`reshape2, melt`

In this course we will generally deal with data.frames in 'wide' (a.k.a. *frozen*) format: each row completely describes a situation—all of the relevant control and response variables are in columns in the row. For example a weather station's report might have columns: date/time, temperature, pressure, humidity, wind direction, wind speed, cloud cover, rain, snow, hail, ... Occasionally data is found in 'long' (a.k.a. *tall* or *melted*) format. A weather station log might then be in a three column form where the first column gives date/time, the second column names the quantity measured, and the third column gives the numerical value. E.g.:

time	**key**	**value**
00:01	"temperature"	68.2
00:02	"pressure"	29.963
00:10	"rain"	0
00:15	"wind direction"	180
00:15	"wind speed"	0
00:16	"temperature"	67.5
00:17	"pressure"	29.979
⋮	⋮	⋮

This format might recognize that measurements are not actual simultaneous, or that some quantities are not relevant (e.g., snow in July), or that some quantities may fluctuate more and hence require more frequent updates. Event monitors (for example: computer web logs) are often in this long form. I think its important for you to see this alternative form and learn a bit about how to convert between the two forms (even if you will probably not meet the long form again in this course). In going **long** $\longrightarrow$ **wide** you need some sort of matching criteria (what rows belong together as columns in a wide row) and recognize that `NA`s will be produced if every `key` in not included in the matching rows. (For example in a web log you might want to group together all the pages visited by a particular IP address during a visit. Unvisted pages would end up as `NA`.) In going **wide** $\longrightarrow$ **long**, note that a data.frame `value` must have a fixed data type (numeric, string, etc...). Thus a wind direction of "South" can not fit with a temperature of 68.2. (Of course, anything can fit in a string, but you really can't plot a temperature of "68.2".)

Using either the github or class web site, find the folder `baby_names` which contains two files `wideF.csv` and `wideM.csv`. These files contain (respectively) female and male names of babies born in the US by years (1880–2017). Each file has 1001 columns (hence: "wide") and 138 rows: one for each year. The first 1000 columns have names `V1, V2, ..., V1000` and contain (respectively) the most common baby name that year, the second most common baby name, etc. Column 1001 is named `year`.

Select either the male or female file, load it into R, and confirm that the most common name in 1880 was John (or Mary). Note the two ways used below to select this information. You can specify the row/column numbers by number: `[1,c(1,1001)]` says row number 1 and `c(1,1001)` is a vector of length 2 with values 1 & 1001 reporting that we want those two columns. (The `c` means 'combine' to make a vector; the fact that we are here talking about **c**olumns has no connection with that `c`.) Or we can select the vector that is a column using the `$` notation and ask for the first row in that vector.

```
> D=read.csv("wideM.csv",stringsAsFactors =F)
> D[1,c(1,1001)]
    V1 year
1 John 1880
```

```
> D$V1[1]
[1] "John"
> D$year[1]
[1] 1880
```

or for females:

```
> D=read.csv("wideF.csv",stringsAsFactors =F)
> D[1,c(1,1001)]
    V1 year
1 Mary 1880
> D$V1[1]
[1] "Mary"
> D$year[1]
[1] 1880
```

Remark: If you're having problems reading in the file, the problem may be the file path: i.e., what folder ("working directory"=wd) has the desired file compared to where the program thinks such files are to be found. See: `getwd()`, `setwd()`, `list.files()`. In R-studio you can set the working directory under the menu item Session.

Remark2: `stringsAsFactors=F` Recall that strings are a sequence of characters, exactly what you would think of for names. However strings are awkward things inside data.frames as they have no fixed size. Thus the default for R is to treat strings as factors. Recall factor: a series of categories internally stored as an integer (nice for a data.frame column), but externally decoded (invisibly to the user) in terms of a string. This is great for categories that repeat in the column (e.g., "male" ,"female"): just store a few integers rather than make possible space for an arbitrary variable-length string. To the user it will appear the same, and compute the same. Not uncommonly the strings are not repeated in a column (e.g., user names), and you might have a factor with a thousand categories. Actually this is still no problem for R (if the set of names does not frequently change). In such circumstances I've found actual strings often are easier to deal with.

We could now very easily answer the question: how has the 100^{th} most common name changed with time: `D[,100]`

```
> D[,100]
  [1] "Calvin"    "Anthony"   "Sidney"    "Clifford"  "Dan"       "Victor"
  [7] "Leroy"     "Emil"      "Lloyd"     "Horace"    "Sidney"    "Norman"
 [13] "Horace"    "Percy"     "Victor"    "Cecil"     "Isaac"     "Everett"

[127] "Devin"     "Richard"   "Kaden"     "Miguel"    "Bentley"   "Jaden"
[133] "Ryder"     "Juan"      "Luis"      "Elias"     "Bentley"   "Jameson"
```

or

```
> D[,100]
  [1] "Emily"      "Olive"      "Charlotte"  "Irene"      "Belle"
  [6] "Susan"      "Kate"       "Mollie"     "Virginia"   "Nannie"
 [11] "Virginia"   "Caroline"   "Caroline"   "Charlotte"  "Charlotte"

[131] "Paige"      "Kaitlyn"    "Andrea"     "Morgan"     "Hadley"
[136] "Alexandra"  "Faith"      "Madeline"
```

Just looking at your output, report (Q1) the 100^{th} most common name in the 39^{th} row which was in the year: 1879+39=1918. Q2: why add 1879 and 39, given that the first year is 1880?

I don't really think people are interested in these sorts of questions. I believe the most likely inquiry would be: plot how some particular name changed in popularity over the years. That is, we'd like to know which column has `"Tom"` (or `"Ellen"`) in the run of years 1880–2017. You can directly answer this question for the year 1880 (or any selected year):

```
> which(D[1,] == "Tom")
[1] 40

> which(D[1,] == "Ellen")
[1] 61
```

This command asks: in the first row (a vector with 1001 entries) which entry equals `"Tom"` (or `"Ellen"`). Q3: I was born in 1952 (Ellen was born in 1956) what rank was `"Tom"` (or `"Ellen"`) in the birth year?

You could programatically use `which` to fill a vector with name-rankings, but it all seems a little awkward. If we convert to long form we could immediately pick out the columns that have any particular value.

```
> library(reshape2)
> Dlong =  melt(D, variable.name = "key", value.names = "value", id.vars = "year")
> str(Dlong)
'data.frame': 138000 obs. of  3 variables:
 $ year : int  1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 ...
 $ key  : Factor w/ 1000 levels "V1","V2","V3",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ value: chr  "John" "John" "John" "John" ...
```

I hope the command arguments of `melt` make sense: the column names will end up in a column named by `variable.name` in this case `key`, the values that were in those columns will be in a column named by `value.names`, in this case `value`, and the column(s) named by `id.vars` will be reported in each new (now thin) row.

You may not have noticed but there were `NA`s in the `D` data.frame. Q4: how many `NA`s in your long (melted) data.frame? It really doesn't matter for what follows, but we might as well get rid of them.

```
> sum(is.na(Dlong$value))
> which(is.na(Dlong$value))
> Dlong=Dlong[! is.na(Dlong$value),]
```

Note that the `!` means NOT, and the `,]` means all the columns: thus all of the columns for any row for which `Dlong$value` is NOT `NA` are used to make (overwrite) the data.frame. Q5: how many rows remain in `Dlong`?

Currently `Dlong$key` is a "Factor w/ 1000 levels". We really want the numerical (integer) rank, i.e., the value that follows the `V` in the column name. Recall that a factor (finite set of possibilities in a category) uses a stand-in integer for storage that can be decoded (using `level`) for display. Typically the `level`s are meaningful names (e.g., character strings or words) whereas the integer stand-in is of limited interest. If we go back to the weather station long data.frame example, the variable names in `key` are probability stored as factors. That is the level `"temperature"` in the factor `key`

might be (internally, invisibly) stored as the integer 1; `"pressure"` as 2, `"wind direction"` as 3, etc. This abbreviation allows much more efficient use and storage of the data.frame (integers are stored rather than variable length strings). If we change the *names* of those *levels* (with the integers in the data.frame untouched) then automatically (and essentially with no CPU cycles) all the `key` values will immediate appear as changed.

```
> levels(Dlong$key)=sub("V", "", levels(Dlong$key))
```

Lets take apart this command. `levels(Dlong$key)` lists the 1000 column names that became levels of the variable `key`: `"V1", "V2", "V3",..., "V1000"`. `sub("V", "",` substitutes nothing (=`""`) for `"V"` in what follows producing 1000 strings `"1", "2", "3",..., "1000"` and that vector of strings will then replace the levels of `Dlong$key`. So `Dlong$key` remains a factor with 1000 levels, but the name of those levels has changed and now looks like a number (but importantly is still a string). So now we convert that column of a factor to a column of the corresponding numerical value of the level:

```
> Dlong$key=as.numeric(Dlong$key)
> str(Dlong)
'data.frame': 137997 obs. of  3 variables:
 $ year : int  1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 ...
 $ key  : num  1 1 1 1 1 1 1 1 1 1 ...
 $ value: chr  "John" "John" "John" "John" ...
> head(Dlong)
  year key value
1 1880   1  John
2 1881   1  John
3 1882   1  John
4 1883   1  John
5 1884   1  John
6 1885   1  John
```

See that `Dlong$key` has now become a `num`, a number.

Important Remark: Strings (a sequence of characters), integers, and real numbers (e.g., `"one"` or `"1"`, 1, 1.) are totally different things to a computer. In R we add to these basic types Factors which are used to denote possibilities in a category. The levels of that factor determines how those categories are displayed (as strings); the internal representation is essentially invisible. We can have vectors (a 1d ordered collection) of any single basic type, but a vector must include just one type. *Lists* in R (a new data type) are a generalization of vectors that allows a mix of basic types. The locations within a list are denoted by doubled square brackets (`list[[1]]` would be the first element of `list`) or optionally by name using the `$` notation for a named element.

Finally we can easily plot the rank of most any name in one line:

```
> plot(Dlong$year[Dlong$value=="Tom"],Dlong$key[Dlong$value=="Tom"])
```

The vector of x values is obtained by subsetting `Dlong$year` to just the ones that relate to `"Tom"` `Dlong$value=="Tom"` (this is a vector of True and False). We get the corresponding y values (the ranks) that are in `Dlong$key` and plot those (x, y) values as points (a 'scatter plot').

If you use Rstudio, you can immediately Export this plot as a pdf. If you use the raw command line (as I do) several additional commands are required:

```
dev.new()
pdf(file="tom.pdf", width=10,height=7)
plot(Dlong$year[Dlong$value=="Tom"],Dlong$key[Dlong$value=="Tom"])
dev.off()
```

Much nicer graphics are available using the `ggplot2` library (which I'll not be covering in this class). Nevertheless we can do a lot better just using default plotting.

```
> plot(Dlong$year[Dlong$value=="Tom"],Dlong$key[Dlong$value=="Tom"],
xlab="Year",ylab="rank", main="title here",pch=20 )
```

`pch` controls the point plot symbol. If lines between points are required: `type="l"`; the line type is then controlled by `lty`. (Q6) Make and hardcopy of a nice plot showing name-rank as a function of time for some name, perhaps your own name; the data points should be filled triangles (use google to find the required `pch` value).

Let's say you want to display together Thomas along with Tom; we can add those points to the existing plot:

```
> points(Dlong$year[Dlong$value=="Thomas"],Dlong$key[Dlong$value=="Thomas"],pch=17)
```

By default `plot` wants to plot the entire data.frame. So another way to make this composite plot is to make a subset data.frame including just Tom & Thomas and then have `plot` do 'all' of that:

```
> Dsub=Dlong[Dlong$value=="Thomas" | Dlong$value=="Tom",]
> Dsub$value=as.factor(Dsub$value)
> plot(Dsub$year,Dsub$key, pch=as.integer(Dsub$value))
```

We needed to automatically change the `pch` for the different names so we needed to get a number out of `"Tom"` and `"Thomas"`. The solution was to convert the `Dsub$value` from a vector of strings to a vector of factors (with two levels: Tom & Thomas). Factors can be converted to numbers (because that's what a factor actually is) whereas in general a string cannot be directly converted to a number (but for example `nchar` is a function that reports the number of characters in a string, so that's one way to do a string →integer conversion). A brief tour throught default plot options is in chapter 10 of RCook, again mostly not covered here. `ggplot2` can do most anything, so there are several book-length treatments of its many options. If I want to make a complex plot I typically use a google search —and include the keyword ggplot— to tell me how to achieve the plot.

Remark: Within the `reshape2` library the reverse of `melt` is `dcast` which does: **long** $\longrightarrow$ **wide**. In the popular `tidr` library, `gather` goes **wide** $\longrightarrow$ **long** and `spread` goes **long** $\longrightarrow$ **wide**. Base R has a `reshape` command which also has these functionalities.

scrape: `RCurl, XML, rvest`, impute: `mice, Hmisc`, ...

These days, the web is a primary source for data. With luck you can directly download a `.csv` file perfect for R to digest. However often data lies behind web pages designed to make access easy for browsing humans. Automating access to such "easily accessible" data can be a challenge.

The Social Security Administration (SSA) provides a list of popular baby names for years since 1880: `https://www.ssa.gov/OACT/babynames/` with a web form that allows you to display (on your browser) tables of common baby names. The form includes a few options (e.g., year, number of names to be provided) the aim perhaps is to answer what SSA considers common non-robotic interests. But what if the question you want answered is more complex than allowed by their options? For example: How has the popularity of your name changed over time? Are baby names becoming more diverse? Are longer baby names becoming more common? If you had the entire dataset, R could happily crunch through that data and answer arbitrary questions.

The topic of robotically mining the web for data ('scrapeing', 'wrangling') is beyond the aims of this course, but I wanted to introduce you to such tools, so if you wanted to pursue such problems (e.g., for your final project), you'd have a place to start.

The library `RCurl` essentially allows you to run a browser inside of R and capture the resulting pages. The library `XML` allows you to convert tables on such pages into R `data.frames`. `rvest` is a higher level package to do all of the above. Often the resulting data, which was designed for human eyes, will have problems when viewed with robotic eyes. (E.g., What are the column names, and can they be converted to nice `colnames` variables? What is a `NA`? How will the computer interpret `1,000,000` (three comma separated values?) or 1×10^6 (superscripts? $\times$=x?) or `1E6`?) I leave all such questions aside, and simply comment that I used these tools to automatically download 138 pages for the years 1880–2017, and used R to massage the resulting 138 tables into a single `.csv` file with each row reporting 1000 columns with the ranked 1000 most common male names and a column for the year. While the Social Security site always provides male and female data in a single table, I've made separate `.csv` files for males: `wideM.csv` and females: `wideF.csv`.

FYI: for srapeing data out of `.pdf`s I've found *tabula* helpful.

The issue of `NA` ("Not Available" or "Not Applicable" or "No Answer") data deserves a bit more comment. If you are training or predicting data with `NA` values there is no foolproof fix. The simplest fix is just to drop rows in a data.frame that include one or more `NA`s (`complete.cases, na.omit, drop_na`). However if you have lots of columns even a small `NA` probability can result in a large fraction of faulty rows. For a factor variable `NA` may be a telling value, but you can't calculate with a numeric `NA` (nor a `NaN` = not a number, for example 1/0). "Imputation" is any process that attempts to fill in `NA` blanks with values. "Multiple imputation" recognizes that any replacement value has uncertainty. Therefore the entire dataset is duplicated with various possible replacement values and the entire analysis is completed on each of these slightly different datasets. The results of these analyses will of course differ slightly allowing you to measure how much the imputation affects the results.