Universität Bonn
Proseminar Wissenschaftliches Arbeiten
Ökonometrie und Statistik

16. April 2024
Elias Wolf

Einführung in R

Die folgenden Seiten geben eine kurze Einführung in das Software-Package R. Wer sich jedoch noch nie mit Programmiersprachen bzw. Skriptsprachen beschäftigt hat, sollte zunächst ein Tutorial durcharbeiten. Für die ersten Schritte bietet sich zum Beispiel dieser kostenlose Kurs auf Datacamp an: https://www.datacamp.com/community/open-courses/einfuhrung-in-r

Darüberhinaus gibt es auch noch viele verschiedene gute und ausführliche Einführungen zu R im Netz:

- Ökonometrie mit R: https://www.econometrics-with-r.org/
- Eine Referenz-Karte (immer praktisch): HIER
- Viele weitere Cheat Sheets: https://www.rstudio.com/resources/cheatsheets/
- Die offizielle Einführung zu R (sehr detailliert): HIER
- Online-Einführungen: https://education.rstudio.com/
- Ein sehr gutes R-Buch (nicht nur für Fortgeschrittene): http://adv-r.had.co.nz/
- Ein weiteres sehr gutes R-Buch: https://r4ds.had.co.nz/
- Ein interaktives R-Paket zum lernen von R: https://swirlstats.com

Warum R?

- R ist kostenlos zu haben: www.r-project.org
- Es gibt eine hervorragende (ebenfalls kostenlose) Entwicklungsumgebung namens **RStudio**: http://www.rstudio.com/
- R kommt mit sehr mächtigen Werkzeugen zur Erstellung von Grafiken.
- R ist der de-facto Standard für die Statistischen Wissenschaften.

Um R zu installieren, gehen Sie folgendermaßen vor:

- R kann auf der Seite www.r-project.org heruntergeladen werden. Klicken Sie dazu den Menüpunkt `Download`. Dieser führt Sie (wenn Sie z.B. den Göttinger Mirror ausgewählt haben) auf die Seite http://ftp5.gwdg.de/pub/misc/cran/. Hier können Sie R für Windows, Mac oder Linux herunterladen.
- Wenn Sie R installiert haben, brauchen Sie noch die Entwicklungsumgebung RStudio, die Siu unter https://rstudio.com/products/rstudio/download/ herunterladen können. Wählen die Sie Free/Open Source Version.

Wichtig: Installieren Sie erst R und dann RStudio.

Kurzes Glossar

Die Klärung folgender Begriffe ist notwendig, um sich mit R zurecht zu finden:

- Konsole (Console): Die eigentliche Schnittstelle zu R. Erkennbar an dem sogenannten Prompt-Zeichen „>“.
- Skriptdatei: Eine gewöhnliche Textdatei mit der Endung „`.R`“ oder „`.r`“. In dieser Übung werden wir diese Datei einfach `myscript.R` nennen.
- Arbeitsverzeichnis (working directory): Der (Verzeichnis-)Ort von dem aus R „denkt“. Falls z.B. beim Laden von Daten nicht explizit ein anderes Verzeichnis spezifiziert wird, dann nimmt R an, dass die Daten im „Arbeitsverzeichnis“ liegen. Das Arbeitsverzeichnis kann jedoch beliebig gesetzt werden was eine häufige Fehlerquelle darstellt. Nützliche Kommandos sind daher: `getwd()` und `setwd()`. Die Verwendung von R-Studios „Projects“ ist ebenfalls sehr hilfreich in diesem Zusammenhang.
- Workspace: Dies ist eine versteckte Datei (gespeichert als „`.RData`“), welche automatisch im Arbeitsverzeichnis abgelegt wird und alle Objekte (Daten, definierte Matrizen, Vektoren, Funktionen, etc.) speichert. Beim Schließen von R wird man gefragt, ob man den workspace der Sitzung speichern oder löschen möchte. Falls man diesen speichert, dann wird er automatisch mit der nächsten R-Sitzung geladen–vorausgesetzt man startet R im entsprechenden Arbeitsverzeichnis. Nützliche Kommandos: `ls()` zur Anzeige aller Objekte im Arbeitsverzeichnis und `rm(list=ls())` zum Löschen all dieser Objekte.

Erste Schritte: Variablen, Vektoren und Matrizen

Wir öffnen R-Studio und erstellen eine Skriptdatei, z.B., `myscript.R`. In der Datei `myscript.R` können dann alle verwendeten Kommandos gespeichert werden. Die Tastenkombination `STRG+ENTER` erlaubt es zeilenweise (oder mehrere zuvor markierte) Kommandos auszuführen.

Wir beginnen mit einfachsten Rechenaufgaben.

```
2+2  # Ebenso: *,/,-,^2,^3,...

## [1] 4
```

Das Zeichen `#` wird verwendet, um Text/Code „auszukommentieren“, sodass R diesen Text/Code ignoriert. Dies ist sehr nützlich, um seine eigenen R-Codes mit Hinweisen zu versehen.

Der sogenannte Zuweisungs-Operator „`<-`“ (assignment operator) wird das am häufigsten verwendete Kommando sein:

```
x <- 4
x

## [1] 4

4 -> x  # Möglich aber ungewöhnlich
x

## [1] 4
```

Ein etwas interessanteres Daten-Objekt, ein Vektor:

```
y <- c(2,7,4,1)

y

## [1] 2 7 4 1
```

Es gibt viele nützliche Kommandos, um bestimmte Zahlenvektoren zu erzeugen:

```
1:10

##  [1]  1  2  3  4  5  6  7  8  9 10

seq(from=2, to=20, by=2)

##  [1]  2  4  6  8 10 12 14 16 18 20
```

Die einzelnen Einträge eines Vektors spricht man folgendermaßen an:

```
z <- c(2,4,8,11,17)

z[2]

## [1] 4

z[1:3]

## [1] 2 4 8

z[c(1,3)]

## [1] 2 8
```

Das Kommando `ls()` gibt alle im Workspace gespeicherten Objekte auf der Konsole aus. Das Kommando `rm(list=ls())` löscht all diese:

```
ls()

## [1] "x" "y"

rm(list=ls())

ls()

## character(0)
```

Rechnen mit Vektoren und Skalaren in R:

```
x <- 4; y <- c(2,7,4,1)
x*y

## [1]  8 28 16  4

y*y

## [1]  4 49 16  1
```

Multiplikation von Vektoren: Z.B. das äußere Produkt $y\,y'$

```
y %*% t(y)

##      [,1] [,2] [,3] [,4]
## [1,]    4   14    8    2
## [2,]   14   49   28    7
## [3,]    8   28   16    4
## [4,]    2    7    4    1
```

Oder das innere Produkt $y'y$

```
t(y) %*% y

##      [,1]
## [1,]   70
```

Konventionsmäßig behandelt R Vektoren als Spaltenvektoren. Als nächstes führen wir Matrizen ein.

```
A <- matrix(data=1:16, ncol=4, nrow=4)
A

##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

Standardmäßig füllt R Matrizen spaltenweise mit Zahlen auf. Man kann aber auch reihenweise auffüllen:

```
A <- matrix(data=1:16, ncol=4, nrow=4, byrow=TRUE)
A

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

Beachte: eine Matrix hat immer zwei Dimensionen, jedoch ein Vektor nur eine:

```
dim(A)      # Dimensionen der Matrix A

## [1] 4 4

dim(A)[1]   # Zeilendimension der Matrix A

## [1] 4

dim(A)[2]   # Spaltendimension der Matrix A

## [1] 4

dim(y)

## NULL

length(y)   # Länge des Vektors y

## [1] 4
```

Die Elemente einer Matrix lassen sich analog wie die eines Vektors ansprechen:

```
A[,1]       # wählt die erste Spalte der Matrix A aus
A[3,2]      # wählt das Element in der dritten Zeile und zweiten Spalte aus
A[c(1,4),] # wählt die erste und vierte Zeile aus
```

Auch sogenannte „boolesche“ Operationen sind möglich (und sehr wichtig). Zum Beispiel, um zu erfahren, welches Element in der ersten Spalte der Matrix `A` strikt größer als 2 ist:

```
A[,1][A[,1]>2]

# Dies erzeugt einen 'Boolean'-Vektor:
A[,1]>2

# Funktioniert auch in 'sinnlosen' Zusammenhängen:
y[A[,1]>2]
```

Matrizen werden wie folgt miteinander multipliziert:

```
B <- matrix(1:4, ncol=2)
C <- matrix(5:8, ncol=2)

B * C    # elementweise Multiplikation
B %*% C  # Matrizenmultiplikation
```

Aufg 1.) *Erzeugen Sie eine 4×4 Matrix A, sodass alle Elemente der i-ten Spalte den Wert i haben für $i = 1, 2, 3, 4$. Setzen Sie nun alle Elemente der zweiten Zeile auf den Wert 5.*

Weitere Daten-Objekte

Neben den klassischen Daten-Objekten wie Skalaren, Vektoren und Matrizen gibt es drei weitere in R:

1. `array`: Wie eine Matrix nur mit mehreren Dimensionen. Hier ist ein Beispiel eines $2 \times 2 \times 2$-dimensionalen arrays:

```
myFirst_Array <- array(c(1:8), dim=c(2,2,2))
```

2. `list`: In `list`en kann man verschiedene Daten-Typen zusammenschließen.

```
myFirst_List <- list("Some_Numbers" = c(66, 76, 55, 12, 4, 66, 8, 99),
                     "Animals"      = c("Rabbit", "Cat", "Elefant"),
                     "My_Series"    = c(30:1))
```

   Sehr nützlich, um die Struktur von Listen zu erfahren ist die `str()`-Funktion:

```
str(myFirst_List)

## List of 3
##  $ Some_Numbers: num [1:8] 66 76 55 12 4 66 8 99
##  $ Animals     : chr [1:3] "Rabbit" "Cat" "Elefant"
##  $ My_Series   : int [1:30] 30 29 28 27 26 25 24 23 22 21 ...
```

3. Der `data.frame`: Ein `data.frame` ist ein `list`en-Objekt aber mit einigen formalen Restriktionen (z.B. gleiche Anzahl an Zeilen für jede Spalte). Wie der Name schon sagt, ein `data.frame`-Objekt ist dafür gedacht, um Daten zu speichern:

```
myFirst_df <- data.frame("Credit_Default"   = c( 0, 0, 1, 0, 1, 1),
                         "Age"              = c(35,41,55,36,44,26),
                         "Loan_in_1000_EUR" = c(55,65,23,12,98,76))
```

For-Schleifen und if-Bedingungen

Die elementweise Ausführung $y * y$, ist eine der zentralen Stärken von R. So kann man viele Operationen auf jedes Element eines Vektors anwenden.

```
y^2
log(y)
exp(y)
y-mean(y)
(y-mean(y))/sd(y)  # Standardisierung
```

Diese Möglichkeit ist eine zentrales Merkmal von sogenannten Matrix basierten Sprachen wie R (oder z.B. MATLAB). Andere Programmiersprachen müssen hierzu oft Schleifen (loops) verwenden:

```r
N <- length(y)
1:N

y.sq <- numeric(N)
y.sq

for(i in 1:N){
   y.sq[i] <- y[i]^2
}
y.sq
```

Viel eleganter ist jedoch:

```r
y * y
```

Beziehungsweise:

```r
y^2
```

Neben der `for`-Schleife ist auch die `if`-Bedingung sehr wichtig. Betrachten wir dazu folgendes Beispiel: Wir wollen einem gegebenen Vektor y einen Vektor z zuordnen, sodass `z[i]` gleich -1 ist, falls `y[i]` negativ ist, und `z[i]` gleich 1 ist, falls `y[i]` positiv ist.

```r
y <- rnorm(100)  # erzeugt einen Vektor der Länge 100, dessen Einträge
                 # Realisierungen einer standard normalverteilen Zufalls-
                 # variablen sind
N <- length(y)
z <- rep(0,N)    # erzeugt einen Vektor der Länge N,
                 # der überall Nullen als Einträge hat

for(i in 1:N){
   if(y[i] < 0)
     z[i] <- -1
   if(y[i] >= 0)
     z[i] <- 1
}
```

Wiederum kann man dies in R viel eleganter ohne eine Schleife bewerkstelligen:

```r
y <- rnorm(100)
N <- length(y)
z <- rep(0,N)

z[(y < 0)] <- -1
z[(y >= 0)] <- 1
```

Aufg 2.) *Der natürliche Logarithmus einer Zahl x kann mit dem Befehl `log(x)` berechnet werden. Erzeugen Sie einen Vektor y mit den Einträgen $1, 2, \ldots, 100$ und schreiben Sie eine Schleife, welche den Logarithmus für die einzelnen Elemente des Vektors y berechnet. Versuchen Sie zusätzlich, dies ohne eine Schleife zu bewerkstelligen.*

Plotten mit R

Es gibt viele verschiedene Wege, um Plots in R zu erstellen. Wir führen hier nur die grundlegende Funktion `plot()` ein. Dazu erzeugen wir zunächst ein Datenobjekt, das es zu plotten gilt. Insbesondere erzeugen wir 100 Beobachtungen einer normalverteilten Zufallsvariablen mit Mittelwert 0 und Varianz 1:

```
wn <- rnorm(100, mean=0, sd=1)
```

Wir wollen den Vektor `wn` nun plotten.

```
plot(wn)
plot(wn, ylab="", xlab="")
```

Programmieren mit R

In R gibt es eine Vielzahl vorprogrammierter Funktionen, z.B.

```
y <- 1:10

y^2      # Quadrieren
exp(y)   # Exponentialfunktion
log(y)   # Logarithmus
mean(y)  # Mittelwert
sd(y)    # Standardabweichung
```

Der wohl größte Vorteil von R ist, dass man sehr einfach eigene Funktionen schreiben kann. Im Prinzip kann man sich so JEDE erdenkliche Routine selbst programmieren, wie zum Beispiel eine eigene `mean()`-Funktion:

```
mittelwert <- function(x){
   x.len  <- length(x)
   result <- sum(x)/x.len
   return(result)
   }

wn <- rnorm(100, mean=2, sd=2)
mittelwert(x=wn)
```

Aufg 3.) *Für einen Datenvektor Datenvektor $(X_1, \ldots, X_n)$ ist die empirische Varianz gegeben durch $\widehat{V} = n^{-1} \sum_{i=1}^{n} (X_i - \overline{X})^2$, wobei $\overline{X} = n^{-1} \sum_{i=1}^{n} X_i$ der empirische Mittelwert ist. Schreiben Sie eine eigene Funktion, die die empirische Varianz berechnet. Erzeugen Sie 100 Beobachtungen einer normalverteilten Zufallsvariablen mit Mittelwert 1 und Varianz 4 und wenden Sie Ihre Funktion auf die erzeugten Daten an.*

Speichern und Laden von Datensätzen

Wir erzeugen zunächst einen kleinen Datensatz bestehend aus zehn standardnormalverteilten Zufallsvektoren der Länge 100.

```r
wn.data <- matrix(0,ncol=10,nrow=100)
for(i in 1:10){
   wn.data[,i] <- rnorm(100)
}
```

Man kann diesen Datensatz nun folgendermaßen abspeichern:

```r
write.table(wn.data, file="WN.txt", row.names=FALSE, col.names=FALSE)
```

Um den Datensatz in R zu laden, benutzt man den folgenden Befehl:

```r
data <- read.table("WN.txt", header=FALSE)
```

Der Befehl `read.table()` erlaubt es, `txt`-Dateien in R einzulesen. Daneben gibt es noch Befehle, um andere Dateiformate einzulesen, z.B. `read.csv()` für `csv`-Dateien.

Simulationen

Wir wollen nun eine erste einfache Simulationsstudie mit R durchführen. Insbesondere wollen wir den empirischen Mittelwert (als Schätzer für den theoretischen Erwartungswert) untersuchen. Die Anzahl der Simulationen soll $M = 5000$ betragen. In jedem Simulationslauf gehen wir folgendermaßen vor:

- Wir erzeugen einen Datensatz $X_1, \ldots, X_n$ der Länge $n = 100$, wobei X_i unabhängige, normalverteilte Zufallsvariablen mit Erwartungswert $\mu = 2$ und Standardabweichung $\sigma = 1$ sind.
- Auf Grundlage der simulierten Daten $X_1, \ldots, X_n$ berechnen wir den empirischen Mittelwert $\hat{\mu} = n^{-1} \sum_{i=1}^{n} X_i$.
- Das wiederholen wir $M = 5000$ mal und zwar jedes mal mit neu simulierten Daten. Wir erhalten schließlich $M = 5000$ verschiedene Mittelwerte $\hat{\mu}^{(1)}, \ldots \hat{\mu}^{(5000)}$

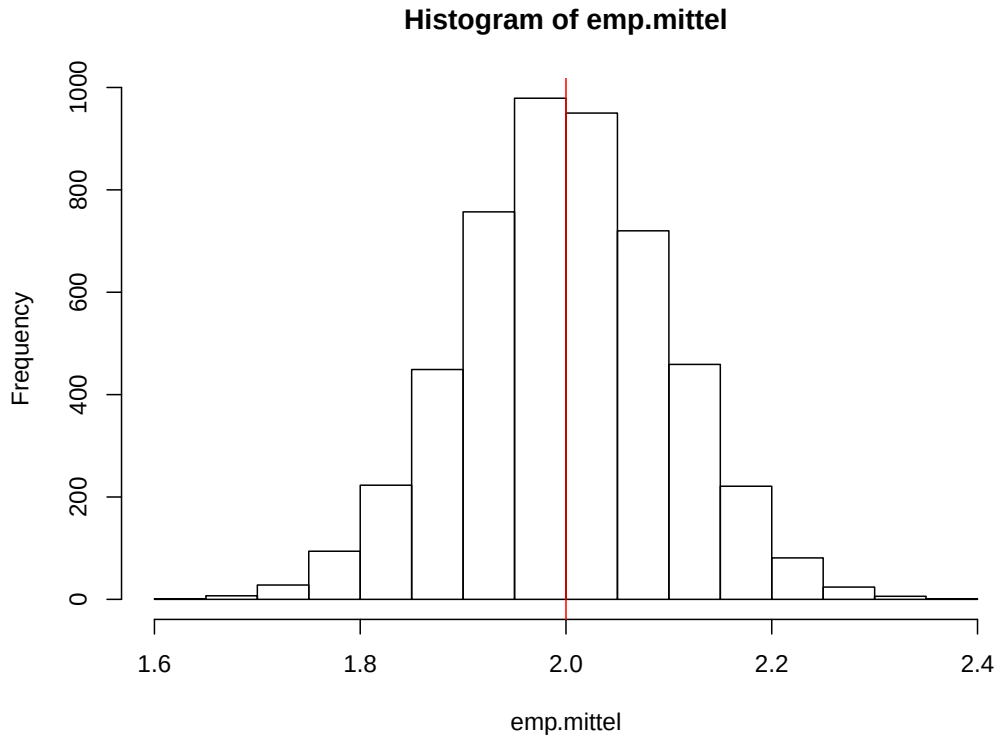
Die berechneten empirischen Mittelwerte $\hat{\theta}^{(k)}$ für die Simulationsläufe $k = 1, \ldots, M$ speichern wir als langen Vektor ab und plotten ein Histogramm dieser Werte, um sie mit dem tatsächlichen Erwartungswert $\mu = 2$ zu vergleichen.

```r
M <- 5000
n <- 100
emp.mittel <- rep(0,N)

for(k in 1:M){
   data <- rnorm(n,mean=2,sd=1)
   emp.mittel[k] <- mean(data)
}

mean(emp.mittel)
sd(emp.mittel)

hist(emp.mittel)
abline(v=2,col="red")
```



R-Hilfe

Zu guter Letzt sei noch angemerkt, dass R zu jedem Befehl eine Dokumentation umfasst, die man mit Hilfe des Befehls `help()` oder `?` aufrufen kann. Z.B.

```
help(read.table)
?rnorm
```