Introduction to R

Dr. Willi Mutschler

Introduction

Aims and prerequisites

- Objective: Learn how to use R for econometrics and statistics
- Prerequisites:
    1. Basics in probability theory and statistical inference
    2. Multiple linear regression

References

All materials and their source-codes are freely available online:

`https://github.com/wmutschl/IntroR/tree/IW`

Course is based on

- Introductory courses held at the Econ Department at University of Münster
- Introductory courses on datacamp.com
- Andreas Behr and Ulrich Pötter (2011): *Einführung in die Statistik mit R*
- Muenchen, Hilbe (2012): *R for Stata Users*

Topics

1. What is R?
2. R-Studio Basics
3. Managing workspace and packages
4. Get help and understand the documentation
5. Programming language basics
6. Controlling functions
7. Data structures and acquisition
8. Selection and transformations of variables and observations
9. Treatment of missing values
10. Data visualization (basic and using gramar)
11. Regression analysis
12. Programming

What is R?

What is R?

“The most powerful statistical computing language on the planet…”

What is R?

It all depends on the use and user, BUT: R is

- an intuitive interface to the most advanced statistical methods available today
- built specifically for data analysis and visualization
- one of the most popular languages for data science
- preferred by statisticians and academic researchers
- language of choice for cutting edge statistics
- a vast collection of community-contributed packages

What is R?

- The language S (an object-oriented statistical computing language) is implemented as S-Plus (commercial) and R (OpenSource)
- R is a
    1. language
    2. package
    3. environment

  for graphics and data analysis
- R is FOSS (think about collaborations!) and easily extendable
- Similar programming languages: Matlab, GAUSS, Julia

Comparison to STATA

In general, five independent parts of a software

- Data input and management (language)
- Statistics and graphics procedures (commands)
- Output management systems
- Macro language
- Matrix language

$\hookrightarrow$In other softwares, e.g. Stata, these are standalone and developed separately

$\hookrightarrow$In R all five were planed to be unified from the beginning

Comparison to STATA

- `Base R` plus recommended packages contains over 1600 functions similar to e.g. STATA
- Tested via extensive validation programs
  - ↪ R is accurate even though no company behind it, R responds very quickly to bugs etc.
  - ↪ Source code of R (*scripts*) are similar to *Do Files* in STATA
- Note that new statistical methods are nowadays often first published in R, and only later included by PROGRAMMERS (not original scientists) into STATA

R Console

The basic command window is called the `R Console`

Prompt: `>`

You can input commands and execute them (by pressing the RETURN key)

```
1+1
1+1 # This is a comment: 1+1
(1+2)*3
(5/3)^4.5
5+2; 7+3; 2*5
pi
Pi
PI
2*((1+2)*(1-2)
```

This will: (1) evaluate it, (2) print the result, (3) count the lines with [n] and (4) delete the result

Editors

- Long computations should not be done interactively in the command window
- Use an editor to write a program and then execute it in R
- There is a built-in editor in R: `Datei -- Neues Skript`
- External editors:
    - R-Studio [RECOMMENDED!]
    - Tinn-R, Notepad++, Atom, Emacs, etc. are also possible

R-Studio

- Overview of four panels in R-Studio:
  1. Script editor
  2. Environment|History|Connections,
  3. Console|Terminal
  4. Files|Plots|Packages|Help|Viewer
- Important shortcuts (see also the Magic Wand)
  - [CTRL+ENTER] for Run
  - [CTRL+SHIFT+C] for commenting a section
  - [CTRL+L] clears command windows
  - [TAB] Function completion
  - [ARROWS UP AND DOWN] in command window: scroll through history
  - [F1] gets you help

Concatenation, Assignments, Strings

Open a new Script in R-Studio and try out the following:

```
c(1,4,7)
a <- c(1,4,7)
print(a)
a
A
b <- c(1,a,3)
(b <- c(1,a,3))
mean(b)
simpsons_kids <- c( #family name
"Bart", #boy name
'Lisa', #girl name
"Maggie" #baby name
)
print(simpsons_kids)
c(2,"3") -> y
```

Execute a single line (or multiple lines by marking them) by pressing CTRL-ENTER

Concatenation, Assignments, Strings

- c() is the concatenation operator and probably the most used command in R
- <- and -> assign values to variables, you could also use = but it is not recommended
- if you did not complete commands, you get a +. Most of the times close a), or hit ESC key or CTRL+C
- comments go from # to the end of line, can be between functions or in the middle with line breaks
- there is no block comment features, simply use [CTRL-SHIFT-C] in R-Studio to (un)comment sections
- Use either double quotation marks ("some string") or single ones ('some other strings'). Note that double quotes are preferred (and output is printed using double quotes)

Parenthesis

(Parenthesis)

- control math order as usual in algebra, e.g.

```
1+1*10
(1+1)*10
```

- print assignment values:

```
(x<-c(1,2,3,4,NA))
```

- provide options to functions, e.g.

```
mean(x, na.rm=FALSE)
mean(x, na.rm=TRUE)
```

Parenthesis

{Curly Braces}

- Can combine many commands into one
- Executes all assignments but returns only value of last expression
- Useful for writing own functions

```
{x<-2; y<-1
z<-x+y; z2<- z^2
z
z2}
```

[Square Braces] and [[Double Square Braces]]

- Used for selecting/indexing elements within objects (we will get to that later)
- Double squares are used in lists (one of the most general data structure)

Managing workspace

Managing workspace

Listing Objects

- `ls()` or `objects()` lists the objects in your workspace
- `rm(list=ls())` clears workspace

Working Directory

- Easiest: Use GUI, i.e. Session - Set Working Directory
- alternatively: `getwd()` and
  `setwd("c:/Users/wmutschl/Documents/RKurs")` (PC)
  `setwd("/Users/wmutschl/Documents/RKurs")` (Mac)
  `setwd("/home/wmutschl/Documents/RKurs")` (Linux)
- Note that the path name is structured by slashes (/), not backslashes (\)

Managing workspace

Quitting

- Quit R by the command `q()`
- Quit RStudio by using the GUI or [CTRL+Q]
- In general, save your script files, but do *not* save your workspace
- Note that in RStudio even unsaved script files are kept

Packages

Packages

- One of the strengths of R is the large and growing collection of packages that can be downloaded from CRAN (or e.g. Github)
- Installation (only once!)
  - Use R-Studio interface for Packages
  - install.packages("packagename")
- Installed packages are activated by `library("packagename")`
- Help about packages: `library(help="packagename")`

Common problems

- install.packages(”ggplot”)
  - ↪ either not available or wrong package name
- install.packages(ggplot2)
  - ↪ forgot quotes
- library(”prettyR”)
  - ↪ forgot to install it
- library(”dplyr”)
  - ↪ masked or covered up, that’s okay
- detach(”package:dplyr”) is opposite of library, which might prevent conflicts in function names and save on memory

Which packages to use?

- The Comprehensive R Archive Network (CRAN): can be searched by name or via Task Views for key programs on `cran.r-project.org`
- `crantastic.org`: rated software
- `rdocumentation.org`: top packages
- `r-bloggers.com`
- Git[hu|la]b, …

Exercise 1

Help and documentation

Help and documentation

- To obtain details about a command, type `?command` or `help(command)`

```
?mean
help(mean)
help("for")
help("while")
?"while"
help(package = "prettyR")
methods(plot) #gives you overview of extra functions, e.g.
help(plot.lm)
help.start()
```

- R-Studio: select/click on a command and hit F1

Programming Language Basics

R objects

- R is object oriented
- An object can be anything: scalar, vector, matrix, string, table, factor, list, data frame, regression results, model, ...
- object name should begin with letter, no numbers, no underscores, no special characters, case matters
- The object type determines how some commands work (e.g. `plot`, `summary`)
- Every object has a unique name

Variables

All kinds of values can be stored in a variable (as we are object-oriented):

- numbers
- letters
- words
- dates
- logical TRUE/FALSE values
- data structures
- plots
- other objects
- ....

Mode, Class, Dimension and Length of Vectors

```
x <- c(1, 2, 1, 2, 1, 2, 1, 2)
print(x); mode(x); class(x); length(x); dim(x)
x+x
2*x
x + 10
x + c(10, 100)
x <- as.matrix(x)
print(x); mode(x); class(x); length(x); dim(x)
```

- `mode`: a variable's type
- `class`: vectors have a class of character or numeric (or many other things), dimension (`dim`) of a vector is NULL
- `length`: number of elements it contains (including (!) missings)
- Note: If one vector is shorter, its values are **recycled** until the lengths match

Operators and functions

Logical operators

- `&` and
- `|` or
- `!` not
- `NA` not available or no answer
- `==` equal (do *not* use =)
- `>`, `>=` greater than, greater than or equal
- `<`, `<=` less than, less than or equal
- `!=` not equal

Operators and functions

Examples logical operators

```
5 < 7
1+1 == 3
a <- c(-1,4,9)
a >= 2 & a < 8
b <- c(NA,1,2,3)
b > 0
is.na(b)
a[a>2]
a == 4
a = 4
```

Exercise 2

Operators and functions

Arithmetic operators and mathematical functions

`+`, `-`	plus, minus
`*`, `/`	multiplication and division
`^`	power (exponentiation)
`Inf`, `-Inf`	infinity (plus or minus)
`NaN`	not a number
`abs`	absolute value
`sqrt`	square root
`exp,log`	exponential function and natural logarithm (*not* `ln`)
`sin`	sinus (other trigonometric functions as well)
`sum`	sum

Examples arithmetic operators and mathematical functions

```
x <- c(-1,0,2,9,3)
abs(x)
sqrt(x)
1/x
-1/x
0/x
log(x)
x^c(2,3,2,3,2)
x^c(2,3)
log(x)<0
```

Exercise 3

Operators and functions

Matrix functions

`matrix`	creates a matrix from a vector
`dim`	dimensions of a matrix
`t`	transpose
`%*%`	matrix multiplication
`det`	determinant
`solve`	inverse
`eigen`	eigenvalues and eigenvectors
`diag`	diagonal
`cbind`	merge matrices column-wise
`rbind`	merge matrices row-wise

Operators and functions

Examples matrix functions

```
X <- matrix(c(2,3,4,5,1,1,9,3,2),3,3)
X
dim(X)
det(X)
solve(t(X)%*%X)
X*c(8,5,1)
X%*%c(8,5,1)
diag(X)
diag(X) <- 0
X
solve(X)%*%X
matrix(NA,4,4)
rbind(cbind(X,X),c(0,1))
```

Note the difference between `*` and `%*%`!

Exercise 4

Operators and functions

Set operations and special functions

`unique`	the set of all unique elements of a vector
`union`	$x \cup y$
`intersect`	$x \cap y$
`setdiff`	$x \backslash y$
`%in%`	$x \in y$
`sort`	sort the elements of a vector
`cumsum`	cumulated sum of a vector (also `cumprod`, `cummin`, `cummax`)
`which(...)`	find the index of the vector element for which some condition is true
`which.min`	find the index of the smallest vector element (also `which.max`)

Exercise 5

Operators and functions

Sequences and replications

`seq` sequence from *a* to *b* of length *n*,
`seq(from=a,to=b,length=n)`,
or by increments of size *d*,
`seq(from=a,to=b,by=d)`

`a:b` integer sequence from *a* to *b*

`rep` replicate a vector *n* times
`rep(what,times=n)`,
or each element *n* times,
`rep(what,each=n)`

Exercise 6

Indexing

Indexing vectors

- R has a rich indexing syntax
- The basic ideas are the same for vectors, matrices and other objects
- Indexing is used to read or manipulate specified elements of the objects
- Indexes are always given in square brackets: `[]` (or sometimes `[[]]` for lists)
- Indexes can be either numerical or logical
- We will start with vectors and then look at matrices and dataframes
- The symbols `i` and `j` denote integer variables (not vectors)

Indexing Vectors

Numerical indexing

`x[1]`	first element
`x[2]`	second element
`x[i]`	*i*-th element
`x[-i]`	all elements, without position *i*
`x[a:b]`	all elements from position *a* to position *b*
`x[k]`	`k` numerical vector: all elements at positions given in *k*

Logical indexing

`x[a]`	`a` logical vector: all elements where *a* is true (`a` must have the same length as `x`)

Indexing Vectors

Indexing vectors

```
x <- c(2,3,4,5,1,1,9,3,2)
x[2]
x[4:7]
x[20]
x[-9]
x[-3]
x[c(1,5,1,9,9)]
a <- (x<4)
x[a]
x[x<4]
```

Exercise 7

Indexing Matrices

Numerical indexing

`x[i,j]`	element in row *i*, column *j*
`x[,j]`	column *j* (as a vector)
`x[i,]`	row *i* (as a vector)
`x[,-j]`	without column *j*
`x[-i,]`	without row *i*
`x[a:b,j]`	elements *a* to *b* in column *j*
`x[k,m]`	`k`,`m` numerical vectors: all elements at positions given in *k* and *m*

Indexing Matrices

Logical indexing Let `a` denote a logical matrix of the same dimension as `x`;
let `k` and `m` denote logical vectors of suitable length

`x[a]`	All elements of `x` at positions where `a` is true, as a *vector!*
`x[,m]`	All columns of `x` where `m` is true
`x[k,]`	All rows of `x` where `k` is true

Of course, one may use numerical indexing for one dimension and logical indexing for the other dimension

`x[k,1:2]`	All elements of columns 1 and 2 where `k` is true
`x[3,m]`	All elements of row 3 where `m` is true

Indexing Matrices

Examples

```
x <- matrix(1:16,4,4)
x[3,3]
x[,4]
x[2,]
x[,-1]
x[-3,]
x[2:4,4]
x[c(1,4,2,2,2),1:2]
```

Indexing Matrices

Examples

```
x <- matrix((-7:8)^2,4,4)
a <- (x<10)
x[a]
x[,c(TRUE,FALSE,TRUE,FALSE)]
x[x[,1]<30,3:4]
x[x[,2]==1 | x[,3]==1,]
x[2:4,4]
x[c(1,4,2,2,2),1:2]
```

Exercise 8

Controlling Functions

Calling Functions

- R is controlled by functions: when you use an R function you call it and pass values to their arguments
- arguments are listed in (parenthesis) and are separated by comas
- argument values are usually single objects and have a unique name
- function calls **return** a **value** (in help file, output is often called return/value)
- value is a single object, may contain much information, optimized for further analysis not (necessarily) for displaying

More on function arguments

Common Error:

```
x1 <-c(1,2,3); x2 <-c(5,6,7); x3 <-c(8,9,0);
mean(x1,x2,x3) #nope!
summary(data.frame(x1,x2,x3)) #better!
```

- When calling a function, the order of the arguments is arbitrary, if the argument names are explicitly used:

  `mean(na.rm=FALSE,trim=.1,x=x1)`

- Without argument names, R assigns the values in the order of the function definition:

  `mean(x1,.1,FALSE)`

- A function definition may include default values for arguments:

  `mean(x, trim = 0, na.rm = FALSE)`

- If an argument with a default value is missing in a function call, R uses the default value

Data Structures

Data Structures

Most Statistics programs have only one data structure, R is more flexible

- Factors
- Arrays
- Vectors
- Matrices
- Data frames
- Tables
- Lists
- or make your own

Factors

Does this make sense?

```
degree <- c(0, 2, 1, 2, 3, 2, 1, 3)
gender <- c("f", "f", "f", NA, "m", "m", "m", "m")
degree[gender=="f"]
degree[gender=="m"]
table(degree)
table(gender)
summary(gender)
summary(degree)
summary(degree[gender=="m"])
```

- No! We need to tell R that these are categorial variables! This is important as this will
    - print the right statistics and summaries
    - automatically include dummy variables in regression model
- Note that NA is always included

Factors

Better:

```
degree <- factor(degree)
degree
summary(degree)
degree <- factor(degree,
        levels=c(1,2,3,4),
        labels = c("BA","MA","PhD","Other"))
degree
summary(degree)
gender <- factor(gender,
        levels = c("m","f"),
        labels = c("Male","Female"))
summary(gender)
degree[gender=="Female"]
degree[gender=="f"] #note this does not work anymore!
```

Note that values you do not include in levels become NA

Data Frames

Why use data frames?

- data frames are rectangular set of variables
- variables are called components (vectors, factors, columns)
- observations are called rows or cases
- mode is list, class is data.frame, components must have equal length (same number of observations)
- variable names and row names are stored as attributes
- Almost never required, but...
    - lock values of observations together
    - ensures proper sorting
    - ensures correct NA removal

Data Frames

```
testscores <- c("1.7", "1.3", "1.0", "1.7", "2.0")
mydata <- data.frame(degree, gender, testscores)
testscores <- c("1.7", "1.3", "1.0", "1.7", "2.0", NA, NA, NA)
mydata <- data.frame(degree, gender, testscores)
mydata
names(mydata)
rownames(mydata)
rownames(mydata) <- c("Bart","Homer","Maggie","Marge","Nelson","Apu","
    Moe","Krusty")
rownames(mydata)
mydata
class(mydata$testscores)
mydata$testscores <- as.numeric(as.character(mydata$testscores))
class(mydata$testscores)
```

- data.frame converts character variables to factors unless you add stringsAsFactor = FALSE

Large Data Frames

For large data frames use `tbl_df` from `dplyr` package

- offers better printing of and defaults for (large) data frames
- reports number of [rows by columns]
- prints only 10 observations (option can be changed)
- prints only enough variables to fill your screen
- class becomes tbl_df, tbl, data.frame
- affects other functions like print()

```
data(Titanic)
detach("package:dplyr")
print(data.frame(Titanic))
plot(data.frame(Titanic))
library(dplyr)
print(tbl_df(Titanic))
plot(Titanic)
detach("package:dplyr")
```

Data Structures Overview

Matrix

- same as data frame, but mode must be the same, i.e. atomic objects (all numeric or all character)
- class is matrix
- actually one long vector stored with a dimension attribute (dim)

Array

- Matrix that may have more than two dimensions
- Vectors are 1D Arrays, Matrices are 2D Arrays
- actually one long vector stored with a dimension attribute (dim)

Data Structures Overview

Lists

- object that can store any other type of objects, called components
- `mylist <- list(name1 = comp1, name2 = comp2, ...)`

```
mylist <- list(degree, gender, testscores)
mylist
mylist <- list(UniversityDegree=degree,
               sex=gender,
               "Test Score"=testscores)
mylist
names(mylist)
identical(mylist[[1]], mylist$UniversityDegree)
identical(mylist[[2]], mylist$sex)
identical(mylist[[3]], mylist$`Test Score`)
```

- modeling functions often output their values as lists
- for indexing we need double square brackets or names with $ sign

Data Structures Overview

Some useful commands

- print
- head
- tail
- names
- rownames
- mode
- class
- attributes
- str

Sorting and merging

- The sort command sorts (numeric or character) vectors
- By default, the elements are sorted ascendingly, but one can also sort descendingly.
- Matrices are sorted as vectors
- Dataframes cannot be sorted by sort
- The function order(x) returns a vector of the position of the smallest, the second smallest, ..., the largest elements of x
- Hence, x[order(x)] returns the sorted vector
- The order command is useful for sorting matrices and dataframes!

Sorting and merging

- Two dataframes can be merged by common column names
- The command `merge(x,y,by=...)` merges two dataframes `x` and `x` by a common variable given in the `by`-option
- What happens if there are observations in `x` that are missing in `y` (or vice versa)?
- There are options to choose the way R deals with missings

Data import and export

Data import and export

General remarks

- R is all about working with data
- There are various ways to read data from different sources in many formats
- In R, datasets are usually represented as `data.frame` objects
- R has a large collection of “standard datasets”, see `data()`

Data import and export - Manual data input

- Very small datasets can be typed in directly, e.g.
  `x <- data.frame(v1=c(2,6,1,1),v2=c(9,9,8,8))`
- To edit existing objects, use `data.entry`, e.g.
  `y <- data.entry(x)`
- However, editing data within R is *not* recommended
- Datasets should be stored outside R, preferably in separate directories
- The datasets should be easily accessible by data-managing programs (e.g. Excel, Stata, ASCII editors, ...)

Data import and export - Saving and loading R objects

- All R objects can be saved by the command `save(obj1,obj2,...,file="c:/path/name.Rdata")`
- In principle, other file name extensions are possible, but not recommended
- All objects saved in a file can be loaded by the command `load("c:/path/name.Rdata")`
- The data format is R specific

Data import and export - Reading and writing text files

- A convenient command to read simple text files is `read.csv("c:/path/filename.txt")`
- The command assumes the following data format:
    1. The first row contains the variables names, delimited by commas
    2. The following rows are the observations, the variables are again delimited by commas
    3. The decimal sign is a dot (not a comma)
- Use `read.csv2` if the variables are delimited by semi-colons and the decimal sign is a comma (i.e. German style)
- More options are available for the command `read.table`
- Exporting text files from R is usually not necessary. If it is, use `write.csv`, `write.csv2` or `write.table`

Exercise 9

Data import and export - Other data formats

- there are many packages that provide easy access to datasets in other data formats
- flat files
  - readr: fast, easy to use, consistent
    - read_delim instead of read.table
    - read_csv instead of read.csv
    - read_tsv instead of read.delim
  - data.table for huge data sets
    - fread just works and is ridiculously fast (infers column types and separators)
- Excel
  - readxl is fast
    - read_excel(”data.xlsx”, sheet = ”my_sheet”)
  - XLConnect to have much more control and bridge Excel into R
  - several other packages, e.g. gdata uses Perl, xlsx uses Java...
- Databases
  - dbConnect from DBI package

Data import and export - Other Statistical Software Packages

haven

- consistent, easy and fast (uses C library)
- SAS (read_sas), STATA (read_stata or read_dta), and SPSS (read_por or read_sav)

foreign

- less consistent, very comprehensive (saves everything into attributes)
- for formats `dbf`, `Stata`, `SPSS`, `SAS`, and a few more (but not Excel)
- read.dta takes also care of STATAs different missing values

Exercises 10, 11, 12 and 13

Selection and Transformations of Variables

Selecting Variables

Most programming packages:

- Select variables by name
- Select observations by logical condition

R can do that as well, but has many more ways to select and transform variables (we can even reverse this order)

Indexing Data Frames

- Dataframes have the same index methods as matrices
- Logical conditions can include strings (character variables)
- There are three additional ways to extract data frame columns:
  1. `x$varname`
  2. `x[[i]]`
     where `i` can also be a numerical vector
  3. `x["varname"]`
     or `x[c("varname1","varname2",...)]`
- Dataframe variables can be addressed directly by their name when you `attach` the dataframe, e.g. `attach(x)`
- `all`, `any` and `which` are also useful here

Indexing Data Frames

Common Error: Forgetting the comma!

```
mean(mydata["testscores"],na.rm = T) #this will give you NA
mean(mydata[,"testscores"],na.rm = T) #Don't forget the comma!
```

Exercise 14

select from dplyr package

- the select function makes life much easier as it selects all kinds of variables and always returns a data frame

```
library(dplyr)
load(”mydata100.RData”)
select(mydata100, workshop, gender) # for as many as I like
select(mydata100, gender:q4) # take all variables that are in between
    and itselft too
select(mydata100, contains(”q”))
select(mydata100, starts_with(”q”))
select(mydata100, ends_with(”shop”))
select(mydata100, num_range(”q”, 1:4))
```

- you can also use regular expressions
- be careful, most stat functions work on vectors, not on data frames

Selecting observations

Put logic in the row position (before the comma)

```
str(mydata100)
summary(mydata100[mydata100$gender == "Female", ]) #don't forget the
    comma!
```

you could actually index on different objects

filter function from dplyr package

Use the filter() function

```
library(”dplyr”)
summary(filter(mydata100, gender == ”Female”))
```

Selecting both variables and observations

Traditional way:

```
myVars <- c("gender", "q1", "q2", "q3", "q4")
myObs <- which(mydata100$gender == "Female")
mysubset <- mydata100[myObs, myVars]
summary(mysubset)
```

Modern way:

```
library("dplyr")
mysubset <- select(mydata100, gender, q1:q4)
mysubset <- filter(mysubset, gender == "Female")
summary(mysubset)
```

- first call select or filter, whichever gives you the smallest subset

Transformations

Very tedious:

```
mydata2 <- mydata100
mydata2[, "diff"] <- mydata100[, "q4"] -mydata100[, "q1"]
mydata2[, "ratio"] <- mydata100[, "q4"] / mydata100[, "q1"]
mydata2[, "q4log"] <- log(mydata100[, "q4"])
mydata2[, "meanQ"] <- (mydata100[, "q1"] + mydata100[, "q2"] + mydata100[,
      "q3"] + mydata100[, "q4"])/4
mydata2
```

Transformations

Much cooler: mutate function from dplyr

```
mydata3 <- mutate(mydata100,
   diff = q4 -q1,
   ratio = q4 / q1,
   q4log = log(q4),
   meanQ = (q1+q2+q3+q4)/4
)
mydata3
```

- lets you use short names without attaching dataframes
- returns original data frame plus the new variables for every row
- works only on columns or variables
- Important: don't forget commas!

Pipes

Advanced feature ”pipes” %>%: feeds results from one to another

```
load(”mydata100.RData”)
mydata100 %>%
  filter(gender==”Female”) %>%
  select(q1:q4) %>%
  mutate(diff = q4 -q1,
          ratio = q4 / q1,
          q4log = log(q4),
          meanQ = (q1+q2+q3+q4)/4
          )
```

Exercise 15

Graphics

Some Remarks on Graphics

Base Graphics (Traditional Way)

- plot() offers many methods
- extremely flexible and extensible, but not easy to use with groups
- Uses “traditional graphics system”

Some Remarks on Graphics

```
load("mydata100.RData")
mydata100 <- na.omit(mydata100)
attach(mydata100)
head(mydata100)
plot(workshop)
plot(workshop,gender)
plot(gender,workshop)
plot(workshop,posttest)
plot(posttest,workshop)
plot(posttest)
plot(pretest,posttest)
hist(posttest)
rug(posttest)
```

Many options

Nicer plots

```
plot(pretest, posttest,
     pch = 19, # plot character
     cex = 2, # character expansion
     main = ”My Main Title”,
     xlab = ”My X Axis Label”,
     ylab = ”My Y Axis Label”
)
grid() #add grid to plot
par() #overview of graphics parameters
```

Subplots

```
par(mfrow = c(2, 1)) # 2 rows, 1 column
plot(workshop[gender == "Female"], main = "The Females")
plot(workshop[gender == "Male"], main = "The Males")
#scatter plot with regression line
par(mfrow = c(1, 1))
plot(pretest,posttest)
abline(c(18,0.8))
myModel <- lm(posttest ~ pretest, data=mydata100)
abline(coefficients(myModel))
```

Problems

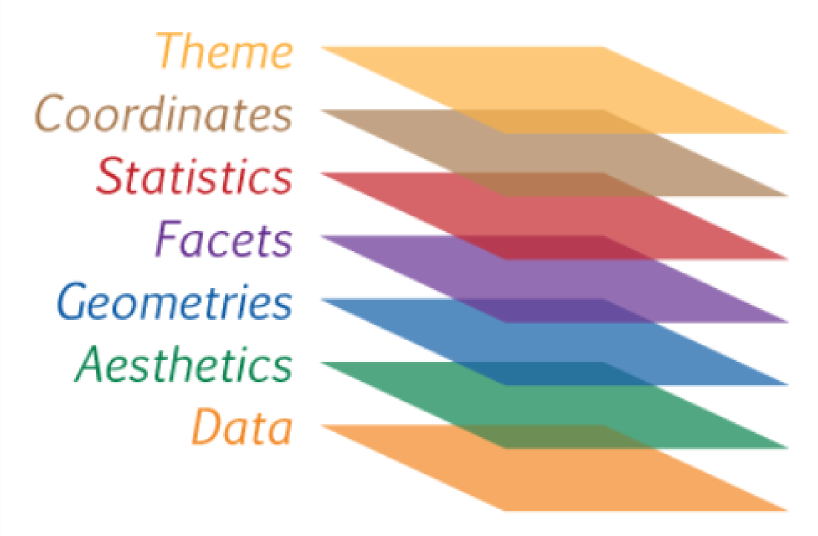
- axis are not standardized
- a lot of white unnecessary space
- titles of each plot are in the vertical position taking valuable space
- one could fix all this with several options...

ggplot2 - Basic idea

ggplot from `ggplot2` package

- follows Wilkinson's Grammar of Graphics
- works with underlying graphics concepts, not pre-defined graph types
- enables to create any data graphic that you can think of
- uses Grid Graphics System instead of traditional system

ggplot2 - Grammar of Graphics



Data, Aesthetics and Geometries are essential.

Data

- The dataset being plotted

```
ggplot(data=mydata100)
```

- This does not show anything

ggplot2 - Grammar of Graphics

Aesthetics

- How are variables **mapped** to visual properties of geometric objects?
- For example: x-axis, y-axis, color, fill, size, labels, alpha, shape, line width, line type

```
ggplot(data=mydata100, aes(x=workshop))
```

- This still does not show anything as it doesn't know what geometry to apply

ggplot2 - Grammar of Graphics

Geoms: Geometric objects

- The visual elements used for data, e.g. point, line, histogram, bar, boxplot

```
ggplot(data=mydata100, aes(x=workshop, fill=gender)) +
  geom_bar()
ggplot(mydata100, aes(x=pretest, y=posttest)) +
  geom_point()
ggplot(mydata3, aes(x=pretest, y=posttest,col=meanQ)) +
  geom_point()
ggplot(mydata100, aes(x=pretest, y=posttest, col=workshop, size=q4)) +
  geom_point()
ggplot(mydata3, aes(x=pretest, y=posttest, shape=workshop, col=meanQ, size=q4)) +
  geom_point()
myboxplot <- ggplot(mydata100, aes(x=workshop, y=posttest)) +
  geom_boxplot()
print(myboxplot)
myboxplot +
  geom_point()
ggplot(mydata100, aes(pretest,posttest, label = workshop)) +
  geom_text()
```

- One great advantage of ggplot2: the plot is an object which can be manipulated!
- Type `geom_` and hit tab to see all possible values of OBJECT (which can be extended by other packages)

Facets

- Facets allow to put multiple charts / plots on one canvas by dividing into columns and rows
- Think of facets in terms of grouping your data

```
p <- ggplot(mydata100, aes(x=posttest,fill=gender)) +
  geom_histogram()
p
p + facet_grid(. ~ gender)
p + facet_grid(gender ~ .)
p + facet_grid(workshop ~ gender)
```

Statistics

- Transformation or summary of your data before mapping to an aesthetic, e.g. binning, smoothing, descriptive, inferential
- Often the geom has default stats, which may work alright
- Sometimes providing a different stats does improve the clarity, e.g. calling
  - stat_bin instead of geom_bar or geom_histogram
  - stat_smooth instead of geom_smooth
  - stat_boxplot instead of geom_boxplot
- help of stat functions is often more informative

```
p <- ggplot(mydata100, aes(x = pretest, y = posttest, col = gender)) +
geom_point(alpha=0.4)
p + geom_smooth()
p + geom_smooth(method="lm")
p + geom_smooth(method="lm",se=FALSE)
```

ggplot2 - Grammar of Graphics

Coordinates

- Coordinate system controls how positions are mapped to the plot
  - Cartesian coordinates
  - Polar coordinates
  - Spherical projection
- Customization (and deception), e.g.:
  - apply limits to x-axis or y-axis
  - tune aspect-ratio
  - zoom in

```
p <- ggplot(mydata100, aes(x = pretest, y = posttest, col = gender)) +
geom_point(alpha=0.4) +
geom_smooth()
p
p + scale_x_continuous(limits = c(65, 75)) #note the error message
p + coord_cartesian(xlim = c(65, 75)) # coord_cartesian(): the proper
    way to zoom in
```

Theme

- ggplot2 provides themes like theme_bw(), theme_classic, theme_dark, theme_light

```
p
p + theme_classic()
p + theme_dark()
p + theme_light()
p + theme_minimal()
p + theme_bw()
p + theme_grey()
p + theme_gray()
```

- useful for corporate design, publication standards, personal choice

More examples

```
library(RColorBrewer)
myColors <- c("black", brewer.pal(4, "Dark2"))
ggplot(mydata100, aes(x = pretest, y = posttest, col = workshop)) +
  geom_point(alpha=0.8,shape="#") +
  geom_smooth(method = "lm", se = FALSE, fullrange=T) +
  geom_smooth(method = "lm",
          aes(group = 1, col = "All"), # Add col inside aes()
          se = FALSE, fullrange=T) +
  scale_color_manual("Participants", values = myColors)

load("recess.Rdata")
data("economics")
ggplot(economics, aes(x = date, y = unemploy/pop)) +
  geom_rect(data = recess,
        aes(xmin = begin, xmax = end, ymin = -Inf, ymax = +Inf),
        inherit.aes = FALSE, fill = "red", alpha = 0.2) +
  geom_line()
```

ggplot2 - attributes

More examples

```
p <- ggplot(mydata100, aes(x=workshop, fill = gender))
p + geom_bar(position="fill")
p + geom_bar(position="stack")
p + geom_bar(position="dodge")
p + geom_bar(position = position_dodge(0.2), alpha = 0.6)
p + geom_bar(position="fill") +
scale_fill_grey()

library(RColorBrewer)
display.brewer.all(n=5) # how many column patterns
p + geom_bar(position = "stack") +
  scale_fill_brewer(palette = "Set2")
```

Exercise 16 and 17

Cleaning Data

Missing Values

- Missing values are neither negative nor positive infinity like in STATA
- Inf is infinity, also a kind of missing value, and you CAN do size comparison to it
- Finding missing values:
  - Not x == NA but is.na(x)
  - Counting missing values:

```
x <- c(NA,2,NA,2,1)
length(x) #number of all variables
sum(is.na(x)) #number of missing values
sum(!is.na(x)) #number of valid values
```

- Hint: have a look at valid.n() from the prettyR package

```
library(prettyR)
valid.n(x)
```

or write your own function:

```
n.missing <- function(x){
  sum(is.na(x))
}
n.missing(x)
```

Missing Values

Setting values to missing

- R reads numeric blanks as missing
- Remember: when creating factors, non-specified levels will become missing values
- When reading text files you can specify NA by option na.string = c(".","99","999")
- Better: use conditional transformations: age[age == 999] <-NA

Action on Missing Values

- Summary functions return NA unless na.rm=TRUE
- Modeling functions (that accept formula) automatically exclude NAs
- Replacing/Imputing missing values
    - VIM (Visualization and Imputation of Missing Values): useful to find patterns in missing values and visualize them in color maps (colormapMiss()), bar charts (barMiss()) and histograms (histMiss())
    - mice (Multivariate Imputation by Chained Equations): md.pattern() function also searches for patterns of missing values

Exercise 18

Linear regressions

Multiple linear regression

- The general syntax of regression models is rather idiosyncratic:

```
a <- lm(formula)
```

- Basic "formula" syntax

```
y ~ x1 + x2 + … + xK
```

- Endogenous variable is on the left of `~`; exogenous variables are on the right of `~`, separated by `+`
- In R modeling functions: accept formulas, create model objects, generic functions show more, extractor functions show more

Multiple linear regression

Example

```
library(foreign)
x <- read.dta("wave2009.dta")
attach(x)
regr1 <- lm(satisfaction ~ age + netincome + children)
regr1
plot(regr1)
summary(regr1)
names(regr1)
print(unclass(regr1))
```

Multiple linear regression

- The `lm`-object is a list containing:
  1. The estimated coefficients $\hat{\beta}$
  2. The residuals $\hat{u}_t$
  3. The fitted values $\hat{y}_t$
  4. Some other things
- If `a` is an `lm`-object one can access its elements using `coefficients(a)`, `residuals(a)`, `fitted.values(a)`
- Alternatively, one can use the `$`-operator: `a$coefficients`, `a$residuals`, `a$fitted.values`

Extensions (I):

- An intercept is added automatically but can be removed: `lm(y~x1+x2-1)`
- If the variables are organized in an unattached dataframe `x`, one can use the syntax: `lm(formula,data=x)`
- The formula may contain mathematical functions, e.g. `lm(log(y)~log(x1))`
- **Attention**: Squares, sums and differences are not allowed!
- Use the function `I()` for squares, sums and differences

Multiple linear regression

Extensions (II):

- Syntax for interaction terms

```
a <- lm(y ~ x1 + x2 + x1:x2)
```

- Abbreviations:
    - a <-lm(y ~x1*x2)
    - a <-lm(y ~(x1+x2)^2) for all interactions up to 2
- Weights can be added using the option `weights`
- One can select a subset of observations using the option `subset`

Multiple linear regression

Extensions (III):

- The `lm`-object can be used to add a regression line to a plot:

```
regr <- lm(y~x)
plot(x,y)
abline(regr)
```

- The `lm`-object can be used for forecasting:

```
regr <- lm(y~x1+x2)
xn <- data.frame(x1=c(...),x2=c(...))
predict(regr,newdata=xn,se.fit=TRUE)
```

Extensions (IV):

- Heteroskedasticity consistent standard errors are not reported by default
- The package `lmtest` supplies functions for robust standard errors
- The syntax for robust standard errors is

```
coeftest(regr,vcov=vcovHC)
```

Polynomial regression

Polynomial regression

```
y ~ x + I(x^2) + I(x^3)
y ~ poly(x,3)
```

Putting it all together:

```
regr2 <- lm(satisfaction~age+netincome,data=x)
regr3 <- lm(satisfaction~age+I(age^2))
regr4 <- lm(satisfaction~log(netincome))
regr5 <- lm(satisfaction~gender*marital)
z <- gender=="Female"
regr6 <- lm(satisfaction~log(netincome),subset=z)
library("lmtest")
coeftest(regr6,vcovHC)
```

Exercise 19 and 20

Programming

User-defined functions

- One can define new functions in R
- Functions are objects of class `function`
- Each function has a name, one or more inputs (arguments) and one output (return)
- Inputs can be any objects (usually vectors)
- The function can return only one object (which can be a list)
- Variables defined within a function are only local

User-defined functions

Syntax

```
fn <- function(x,y){
  block of commands to compute output out
  return(out)
  }
```

Example

```
utility <- function(cons,gam){
  U <- (cons^(1-gam)-1)/(1-gam)
  return(U)
  }
```

User-defined functions

Example

```
mystats <- function(x) {
  mymean <- mean(x, na.rm=TRUE)
  mysd <- sd(x, na.rm=TRUE)
  c(mean=mymean, sd=mysd) #only last thing is remembered
}
mystats(posttest)
mymean #not found
```

- functions return only a single object, the last one created, but can contain many results
- applying functions by group

```
by(posttest, workshop, mean)
by(posttest, workshop, mystats)
```

Exercise 21

- If the same commands should be executed for different values of some variable, loops are useful
- There are three kinds of loops: `for`, `while`, `repeat`
- By far the most important loop is the `for`-loop
- General syntax:

```
for([var] in vector) {
    [commands]
}
```

- The commands are executed for each value of `vector`

Example

```
z <- rep(NA,10)
for(i in 1:10) {
   z[i] <- i^2
}
print(z)
```

Loops

- Syntax of the `while`-loop:

```
while([condition]) {[commands]}
```

- Syntax of the `repeat`-loop:

```
repeat {[commands]}
```

- The `repeat`-loop does never stop but can be left using the command `break`

Syntax of the `if`-command

```
if([condition]) {
   [commands]
}
```

- The condition must not be a vector (else only its first element is used)
- If there is just a single command, the brackets can be omitted
- The opening curly bracket must appear in the same line as the `if`-command
- It is possible to add `else {[commands]}`

Random numbers

- A large number of standard distributions is implemented in R
- There is a common syntax for cdfs, density functions, quantile functions, and random number generation:

  `pNAME(x,pars)` cumulative distribution function at x
  `dNAME(x,pars)` density (or probability) function at x
  `qNAME(p,pars)` quantile function at p
  `rNAME(n,pars)` generate n random draws
- Here `NAME` is the abbreviated name of the distribution and `pars` are its parameters

Some continuous distribution names:

`norm`	normal
`unif`	uniform
`lnorm`	log-normal
`exp`	exponential
`t`	t-distribution
`chisq`	χ^2-distribution
`F`	F-distribution

Random numbers

Some discrete distribution names:

`binom`	binomial
`pois`	Poisson
`geom`	geometric
`hyper`	hyper-geometric
`nbinom`	negative binomial
`multinom`	multinomial

Random numbers

- Define a vector x on an appropriate grid $[a, b]$
- Plots of cdf and density functions:

```
plot(x,pNAME(x,pars))
plot(x,dNAME(x,pars))
```

- Define a grid vector p on $[0, 1]$; plot of quantile function:

```
plot(x,qNAME(p,pars))
```

Simulations

Example: Simulate the distribution of the moment estimator of the exponential distribution

```
R <- 10000
Z <- rep(NA,R)
for(r in 1:R) {
  x <- rexp(n=10,rate=0.5)
  Z[r] <- 1/mean(x)
}
truehist(Z)
abline(v=2,col="red")
```

Exercises 22, 23, and 24

High Quality Output

High Quality Output

- Paste into word processor
- Use packages, e.g. xtable and texreg
- rtf or R2DOCX to write complex Word docs, but hard to set up
- Reproducable research: knitr and rnotebook

High Quality Output

Example

```
myM1 <- lm(q4 ~ q1 + q2 + q3, data=mydata100)
myM2 <- lm(q4 ~ q1, data=mydata100)

library("xtable")
print(xtable(myM1),type="html",file="myM1—xtable.doc")

library("texreg")
htmlreg(myM1, single.row=TRUE,file="myM1—htmlreg.doc")
htmlreg(list(myM1,myM2), file="myM1—myM2—htmlreg.doc")

texreg(list(myM1, myM2))
```