SOC 7717 Event History Analysis and Sequence Analysis

Weeks 10–11: Introduction to R

Wen Fan

Spring 2019

Outline

Software Installation and Checklist

Software installation

- Download R
- Download RStudio

Checklist

- Do you have the most recent version of R? (the latest should be 3.5.3 as of March 28, 2019)
  - `version$version.string`
- Do you have the most recent version of RStudio? (the latest should be 1.1.463 as of March 28, 2019)
  - `RStudio.Version()$version`
- Have you updated all of your R packages?
  - `update.packages(ask = FALSE, checkBuilt = TRUE)`

R Principles

Some R basics

An object-orientated programming (OOP) approach

- Everything is an object
- Everything has a name
- You do things using packages (i.e., "libraries"), although you can (and perhaps should) learn to write your own functions

R vs. Stata

- Multiple objects (e.g., data frames) can co-exist in the same workspace
  - No more `keep`, `preserve` / `restore`
  - A direct consequence of the OOP approach
- You will load packages at the start of every new R session
  - "Base" R comes with many useful functions
  - However, many of R's best data science functions and tools come from external packages written by other users

```
install.packages('tidyverse', dependencies = TRUE)
```

R SCRIPTS

- .R: R code
- .Rmd: R markdown (R code + human language)

R Basics

BASIC ARITHMETIC

R is a powerful calculator and recognizes all of the standard arithmetic operators

```
1 + 2  ##Addition

6 - 7  ##Subtraction

4/2    ##Division

2^3    ##Exponentiation
```


BASIC ARITHMETIC

We can also invoke modulo operators (integer division and remainder) $\longrightarrow$ Very useful when dealing with time

```
100 %/% 60  ##How many whole hours in 100 minutes?

100 %% 60 ##How many minutes are left over?
```

Logical operators

R also comes equipped with a full set of logical operators (and Boolean functions), which follow standard programming protocol. For example:

```
1 > 2

1 > 2|0.5 ##"|" stands for "or"

1 > 2&0.5 ##"&" stands for "and"

isTRUE(1 < 2)
```


Logical operators

Negation: we use `!` as a short hand for negation. This will come in very handy when we start filtering data objects based on non-missing (i.e., non-NA) observations

```
is.na(1:10)

!is.na(1:10)
```

Value matching: To see whether an object is contained within a list of items, use `%in%`

```
4 %in% 1:10

4 %in% 5:10
```


Logical operators

In R, as in Stata, we always use two equal signs for logical evaluation

```
1 = 1 ##This doesn't work

1 == 1 ##This does

1 != 2
```


Assignment

In R, we can use either `=` or `<-` to handle assignment. `<-` is normally read as "gets". Of course, an arrow can point in the other direction too (i.e., `->`). But this is seldom used

```
a <- 10 + 5
a
```

You can also use `=` for assignment

```
b = 10 + 10 ##Note that the assigned object must be on the left with "="
b
```

Most R users prefer `<-` for assignment, since `=` also has specific role for evaluation within functions

Read data

- `read.xlsx()`: read an Excel file
- `read.csv()`: read a csv file
- `read.dta()`: read an Stata file
  - » In order to read an Stata format after version 12, need to use `read.dta13` from the `readstata13` package

When reading data into R, use `getwd()` and `setwd()` to work with working directories

Some useful descriptive commands

- `dim()`
- `head`
- `colnames()`
- `unique()`
- `table()`

Help and comments

If you are struggling with a function or object in R, simply type `help` or `?`

For many packages, you can try the `vignette()` function, which will provide an introduction to a package and its purpose through a series of helpful examples. But you need to know the exact name of the package vignette(s)

- You can run `vignette()` (i.e. without any arguments) to list the available vignettes of every installed package installed on your system
- Or, run `vignette(all = FALSE)` if you only want to see the vignettes of any loaded packages

`example()` can also be useful

Comments in R are demarcated by `#`

Object and Name

What are objects?

There are many different types (or classes) of objects. Some most useful ones include:

- vectors
- matrices
- data frames
- lists
- functions

What are objects?

Each object class has its own set of rules governing how that object can be used in R

- For example, you can perform many of the same operations on matrices and data frames. But there are some operations that only work on a matrix, and vice versa
- Often you can convert an object from one type to another

```
## Create a small data frame called "df"
df <- data.frame(x = 1:8, y = 9:16)
df

## Convert it to a matrix called "m"
m <- as.matrix(df)
m
```


Object class, type, and structure

Use the `class`, `typeof`, and `str` commands if you want to understand more about a particular object

```
class(df) ##Evaluate its class
typeof(df) ##Evaluate its type
str(df) ##Show its structure

View(df)
```


Global environment

Now, let's try to run a regression on these `x` and `y` variables:

```
lm(y ~ x) ##The "lm" stands for linear model(s)
```

R can't find the variables in our Global Environment

- Because the variables `x` and `y` live as separate objects in the global environment, we have to tell R that they belong to the object `df`

Global environment

There are a various ways to solve this problem. One is to simply specify the data source:

```
lm(y ~ x, data = df)
```

This global environment issue represents a profound difference between Stata and R

- In Stata, the entire workspace consists of one (and only one) data frame $\longrightarrow$ no ambiguity where variables are coming from
- However, this convenience comes at a really high price. You can never read more than two separate datasets into memory at the same time, have to find all sorts of ways (e.g., `egen`) to add summary variables to your data, etc.

Global environment

Some other ways to solve the problem are

- `attach` the data set right before using it, and `detach` immediately after using it
- Use `with`

Working with multiple objects

R's ability to keep multiple objects in memory at the same time is a huge plus when it comes to effective data work

However, it also means that you have to pay attention to the names of those distinct data frames and be specific about which objects you are referring to

Reserved words

We can assign objects to different names. However, there are a number of special words that are "reserved" in R (e.g., if, else, while, function, for, TRUE, FALSE, NULL, Inf, NaN, NA)

There are named functions or constants that you can re-assign if you really wanted to, but already come with important meanings from base R

- Try to type `pi` and see what happens

Namespace conflicts

A similar issue crops up when we load two packages, which have functions that share the same name. E.g., look what happens when we load the `dplyr` package

```
library(dplyr)
```

The messages that you see about some object being masked from "package:stats" are warning you about a namespace conflict

- Both the `dplyr` and the `stats` package (which gets loaded automatically when you start R) have functions named `filter` and `lag`

Namespace conflicts

Whenever a namespace conflict arises, the most recently loaded package will gain preference. So the `filter()` function now refers specifically to the `dplyr` variant

But what if we want the `stats` variant?

- Temporarily use `stats::filter()`
- Permanently assign `filter <- stats::filter`

Solving namespace conflicts

1. We can explicitly call a conflicted function from a particular package using the `package::function()` syntax

```
stats::filter(1:10, rep(1, 2))
```

We can use `::` for more than just conflicted cases

- E.g. Being explicit about where a function (or data set) comes from can help add clarity to our code. Try these lines of code in your R console

```
dplyr::starwars ##Print the starwars data frame from the dplyr package

scales::comma(c(1000, 1000000)) ##Use the comma function, which comes
    from the scales package
```


Solving namespace conflicts

2. A more permanent solution is to assign a conflicted function name to a particular package. This will hold for the remainder of your current R session, or until you change it back

```
filter <- stats::filter
filter <- dplyr::filter ##Change it back again
```

Pay attention to any warnings when loading a new package

Indexing

Indexing

We can use `[]` to index objects that we create in R

```
a <- 1:10
a[4] ##Get the 4th element of object "a"

a[c(4, 6)] ##Get the 4th and 6th elements
```

It also works on larger arrays (vectors, matrices, data frames, and lists). For example:

```
starwars[1, 1]
```


Indexing for lists

Lists are a more complex type of object in R. They can contain an assortment of objects that don't share the same class, or have the same structure

- E.g., a list can contain a scalar, a string, and a data frame. Or you can have a list of data frames, or even lists of lists

Lists require two square brackets `[[]]` to index the parent list item and then the standard `[]` within that parent item

```
my_list <- list(a = "hello", b = c(1, 2, 3), c = data.frame(x = 1:5, y =
    6:10))
my_list[[1]] ##Return the 1st list object
my_list[[2]][3] ##Return the 3rd element of the 2nd list object
```


Indexing

Another indexing operator is `$`

```
my_list
```

Notice how our (named) parent list objects are demarcated: `$a`, `$b`, and `$c`. We can call these objects directly by name using the dollar sign

```
my_list$a ##Return list object "a"
my_list$b[3] ##Return the 3rd element of list object "b"
my_list$c$x ##Return column "x" of list object "c"
```

Indexing

In some cases, you can combine the two index options

- E.g., Get the 1st element of the `name` column from the `starwars` data frame

```
starwars$name[1]
```


Removing objects

Use `rm()` to remove an object or objects from your working environment

```
a <- "hello"
b <- "world"
rm(a, b)
```

You can also use `rm(list = ls())` to remove all objects in your working environment (except packages)

Removing plots

You can use `dev.off()` to remove all plots that have been generated during your session. For example, try this in your R console:

```
plot(1:10)
plot(11:60)
dev.off()
```

You may also have noticed that RStudio has convenient buttons for clearing your workspace environment and removing plots

Tidyverse Packages

Tidyverse

Let's install and load the tidyverse meta-package and check the output

```
library(tidyverse)
tidyverse_packages()
```

We're going to focus on two packages: `dplyr` and `tidyr`, two workhorse packages for cleaning and wrangling data

Pipes

In R, the pipe operator is denoted `%>%` and is automatically loaded with `tidyverse`

Pipes can dramatically improve reading and writing code. Compare:

```
mpg %>% filter(manufacturer == "audi") %>% group_by(model) %>% summarise(
    hwy_mean = mean(hwy))
summarise(group_by(filter(mpg, manufacturer == "audi"), model), hwy_mean
    = mean(hwy))
```

- The first line reads from left to right, exactly how we thought of the operations: take this object `mpg`, do this (`filter`), then do this (`group by`), etc.
- The second line totally inverts this logical order

Key dplyr verbs

There are five key `dplyr` verbs that you need to learn

- `filter()`: Filter (i.e., subset) rows based on their values
- `arrange()`: Arrange (i.e., reorder) rows based on their values
- `select()`: Select (i.e., subset) columns by their names
- `mutate()`: Create new columns
- `summarise()`: Collapse multiple rows into a single summary value

dplyr::filter()

We can chain multiple `filter` commands with the pipe (`%>%`), or just separate them within a single `filter` command using commas

```
starwars %>% filter(species == "Human", height >= 190)
```

A very common `filter()` use case is identifying missing cases

```
starwars %>% filter(is.na(height))
```

To remove missing observations, simply use negation: `filter(!is.na(height))`

DPLYR::ARRANGE()

```
starwars %>% arrange(birth_year)
```

Arranging on a character-based column (i.e., strings) will sort alphabetically

We can also arrange items in descending order using `arrange(desc())`

```
starwars %>% arrange(desc(birth_year))
```


DPLYR::SELECT()

Use commas to select multiple columns (variables) out of a data frame. (You can also use `first:last` for consecutive columns)

Deselect a column with `-`

```
starwars %>% select(name:skin_color, species, -height)
```

You can also rename your selected variables

```
starwars %>% select(ID = name, homeworld, sex = gender)
```


DPLYR::SELECT()

The `select(contains())` option provides a nice shortcut in relevant cases

```
starwars %>% select(name, contains("color"))
```

The `select(..., everything())` option is another useful shortcut if you want to bring some variable(s) to the front of a data frame

```
starwars %>% select(species, homeworld, everything())
```


DPLYR::MUTATE()

You can create new columns from scratch, or as transformations of existing columns

```
starwars %>%
    select(name, birth_year) %>%
    mutate(birth_decade = birth_year / 10) %>%
    mutate(comment = paste0(name, " was born in decade ", birth_decade, "
        ."))
```

DPLYR::MUTATE()

You can chain multiple mutates in a single call

```
starwars %>%
    select(name, birth_year) %>%
    mutate(
        birth_decade = birth_year / 10, ##Separate with a comma
        comment = paste0(name, " was born in decade ", birth_decade, ".")
    )
```


DPLYR::MUTATE()

Boolean, logical, and conditional operators all work well with `mutate()`

```
starwars %>%
    select(name, height) %>%
    filter(name %in% c("Luke Skywalker", "Anakin Skywalker")) %>%
    mutate(tall1 = height > 180) %>%
    mutate(tall2 = ifelse(height > 180, "Tall", "Short")) ##Same effect
        but can add labels
```


DPLYR::MUTATE()

Lastly, there are variants of `mutate()` that work on a subset of variables

- `mutate_all()` affects every variable
- `mutate_at()` affects named or selected variables
- `mutate_if()` affects variables that meet some criteria

```
starwars %>% select(name:eye_color) %>% mutate_if(is.character, toupper)
    %>% head(5) ##Convert characters into uppercase and show the first
    five rows
```


DPLYR::SUMMARISE()

Particularly useful in combination with the `group_by()` command

```
starwars %>%
    group_by(species, gender) %>%
    summarise(mean_height = mean(height, na.rm = T))
```

Note that including `na.rm = T` is usually a good idea with `summarise`. Otherwise, any missing value will propagate to the summarized value

dplyr::summarise()

The variants that we saw earlier also work with `summarise()`

- `summarise_all()` affects every variable
- `summarise_at()` affects named or selected variables
- `summarise_if()` affects variables that meet some criteria

```
starwars %>%
    group_by(species, gender) %>%
    summarise_if(is.numeric, list(avg = mean), na.rm = T) %>% head(5)
```


Other dplyr commands

`group_by()` and `ungroup()`: for (un)grouping

- Particularly useful with the `summarise()` and `mutate()` commands

`slice()`: subset rows by position rather than filtering by values

```
starwars %>% slice(c(1, 5))
```


Other dplyr commands

`count()` and `distinct()`: number and isolate unique observations

```
starwars %>% count(species)
starwars %>% distinct(species)
```

You could also use a combination of `mutate()`, `group_by()`, and `n()`

```
starwars %>% group_by(species) %>% mutate(num = n())
```

There are also a whole class of window functions for getting leads and lags, ranking, creating cumulative aggregates, etc.

```
vignette("window-functions")
```


tidyr

Key tidyr verbs

- `gather()`: gather (or "melt") wide data into long format
- `spread()`: spread (or "cast") long data into wide format
- `separate()`: separate (i.e., split) one column into multiple columns
- `unite()`: unite (i.e., combine) multiple columns into one

tidyr::gather()

```
stocks <- data.frame( ##Could use "tibble" instead of "data.frame"
    time = as.Date('2019-01-01') + 0:1,
    X = rnorm(2, 0, 1),
    Y = rnorm(2, 0, 2),
    Z = rnorm(2, 0, 4)
)
stocks

tidy_stocks <- stocks %>% gather(key = stock, value = price, -time)
tidy_stocks
```


tidyr::spread()

```
tidy_stocks %>% spread(stock, price)

tidy_stocks %>% spread(time, price)
```


tidyr::separate()

```
starwars %>% separate(name, c("first_name", "last_name"))
```

To avoid ambiguity, you can also specify the separation character with, for example, `separate(..., sep = ".")`

A related function is `separate_rows()`, for splitting up cells that contain multiple fields or observations

```
jobs <- data.frame(
    name = c("Jack", "Jill"),
    occupation = c("Homemaker", "Philosopher, Philanthropist")
)
## Now split out Jill's various occupations into different rows
jobs %>% separate_rows(occupation)
```

tidyr::unite()

```r
gdp <- data.frame(
    yr = rep(2016, times = 4),
    mnth = rep(1, times = 4),
    dy = 1:4,
    gdp = rnorm(4, mean = 100, sd = 2)
)
gdp

## Combine "yr", "mnth", and "dy" into one "date" column
gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-")
```


Other tidyr commands

Use `crossing()` to get the full combination of a group of variables

```r
crossing(side=c("left", "right"), height=c("up", "down"))
```

See `?expand()` and `?complete()` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames

Resources

- Cheat Sheets
- R for Data Science
- R Markdown
- Data Visualization
- R for Stata Users