iterator that excludes the missing values and, when used in conjunction with `collect()`, gives an array of non-missing values. This approach can be used with functions that take non-missing values only.

```
In [17]: ## calculates the mean of the non-missing values
         mean(skipmissing(v2))

Out[17]: 3.141592653589793
```

```
In [18]: ## collects the non-missing values in an array
         collect(skipmissing(v2))

Out[18]: 1-element Array{Float64,1}:
          3.141592653589793
```

1.2 Categorical Data

In Julia, categorical data is represented by arrays of type `CategoricalArray`, defined in the `CategoricalArrays.jl` package. Note that `CategoricalArray` arrays are analogous to factors in *R*. `CategoricalArray` arrays have a number of advantages over `String` arrays in a dataframe:

- They save memory be representing each unique value of the string array as an index.
- Each index corresponds to a level.
- After data cleaning, there are usually only a small number of levels.

`CcategoricalArray` arrays support missing values. The type `CategoricalArray{Union{T, Missing}}` is used to represent missing values. When indexing/slicing arrays of this type. `missinng` is reutrned when it is present at that location.

```
In [22]: ## Number of entries for the categorical Arrays
         Nca =10

         ## Empty array
```

```
v3 = Array{Union{String,Missing}}(undef, Nca)

##Array has string and missing values
v3 = [isodd(i) ? sample(["High", "Low"], pweights([0.35,0.65])) :
        missing for i = 1:Nca]
## v3c is of type CategoricalArray{Union{Missing,String},1,UInt32}
v3c = categorical(v3)

## Levels should be ["High","Low"]
println("1. levels(v3c): ", levels(v3c))
println("1. v3c: $(v3c)")
```

```
1. levels(v3c): ["High", "Low"]
1. v3c: Union{Missing, CategoricalString{UInt32}}["Low", missing, "High", missing, "Low", miss
```

In [20]:
```
## Reordered levels - does not change the data
levels!(v3c, ["Low","High"])
println("2.levels(v3c): $(levels(v3c))")
```

```
2.levels(v3c): ["Low", "High"]
```

In [21]:
```
println("2.v3c: $(v3c)")
```

```
2.v3c: Union{Missing, CategoricalString{UInt32}}["High", missing, "Low", missing, "Low", missi
```

Here are several useful functinos that can be used with `CatoegoricalArray` arrays:

- `levels()` returns the levels pf the `CategoricalArray`.
- `levels()` changes the order of the array's levels.
- `compress()` compresses the array saving memory.
- `decompress()` decompresses the compressed array.
- `categorical(ca)` converts the array `ca` into an array of type `CategoricalArray`.
- `droplevels!(ca)` drops levels no longer present in the array `ca`. This is useful when a dataframe has been subsetted and some levels are no longer present in the data.
- `recode(a, pairs)` recodes the levels of the array. New levels should be the same type as the original ones.
- `recode!(new, orig, pairs)` recodes the levels in `orig` using the pairs and puts the `new` levels in `new`.