R for the ICPSR data aRchive.

Dave Armstrong
University of Western Ontario
Department of Political Science

e: dave.armstrong@uwo.ca
w: www.quantoid.net/teachicpsr/dataarchive

Contents

1 Introduction

This handout will provide some of the basic information you need to manage and distribute R data files. You can download the most recent version of R (as of this writing, 3.4.1) at the Comprehensive R Archive Network (CRAN). R is open source software and has currently over 11,000 user-developed packages. Once you download R, you can download and install the packages you need. This takes two steps.

1. `install.packages('<package name>')` has to be done only one time untill you upgrade to a new minor or major version of R. This downloads the package's code and installs it in R. This will also install any dependencies that are required.

2. `library('<package name>')` has to be done in each R session[1] when you want to use the functions in the package.

    - One way to get around this is to build an `.Rprofile` file that has in it a list of instructions for R to execute as it starts up. In this file, you could put instructions to load packages you commonly use. An example of on such file is:

      ```
      .First <- function(){
        library(readstata13)
        library(foreign)
        library(tidyverse)
      }
      ```

      The `.Rprofile` file goes in your home directory. Note, that there is no prefix, the file starts with a period.

We will need the following packages today: `tidyverse`, `plyr`, `readstata13`, `foreign`, `sas7bdat`.

1.1 What happens when you type library('<package name>')

R only knows to look in locations that are in its *search path*. When you type `library('<package name>')` R puts `<package name>` in its search path. That means it can look inside that package for a particular function.

```
search()

##  [1] ".GlobalEnv"          "package:coda"        "package:runjags"
##  [4] "package:foreign"     "package:zoo"         "package:readr"
##  [7] "package:readstata13" "package:knitr"       "package:haven"
## [10] "tools:RGUI"          "package:stats"       "package:graphics"
## [13] "package:grDevices"   "package:utils"       "package:datasets"
## [16] "package:methods"     "Autoloads"           "package:base"

library(readstata13)
search()

##  [1] ".GlobalEnv"          "package:coda"        "package:runjags"
##  [4] "package:foreign"     "package:zoo"         "package:readr"
##  [7] "package:readstata13" "package:knitr"       "package:haven"
## [10] "tools:RGUI"          "package:stats"       "package:graphics"
## [13] "package:grDevices"   "package:utils"       "package:datasets"
## [16] "package:methods"     "Autoloads"           "package:base"
```

[1]A session exists for the time that a particular R instance is open on your computer.

1.2 Object Orientation

R is an object oriented language. One implication of this is that output from all functions can be saved as objects, including functions that read in data. For example, the following will save the contents of `anes2004.dta` into the object `dat`.

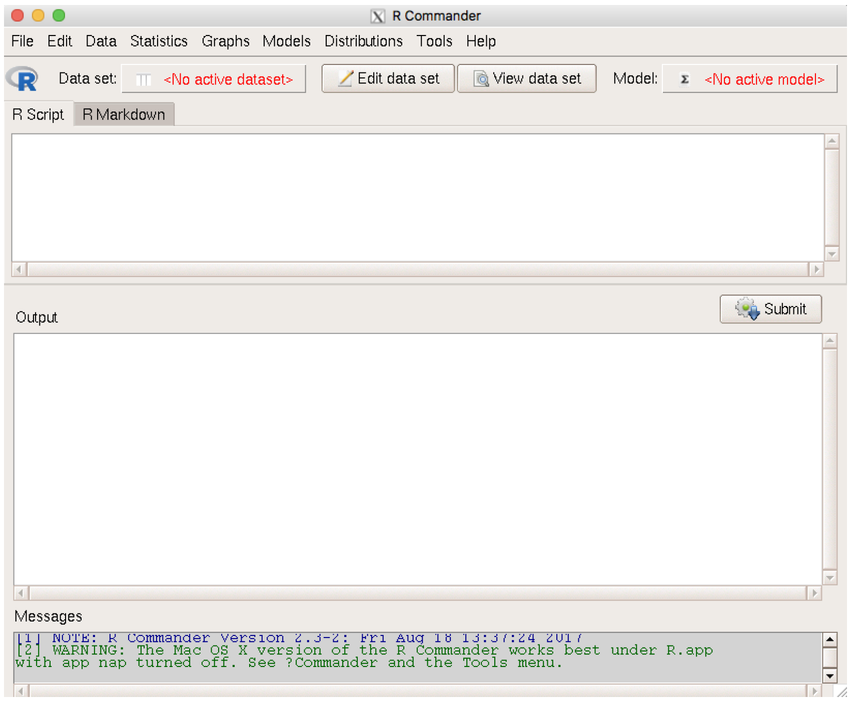
```r
library(readstata13)
dat <- read.dta13("anes2004.dta")
```

2 IDEs and GUIs

There are a number of ways of interacting with R other than through the command line. These fall under two different headings - graphical user interfaces (GUIs) and integrated development environments (IDEs).

There are two prominent GUIs for R - the R Commander and JGR (Java GUI for R). These have different strengths and weaknesses, but R Commander is likely more prominent. One reason for this is that R Commander is built on the tcl/tk framework and as such, doesn't require that the Java runtime environment be installed. This can be somewhat troublesome depending on permissions, etc... There is a plugin to JGR called Deducer that gives it more flexibility. There are also several add-ons for the R Commander that give it greater functionality. The R Commander website has a listing of these add-ons. There are download linkes, etc... for JGR at its website.

```r
install.packages("Rcmdr")
library(Rcmdr)
```

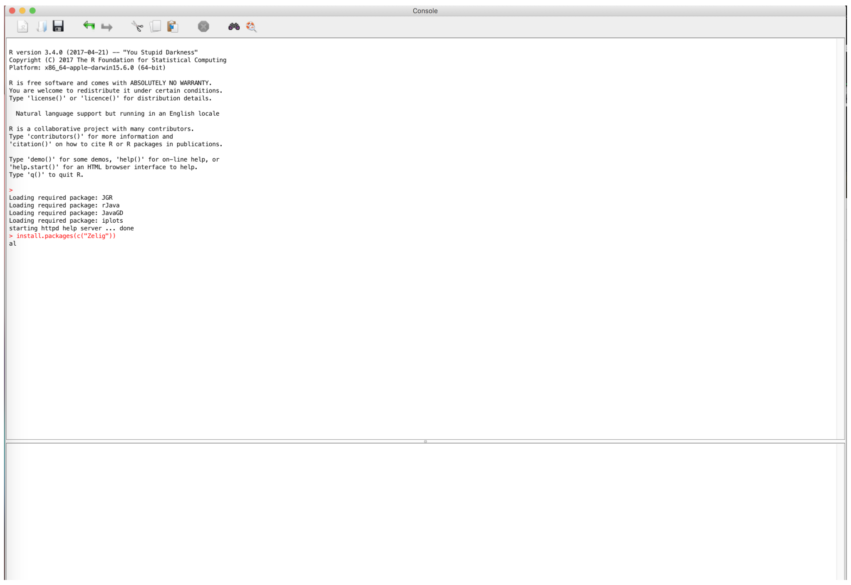


```
[1] NOTE: R Commander Version 2.3-2: Fri Aug 18 13:37:24 2017
[2] WARNING: The Mac OS X version of the R Commander works best under R.app
with app nap turned off. See ?Commander and the Tools menu.
```

The R Commander is better for an Intro Statistics class than it would be as a high-powered data management tool. One of the problems is that some of the newer data management features (like using `haven` or `readstata13`) are not included as drop-down options. Ultimately, this doesn't get you away from knowing and using the code directly.

The same thing could be said for JGR.

```r
install.packages("JGR")
library(JGR)
JGR()
```

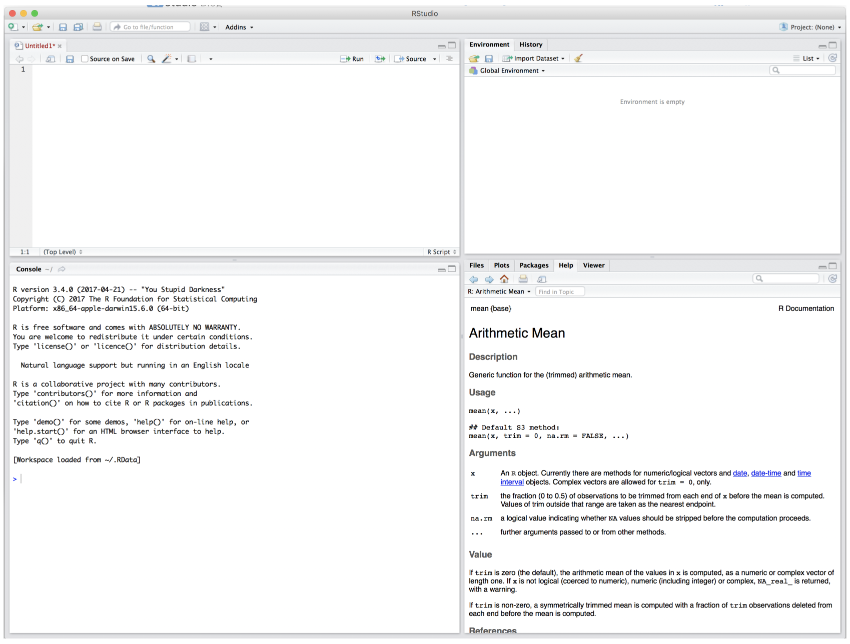


```
R version 3.4.0 (2017-04-21) -- "You Stupid Darkness"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
Loading required package: JGR
Loading required package: rJava
Loading required package: JavaGD
Loading required package: iplots
starting httpd help server ... done
> install.packages(c("Zelig"))
al
```

Most people looking for an interface to R choose RStudio these days. RStudio has less in the way of drop-down menus, but does have some other nice features.

- Relatively full-featured code editor with regular expression search and replace and multiple cursors.
- Code completion with the `tab` key.
- Function arguments with the `ctrl+space` key combination.
- Nice interactive debugging routine.
- RMarkdown, LaTeX+`knitr` support

You can download RStudio from its website.



```
R version 3.4.0 (2017-04-21) -- "You Stupid Darkness"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> 
```

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

3 Importing and Exporting

R has lots of functionality to read in data from other sources, but there are user-developed packages that enhanice this functionality. Loading the `tidyverse` package will make available a number of functions through the `readr` package.

- `read_csv()` is a function that reads `.csv` files with smart parsers for columns. This improves on R's native functionality.
- `read_fwf()` imports fixed-width format data and improves on R's native functionality because it allows for explicit column parsing in different forms.

3.1 Short Digression on Tibbles

These functions read data in to a *tibble*. First, we can back up a bit. When R reads in data, it generally stores the data in a `data.frame`. This is a rectangular array of data that can have different types of data - numeric, logical, factors, strings. A `matrix` in R can only hold one type of data. When making and modifying data frames with the `data.frame` function, there is the possibility for some undesirable behavior.

- One of the biggest problems is recycling vectors. All vectors will be recycled regardless of their length. A warning will only get printed if the number of rows of the data frame is not divisible by the length of the small vector.

```
df1 <- data.frame(x = c(1,2,3,4,5,6,7,8), y = c(9, 10, 11, 12),
  z = c(13,14))
df1
```

```
##   x  y  z
## 1 1  9 13
## 2 2 10 14
## 3 3 11 13
## 4 4 12 14
## 5 5  9 13
## 6 6 10 14
## 7 7 11 13
## 8 8 12 14
```

```
df2 <- data.frame(x = c(1,2,3,4,5,6,7,8), y = c(9, 10, 11, 12),
  z = c(13,14,15))
```

```
## Error in data.frame(x = c(1, 2, 3, 4, 5, 6, 7, 8), y = c(9, 10, 11,
12), :  arguments imply differing number of rows:  8, 4, 3
```

Tibbles have a number of nice features,

- It never changes an input's type (i.e., no more `stringsAsFactors = FALSE`!).

```
library(tidyverse)
tibble(x = letters)
```

```
## # A tibble: 26 x 1
##         x
##     <chr>
##   1     a
##   2     b
##   3     c
##   4     d
##   5     e
##   6     f
##   7     g
##   8     h
##   9     i
## 10      j
## # ... with 16 more rows
```

  This makes it easier to use with list-columns:

```
tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
```

```
## # A tibble: 3 x 2
##       x          y
##    <int>      <list>
## 1      1  <int [5]>
## 2      2 <int [10]>
## 3      3 <int [20]>
```

List-columns are most commonly created by `do()`, but they can be useful to create by hand.

- It never adjusts the names of variables:

```
names(data.frame(`crazy name` = 1))

## [1] "crazy.name"

names(tibble(`crazy name` = 1))

## [1] "crazy name"
```

- It evaluates its arguments lazily and sequentially:

```
tibble(x = 1:5, y = x ^ 2)

## # A tibble: 5 x 2
##         x     y
##     <int> <dbl>
## 1       1     1
## 2       2     4
## 3       3     9
## 4       4    16
## 5       5    25
```

- It never uses `row.names()`. The whole point of tidy data is to store variables in a consistent way. So it never stores a variable as special attribute.
- It only recycles vectors of length 1. This is because recycling vectors of greater lengths is a frequent source of bugs.

If you're distributing R data, doing it as a tibble would be in accordance with best practices.

3.2 Importing from Statistical Software

R can easily import data in formats used by most major statistical packages. The `haven` (part of the `tidyverse`) package has a number of these functions. Typing the following will show you all of the functions.

```r
help(package='haven')
```

There are a couple of important differences between R and other software that can result in confusion.

- R does not really allow partially labeled categorical variables (what R calles `factors`). There is functionality in `haven` to enable such, but it only does this as an intermediary step. In R, a factor has to be entirely labeled or not.
- When R makes a factor, it replaces whatever values underly the factor with sequential integer values starting with 1.

Example with `haven`.

```r
library(haven)
dat <- read_dta("st_example.dta")
dat

## # A tibble: 4 x 3
##         ccode     x           y
##     <dbl+lbl> <dbl> <dbl+lbl>
## 1           2     1       NaN
## 2          20     2         5
## 3          40     3         6
## 4         200     4       -99

dat$y

## <Labelled double>
## [1] NaN   5   6 -99
##
## Labels:
##  value label
##    -99    DK

## turn y into a factor
as_factor(dat$y)

## [1] NaN 5   6   DK
## attr(,"label")
## [1] another variable
## Levels: DK 5 6 NaN

## look at the numeric values
as.numeric(as_factor(dat$y))

## [1] 4 2 3 1

## look at numeric values after turning ccode into
## a factor
as.numeric(as_factor(dat$ccode))

## [1] 1 2 3 4
```

One of the other big differences between R and other stats packages is that the levels of a factor (rather than the underlying numbers) are the primary means for identifying values. For example:

```
dat$ccode_fac <- as_factor(dat$ccode)
dat$ccode_fac == "USA"

## [1]  TRUE FALSE FALSE FALSE

dat$ccode_fac == 1

## [1] FALSE FALSE FALSE FALSE
```

the functionality in `haven` keeps the information related to these partially labeled factors. One reason is that often the labels are attached to missing values. For example, age might exist with unlabeled values between 18 and 97 with 999 labeled as a non-response. The function below, `zap_missing2` will change those values to missing. This is the situation with the `y` variable in the example above.

```
zap_missing2 <- function (x, miss.levels=NULL){
    x[is.na(x)] <- NA
    labels <- attr(x, "labels")
    labels <- labels[!is.na(labels)]
    if(!is.null(miss.levels)){
      x[which(x %in% miss.levels)] <- NA
      attr(x, "labels") <- labels[-which(labels %in% miss.levels)]
    }
    else{
      x[which(x %in% labels)] <- NA
      x <- as.numeric(x)
  }
    x
}
dat$y2 <- zap_missing2(dat$y)
dat$y2

## [1] NA  5  6 NA
```

Now the `y2` variable in `dat` is a numeric variable, such that all labeled values were recoded as R's missing value `NA`.

Using `haven`, whether you read the data in from SPSS or Stata or Sas, the results and the way you would interact with the data are pretty much the same.

3.2.1 Short Digression about Missing Values

In R, missing values work differently than in Stata. In Stata, missing values have a numerical value of ∞. Thus, inequalities for example $x > 1$ will evaluate to true if $x = .$.

In R, missing values (`NA`) always evaluate to missing in any logical expression. Further, many low-level statistical routines evaluate to missing if there is missing data in the vector of values.

```
mean(dat$y2)

## [1] NA

mean(dat$y2, na.rm=T)

## [1] 5.5
```

Also, to evaluate logical expressions about missingness, you can use the `is.na()` function.

```
dat$y2 == NA

## [1] NA NA NA NA

is.na(dat$y2)

## [1]  TRUE FALSE FALSE  TRUE
```

There are similar functions `is.null()` and `is.infinite()` to evaluate other conditions as well.

3.3 Back to haven

All of this suggests that one way of managing data in R is to actually manage the data in something else and then read it in to R. Another question is, what to do if you get an R dataset, and you want to send it to other types of software? The answer here depends on the underlying structure of the data. The important distinction is the storage format of the data: tibble or data frame and (related) how the variable descriptions are stored - as an attribute of the varaible (as in a tibble) or as an attribute of the data frame.

```
df1 <- data.frame(
  x = c(1,2,3,4,5),
  y=factor(c(1,2,1,2,1), labels = c("no", "yes")))
attr(df1, 'var.labels') <- c("x variable", "y variable")
attributes(df1)

## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3 4 5
##
```

```
## $class
## [1] "data.frame"
##
## $var.labels
## [1] "x variable" "y variable"
```

If the variable labels are in the attributes to the data frame, then `save.dta13` from the `readstata13` package will do the trick.

```
save.dta13(df1, "df1_example.dta")
```

This can be the case whether or not the dataset is a tibble. However, there is another option, which is that the variable labels are an attribute to the variable, rather than the data frame.

```
df2 <- data.frame(
  x = c(1,2,3,4,5),
  y=factor(c(1L,2L,1L,2L,1L), labels = c("no", "yes")))
attr(df2$x, "label") <- "x variable"
attr(df2$y, "label") <- "y variable"
attributes(df2)

## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3 4 5
##
## $class
## [1] "data.frame"

attributes(df2$x)

## $label
## [1] "x variable"
```

In this case, you could use `write_dta()` from the `haven` package:

```
write_dta(df2, "df2_tib_example.dta")
```

4 Data Attributes

Another reason that you might want to work directly in R is that you can set attributes of objects. Reading data in from Stata (or any other package) produces an object and the attributes of that object depend on the routine used to create the data. However, those attributes can be modified or extended as you like.

For example, the data source and citation could be added as attributes to the data. You can add any attributes, with the following code:

```
attr(data, "<attribute name>") <- <attribute value>
```

5 Multiple Datasets

Finally, one of the advantages of R is that you can distribute a single workspace with multiple files in it. When users load the workspace, all of the datasets corresponding to the study could be loaded simultaneously.

- This could be useful if the data are not particularly big.
- You wouldn't want to burden users by loading a really large dataset they might not want to use, but could be good if all datasets are small(ish).

6 Example

Thinking about how the distribution system works now and how it could be changed/improved, consider the example data, here, the 21st Century Americanism: Nationally Representative Survey of the United States Population, 2004 (ICPSR 27601). Currently, this is distributed as a single R workspace (`.rda` file) that contains the single object `da27601.0001` which holds the data. The archive acknowledges that R reads in factors in an unusual way (that is it does not natively respect the underlying numeric values). To counter this, files distributed with the dataset include the file `factor_to_numeric_icpsr.r`, which contains the R code:

```
library(prettyR)
lbls <- sort(levels(da99999.0001$MYVAR))
lbls <- (sub("^\\([0-9]+\\) +(.+$)", "\\1", lbls))
da99999.0001$MYVAR <- as.numeric(sub("^\\(0*([0-9]+)\\).+$", "\\1", da99999.0001$MYVA
da99999.0001$MYVAR <- add.value.labels(da99999.0001$MYVAR, lbls)
```

One of the problems with such code is that it requires the user to change the name of the dataset and the name of the variable each time they want to do this. It would be much easier to make this a function that people could use directly, for example:

```
factor_to_numeric <- function(x){
  require(prettyR)
  lbls <- sort(levels(x))
  lbls <- (sub("^\\([0-9]+\\) +(.+$)", "\\1", lbls))
  out.x <- as.numeric(sub("^\\(0*([0-9]+)\\).+$", "\\1", x))
  out.x <- add.value.labels(out.x, lbls)
  out.x
}
```

Then, users would just have to do something like:

```
da27601.0001$Q89n <- factor_to_numeric(da27601$Q89)
```

This could be included as an extra `.r` file, or it could be included as a function in the workspace.

6.1 Using haven to Produce Data

Another option, would be to present the data as a tibble created by reading the Stata dataset in with the `haven` package. This would permit the use of `labelled` variables that preserve the numeric information, but can be transformed into factor variables with the `as_factor()` function from the `haven` package.

6.2 Distributing Data in a Package

Finally, a more time-consuming, but perhaps better, option would be to distribute the data as an R package. This would allow you to include functions (with help files) and the data (with its own help file). I provide an example of the package approach for the ICPSR 27601 dataset. This could include all of the original documentation along with the help files. Much of the transfer from the current system to a package could be automated.