4 - Dataframes und Daten einlesen

author: Benedict Witzenberger date: 17. April 2019

Recap

Ich habe fÃ¼r gestern einen kleinen Test vorbereitet.

Ihr findet ihn unter: XYZ.de

Versucht euch an den Aufgaben. Wenn es Fragen gibt, gebt Bescheid.

Was wir heute vorhaben

- Subsetting und Arbeiten mit Dataframes
- Packages installieren und laden
- Daten einlesen
- Daten vorbereiten
- Guter Stil
- Conditional Flows

Subsetting

Wir kÃ¶nnen Daten auf verschiedene Arten zuschneiden (= subsetting)

Das BlÃ¶de daran: R hat verschiedene Konzepte, die teilweise zusammenspielen, aber dann bei verschiedenen Datentypen verschieden reagieren.

Deswegen fangen wir langsam an:

```
mein_vector <- c("a", "b", "c")
mein_vector[1]
```

```
[1] "a"
```

[1] ruft das erste Element in einem Vektor auf. Wir nennen die 1 “Index”.

R zÃ¤hlt im Unterschied zu vielen anderen Programmiersprachen ab 1, nicht ab 0.

Subsetting bei einem Vektor: Positiv

Positive Zahlen geben Elemente an den angegebenen Positionen zurÃ¼ck:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[c(1,4)]
```

```
[1] 2.1 5.4
```

```
x[order(x)] # aufsteigend sortiert
```

```
[1] 2.1 3.3 4.2 5.4
```

```
x[c(1, 1)] # doppelte Integers geben das Ergebnis doppelt zurÃ¼ck
```

```
[1] 2.1 2.1
```

```r
x[c(1.01, 1.9)] # double Zahlen werden in Integers umgewandelt
```

```
[1] 2.1 2.1
```

Subsetting bei einem Vektor: Negativ

Negative Integers geben Elemente zurÃ¼ck, die nicht angegeben wurden.

```r
x[-c(3, 1)]
```

```
[1] 4.2 5.4
```

```r
x[-(1:2)]
```

```
[1] 3.3 5.4
```

Positive und negative Integers dÃ¼rfen beim Subsetting nicht gemischt werden.

Subsetting bei einem Vektor: Logical

TRUE und FALSE helfen ebenfalls beim Subsetting und geben die zutreffenden Werte zurÃ¼ck:

```r
x[c(TRUE, FALSE, TRUE, FALSE)]
```

```
[1] 2.1 3.3
```

Ein zu kurzer Vektor wird recyclet:

```r
x[c(TRUE, FALSE)]
```

```
[1] 2.1 3.3
```

Das ist vor allem beim Filtern hilfreich, wo TRUE und FALSE im Hintergrund arbeiten:

```r
x[x > 3]
```

```
[1] 4.2 3.3 5.4
```

Subsetting bei einem Vektor: Sonstiges

Ein `NA` gibt immer eine `NA` zurÃ¼ck. `x[c(TRUE, FALSE, NA, FALSE)]`

Leeres Subsetting bei einem Vektor gibt alles zurÃ¼ck: `x[]`

Ein Subsetting mit [0] gibt einen leeren Vektor zurÃ¼ck.

Subsetting bei einem Vektor: Namen

Ein Vektor mit Namen, kann auch Ã¼ber die Namen gesubsettet werden:

```r
names(x) <- c("a", "b", "c", "d")

x[c("a", "d")]
```

```
  a   d
2.1 5.4
```

Subsetting bei Matrizen I

Matrizen und Arrays kÃ¶nnen reacht einfach angepsrochen werden:

`matrix[Zeilen, Spalten]`

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
```

```
     A B C
[1,] 1 4 7
[2,] 2 5 8
```

Jetzt funktionert auch leeres Subsetting - dann werden alle Zeilen oder Spalten zurÃ¼ckgegeben.

Subsetting bei Matrizen II

```
a[c(TRUE, FALSE, TRUE), c("B", "A")]
```

```
     B A
[1,] 4 1
[2,] 6 3
```

Subsetting bei Dataframes I

```
df <- data.frame(x = 1:4, y = 4:1, z = letters[1:4])
df
```

```
  x y z
1 1 4 a
2 2 3 b
3 3 2 c
4 4 1 d
```

Subsetting bei Dataframes II

```
df[1,1]
```

```
[1] 1
```

```
df[c(1, 3), ]
```

```
  x y z
1 1 4 a
3 3 2 c
```

```
df$x
```

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

```
df[df$x == 2, ]
```

```
  x y z
2 2 3 b
```

Subsetting bei Dataframes III

Subsetten wir einen Dataframe mit einem einzelnen Vector, verhÃ¤lt er sich wie eine Liste, bei zwei Vektoren, wie eine Matrix.

```
df[c("x", "z")] # wie bei einer Liste
```

```
  x z
1 1 a
2 2 b
3 3 c
4 4 d
```

```
df[, c("x", "z")] # wie bei einer Matrix
```

```
  x z
1 1 a
2 2 b
3 3 c
4 4 d
```

Subsetting bei Dataframes IV: [[]]

Es gibt zwei weitere Subsetting-Operatoren, die bei Dataframes und Listen sehr wichtig werden: `$` und `[[]]`

[[]]

```
a <- list(a = 1, b = 2)
a[1]
```

```
$a
[1] 1
```

```
a[[1]]
```

```
[1] 1
```

> "If list x is a train carrying objects, then x[[5]] is the object in car 5; x[4:6] is a train of cars 4-6."
>
> (Quelle: @RLangTip auf Twitter)

Preserving vs. Simplifying

Beim Subsetting gehen Informationen verloren. Es ist wichtig, dass wir das kontrollieren kÃ¶nnen. Vor allem beim Programmieren, wollen wir meistens, dass der Datentyp des Inputs gleich dem Output ist.

	Vereinfachend	Erhaltend
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]
Factor	x[1:4, drop = T]	x[1:4]
Array	x[1,] oder x[, 1]	x[1, , drop = F] oder x[, 1, drop = F]
Dataframe	x[, 1] or x[[1]]	x[, 1, drop = F] oder x[1]

Beispiel Vektor: Namen gehen verloren

```
> x <- c(a = 1, b = 2)
```

```
> x[[1]] # vereinfachend
[1] 1
> x[1] # erhaltend
a
1
```

Weitere Probleme beim Subsetting

- Liste: Gibt nur das Element zurÃ¼ck, nicht das Element in der Liste
- Factor: Wirft alle ungenutzen Levels weg
- Matrix: Wenn nur noch eine Deminsion Ã¼brig bleibt, geht die Matrix verloren:

```
a <- matrix(1:4, nrow = 2)
a[1, , drop = FALSE]
```

```
     [,1] [,2]
[1,]    1    3
```

```
a[1,]
```

```
[1] 1 3
```

- Dataframes: Wenn nur noch eine Spalte/Zeile Ã¼brig bleibt, wird die als Vektor ausgegeben

Subsetting bei Dataframes IV: $

$

x$y ist die Kurzschreibung fÃ¼r `x[["y", exact = FALSE]]`

```
a <- list(a = 1, b = 2)
a$a
```

```
[1] 1
```

Besonders praktisch bei Dataframes:

```
df <- data.frame(a = 1:4, b = 4:1, c = letters[1:4])
df$a
```

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

```
df$c
```

```
[1] a b c d
Levels: a b c d
```

Beliebter Fehler

Die $-methode funktioniert nicht, wenn der Spaltenname in einer Variable liegt:

```
var <- "a"

df$var
```

```
NULL
```

Stattdessen geht:

```
df[[var]]
```

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

Out of Bounds

Wollen wir ein Vektor-Element aufrufen, das nicht exisitert, bekommen wir bei [] meistens NA oder 0 zurÃ¼ck, bei [[]] meist direkt einen Fehler.

Subsetting und Zuweisen

Wir kÃ¶nnen Variablenzuweisung und Subsetting kombinieren

```
df[df$a ==2, ]$b
```

```
[1] 3
```

```
df[df$a ==2, ]$b <- "Benedict"
df[df$a ==2, ]$b
```

```
[1] "Benedict"
```

Oder:

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
```

```
[1] 2 3 3 4 5
```

Wozu kÃ¶nnen wir das brauchen?

Drei Anwendungsbeispiele:

Lookup-Table

Match und Merge

Samplen

Dataframes: order, character subsetting und logical subsetting

Anwendung: Lookup-Table

Wir kÃ¶nnen mithilfe eines Vektor Werte zuordnen:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "MÃ¤nnlich", f = "Weiblich", u = NA)
```

Frage: Wie wÃ¼rdet ihr jetzt die Werte von x in lookup nachschlagen?

Anwendung: Lookup-Table LÃ¶sung

Wir kÃ¶nnen mithilfe eines Vektor Werte zuordnen:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "MÃ¤nnlich", f = "Weiblich", u = NA)
lookup[x]
```

```
         m          f          u          f          f          m
"MÃ¤nnlich" "Weiblich"         NA "Weiblich" "Weiblich" "MÃ¤nnlich"
         m
"MÃ¤nnlich"
```

Anwendung: Match und Merge

```
noten <- c(1, 2, 2, 5, 3, 1, 4, 2, 6)

bewertung <- data.frame(
  note = 1:6,
  worte = c("Sehr gut", "gut", "befriedigend", "ausreichend", "mangelhaft", "ungenÃ¼gend"),
  durchgefallen = c(F, F, F, F, T, T)
)
```

Wir wollen jetzt die Informationen aus `noten` und `bewertung` zusammenbringen (mergen). Das geht mit zwei Varianten...

Variante 1: match()

```
id <- match(noten, bewertung$note)
bewertung[id, ]
```

```
    note        worte durchgefallen
1      1     Sehr gut         FALSE
2      2          gut         FALSE
2.1    2          gut         FALSE
5      5   mangelhaft          TRUE
3      3 befriedigend         FALSE
1.1    1     Sehr gut         FALSE
4      4  ausreichend         FALSE
2.2    2          gut         FALSE
6      6   ungenÃ¼gend          TRUE
```

Variante 2: rownames()

```
rownames(bewertung) <- bewertung$note
bewertung[as.character(noten), ]
```

```
    note        worte durchgefallen
1      1     Sehr gut         FALSE
2      2          gut         FALSE
2.1    2          gut         FALSE
5      5   mangelhaft          TRUE
3      3 befriedigend         FALSE
1.1    1     Sehr gut         FALSE
4      4  ausreichend         FALSE
2.2    2          gut         FALSE
6      6   ungenÃ¼gend          TRUE
```

Hinweis

Die Funktion `merge()` kann bei einem solchen Problem auch helfen.

Anwendung: Samplen

Wir wollen aus einem Dataframe zufÃ¤llige Zeilen ziehen.

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])

set.seed(10)

df[sample(nrow(df)), ]
```

```
  x y z
4 2 3 d
2 1 5 b
5 3 2 e
3 2 4 c
1 1 6 a
6 3 1 f
```

`set.seed()` sorgt dafÃ¼r, dass Zufallszahlen in Skripten reproduzierbar bleiben.

Samplen I

Wir ziehen drei zufÃ¤llige Zeilen:

```
df[sample(nrow(df), 3), ]
```

```
  x y z
2 1 5 b
6 3 1 f
3 2 4 c
```

Mit ZurÃ¼cklegen:

```
df[sample(nrow(df), 3, replace = T), ]
```

```
    x y z
3   2 4 c
4   2 3 d
4.1 2 3 d
```

Anwendung Dataframes: Order

```
x <- c("b", "c", "a")
order(x)
```

```
[1] 3 1 2
```

```
x[order(x)]
```

```
[1] "a" "b" "c"
```

Um GleichstÃ¤nden vorzubeugen kÃ¶nnen wir auch mehrere Variablen sortieren lassen.

Anwendung Dataframes: character subsetting

Oder: Eine Spalte entfernen

DafÃ¼r gibt es zwei Wege: Eine Spalte kann auf `NULL` gesetzt werden:

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])
df$z <- NULL
```

Oder: Wir wÃ¤hlen nur die Spalten aus, die wir gerne hÃ¤tten:

```
df[c("x", "y")]
```

```
  x y
1 1 6
2 1 5
3 2 4
4 2 3
5 3 2
6 3 1
```

Anwendung Dataframes: logical subsetting

Oder: Filtern

DafÃ¼r brauchen wir zunÃ¤chst einmal die Befehle fÃ¼r Vergleiche:

Befehl	Bedeutung
!x	Nicht x
& oder &&	UND
|	oder
x %in% vector	Vektor enthÃ¤lt x
is.na(x)	NAs in x
>	grÃ¶Ãßer als
<	kleiner als
>=	ist grÃ¶Ãßer gleich
<=	ist kleiner gleich
==	ist gleich
!=	ist nicht gleich

`!(X & Y) == !X | !Y !(X | Y) == !X & !Y`

Achtung: Die doppelten && und || funktionieren bei Vektoren anders: Da prÃ¼fen sie nur das erste Element.

Filtern

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])
df[df$x == 3, ]
```

```
  x y z
5 3 2 e
6 3 1 f
```

```
df[df$x == 3 & df$y == 2, ]
```

```
  x y z
5 3 2 e
```

```
df[df$x == 3 & df$z != "e", ]
```

```
  x y z
6 3 1 f
```

Subset()

Wir kÃ¶nnen auch die Funktion `subset()` zum Subsetting verwenden:

```
subset(df, x == 3 & z != "e")
```

```
  x y z
6 3 1 f
```

Ãœbung: Booleans

Was sind die Ergebnisse dieser Tests? Schreibt sie euch auf, wir vergleichen:

1. `TRUE == FALSE`
2. `TRUE != FALSE`
3. `!(TRUE | FALSE)`
4. `FALSE != !(FALSE)`
5. `df <- data.frame(a = c(1, 2, 3), b = c(3, 2, 1), c = c("a", "b", "c"))`
   `df[df$a != 2,]$c`
6. gleicher df: `df[df$c != "a" & df$a != 3,]$a`

LÃ¶sung: Booleans I

```
TRUE == FALSE
```

```
[1] FALSE
```

```
TRUE != FALSE
```

```
[1] TRUE
```

```
!(TRUE | FALSE)
```

```
[1] FALSE
```

```
FALSE != !(FALSE)
```

```
[1] TRUE
```

LÃ¶sung: Booleans II

```
df <- data.frame(a = c(1, 2, 3), b = c(3, 2, 1), c = c("a", "b", "c"))
df[df$a != 2,]$c
```

```
[1] a c
Levels: a b c
```

```
df[df$c != "a" & df$a != 3,]$a
```

```
[1] 2
```

Packages installieren

Viele R-Beispiele kommen mit den gleichen DatensÃ¤tzen.

Das Packet ggplot2 installiert sie:

`install.package(ggplot2)`

Pakete werden in R ganz zu Beginn des Skripts geladen:

```
library(ggplot2)
```

Wo kommen R-Packages her?

CRAN

"Comprehensive R Archive Network" managt die offiziellen R-Versionen und die Packages

- installierbar mit `install.package()` oder `install.packages()` oder RStudio
- Packages durchlaufen PrÃ¼fung bei Upload
- Ã¼ber 14.000 Packages verfÃ¼gbar

Github

installierbar mit dem Package `devtools` und `install_github("DeveloperName/PackageName")`

- Unfertige Versionen
- KÃ¶nnen Fehler enthalten
- Work in Progress -> schneller verfÃ¼gbar, einfacher eigene Anmerkungen und VerÃ¤nderungen zu machen/vorzuschlagen
- keine zentrale Ãœbersicht, man muss die Projekte selbst finden

Packages updaten

CRAN

`update.packages()`

Github

einfach nochmal `install_github("DeveloperName/PackageName")` laufen lassen

Bekannte DatensÃ¤tze

Es gibt zahlreiche BeispieldatensÃ¤tze, die mit R, bzw. Packages mitgeliefert werden:

Name in R	Beschreibung	VerfÃ¼gbar mit
AirPassengers	Monthly Airline Passenger Numbers 1949-1960	R
ChickWeight	Weight versus age of chicks on different diets	R
EuStockMarkets	Daily Closing Prices of Major European Stock Indices, 1991-1998	R
iris / iris3	Edgar Anderson's Iris Data	R
mtcars	Motor Trend Car Road Tests	R
Titanic	Survival of passengers on the Titanic	R
uspop	Populations Recorded by the US Census	R
diamonds	Prices of 50,000 round cut diamonds	ggplot2
economics / economics_long	US economic time series	ggplot2
faithfuld	2d density estimate of Old Faithful data (Geysir)	ggplot2
mpg	Fuel economy data from 1999 and 2008 for 38 popular models of car	ggplot2
nycflights13	Airline on-time data for all flights departing NYC in 2013 with 'metadata' on airlines, airports, weather, and planes.	eigenes Package

Alle DatensÃ¤tze, die wir bereitstehen haben, finden wir mit `data()`

Wichtige Funktionen fÃ¼r Dataframes

- head: Zeigt die ersten sechs Zeilen eines Dataframes an
- tail: Zeigt die letzten sechs Zeilen eines Dataframes an
- summary: Zeigt eine Zusammenfassung Ã¼ber alle Spalten hinweg an.

Best-Practise: Ein frisch eingelesener Dataframe sollte immer mit diesen Funktionen untersucht werden, damit alles stimmt.

Ãœbung: Bekannte DatensÃ¤tze ausprobieren

Nehmt euch etwas Zeit, ladet einen der DatensÃ¤tze mit:

```
library(datasets)
data([DATENSATZ])
DATENSATZ
```

- Wie ist der Datensatz aufgebaut?
- Welche Variablen enthÃ¤lt er?
- Findet ihr bereits spannende Ergebnisse?

Daten einlesen

R kann viele verschiedene Datentypen einlesen.

Die gÃ¤ngigsten sind:

- TXT
- TSV

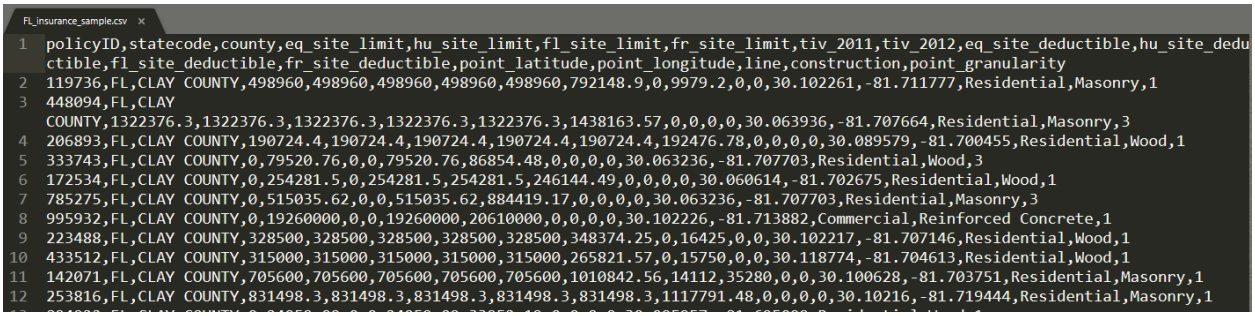

Figure 1:

- CSV
- Excel
- SPSS
- JSON
- RData

Wo sind wir?

Immer aufpassen, wo sich R gerade in unserem Dateisystem befindet:

`getwd()`

`setwd()`

`here::here()` schafft relative Pfade im Projekt. Gut, wenn wir das Projekt mit anderen teilen wollen.

Comma separated Values

Datenjournalistinnens Liebling

Trennung von einzelnen Daten durch `,` oder `;` (im deutschsprachigen Raum)

In R laden mit: `read.csv()` (Kommatrennung) oder `read.csv2()` (Semikolontrennung)

read.csv(): Argumente

basiert auf `read.table()`, bei dem wir den Trenner hÃ¤ndisch einstellen kÃ¶nnen.

`header`: TRUE oder FALSE, falls die erste Zeile die Spaltentitel enthÃ¤lt

`sep`: Der Trenner, hier per default ein Komma

`quote`: Die Zeichen, die dafÃ¼r sorgen, dass SÃ¤tze mit Kommas nicht getrennt werden.

`dec`: Das Zeichen, das fÃ¼r Dezimaltrennung verwendet wird

`col.names`: eigene Titel fÃ¼r die Spalten als Vektor (`colClasses`fixiert die Klassen)

`nrows`: maximale Anzahl der Zeilen, die eingelesen werden sollen (im Gegensatz zu `skip`, das Zeilen auslÃ¤sst)

`strip.white`: LÃ¶scht Leerzeichen vor und nach Texten

`stringsAsFactors`: Wenn FALSE werden Strings nicht automatisch zu Factors umgewandelt -> empfohlen!

`fileEnconding`: Setzt manuell das Encoding (oft UTF-8). HÃ¤ufiges Problem beim Wechsel zwischen Win und Mac

Nutzung von read.csv()

```
df <- read.csv("DATEIPFAD.csv", ARGUMENTE)
```

Weitere read.table()-Abwandlungen

FÃ¼r Text-Files

- `read.table()` flexible Ãœberfunktion fÃ¼r Dateien mit festgelegtem Trenner
- `read.delim()` tab-separiert, mit . als Dezimalseparator
- `read.delim2()` tab-separiert, mit , als Dezimalseparator
- `read.csv()` komma-separiert, mit . als Dezimalseparator
- `read.csv2()` semikolon-separiert, mit , als Dezimalseparator
- `read.fwf()` festgelegte Zahl an Bytes pro Spalte

Bei schwierigeren Dateiformaten:

`readLines()`: hier lesen wir eine Datei Zeile fÃ¼r Zeile ein und kÃ¶nnen genau steuern, wie die Zeile vearbeitet wird

Dateien schreiben

Zu jeder read-Funktion gibt es auch eine write-Funktion:

```
write.csv()
write.csv2()
write.fwf()
write.delim()
write.delim2()
```

Alternative: readr-Package

Als Variante der Base R-Einlesefunktionen gibt es das Package `readr` von Hadley Wickham.

Es ist 10x schneller und funktioniert genauso, wie die Base-Funktionen. Bringen also was, wenn wir groÃŸe oder viele Dateien einlesen wollen.

Beispiele:

```
read_csv()
read_csv2()
read_tsv()
```

==========

Ãœbrigens: Die read-Funktionen funktionieren auch mit Dateien aus dem Internet

```
readr::read_tsv("http://www.sthda.com/upload/boxplot_format.txt")
```

```
# A tibble: 72 x 3
   Nom   variable Group
```

```
   <chr>    <int> <chr>
 1 IND1        10 A
 2 IND2         7 A
 3 IND3        20 A
 4 IND4        14 A
 5 IND5        14 A
 6 IND6        12 A
 7 IND7        10 A
 8 IND8        23 A
 9 IND9        17 A
10 IND10       20 A
# ... with 62 more rows
```

Excel

StandardmÃ¤ssig kann R keine Exceldateien lesen.

Es gibt aber - wie so oft - ~~ein~~ drei Packages dafÃ¼r: XLConnect, readxl und xlsx.

Wir sollten uns nur fÃ¼r eines entscheiden.

```
install.package(xlsx)
```

```
read.xlsx(DATEI, ARBEITSBLATT, header=TRUE, colClasses=NA)
read.xlsx2(DATEI, ARBEITSBLATT, header=TRUE, colClasses="character")
```

read.xlsx2 ist schneller bei groÃŸen Dateien, read.xlsx versucht die Klassen der Spalten zu erhalten

Schreiben:

```
write.xlsx(x, DATEI, sheetName="ARBEITSBLATT", col.names=TRUE, row.names=TRUE, append=FALSE)

write.xlsx2(x, DATEI, sheetName="ARBEITSBLATT",col.names=TRUE, row.names=TRUE, append=FALSE)
```

VerrÃ¼cktere Formate

Eine kleine Zusammenstellung, welche Befehle und Packages bestimmte Dateiformate Ã¶ffnen kÃ¶nnen

Format	Befehl	Package
JSON	fromJSON()	rjson oder jsonlite
XML	xmlTreeParse()	XML
HTML	readHTMLTable(url)	RCurl und XML, oder rvest
SPSS (SAV)	read.spss()	foreign
Stata (dta)	read.dta()	foreign
SAS	read.sas7bdat()	sas7bdat
RData	load() oder readRDS()	keines

Was natÃ¼rlich auch noch mÃ¶glich ist: Verbindung zu Datenbanken oder Webscraping

Cheaterpackage `Data.table`: `fread()` rÃ¤t alle Argumente von selbst und ist sehr schnell

Daten konvertieren

- `as.numeric`

- `as.integer`
- `as.character`
- `as.logical`
- `as.factor`
- `as.ordered`
- `as.Date`
- `as.POSIXct`

Trick: Wir kÃ¶nnen auf einen Blick alle Klassen eines Dataframes mit `sapply(df, class)` bekommen

Daten in Dataframe konvertieren: mtcars

> The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973â€“74 models).

```
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Factors bearbeiten: mtcars I

Variable als Factor

```
mtcars$gear <- as.ordered(mtcars$gear)

class(mtcars$gear)
```

```
[1] "ordered" "factor"
```

Factor umcodieren

am: Transmission (0 = automatic, 1 = manual)

```
recode <- c(automatic = 0, manual = 1)
factor(mtcars$am, levels = recode, labels = names(recode))
```

```
 [1] manual    manual    manual    automatic automatic automatic automatic
 [8] automatic automatic automatic automatic automatic automatic automatic
[15] automatic automatic automatic manual    manual    manual    automatic
[22] automatic automatic automatic automatic manual    manual    manual
[29] manual    manual    manual    manual
Levels: automatic manual
```

Factors bearbeiten: mtcars II

```
mtcars$am <- factor(mtcars$am, levels = recode, labels = names(recode))
plot(mtcars$am, mtcars$mpg)
```

POSIXct

- speichert Datum und Zeit
- berechnet ab dem 1. Januar 1970 00:00 Uhr

```
aktuelle_zeit <- Sys.time()
class(aktuelle_zeit)
```

```
[1] "POSIXct" "POSIXt"
```

```
aktuelle_zeit
```

```
[1] "2019-04-13 17:30:40 CEST"
```

Die Umwandlung von POSIXct ist kompliziert, denn man muss Zeitzonen und Formate im Blick behalten. Aber: as.POSIXct kann das alles handlen.

POSIXct umwandeln

Ein Beispiel: Wir haben ein deutsches Datum und wollen das ins POSIXct-Format umwandeln

```
de_datum <- "16.03.1991 02:10"
as.POSIXct(de_datum, format = "%d.%m.%Y %H:%M", tz = "MET")
```

```
[1] "1991-03-16 02:10:00 MET"
```

Ãœbung: Dateien Ã¶ffnen

Auf dem Github-Repo liegen drei originale DatensÃ¤tze in CSV und XLSX.

Ã–ffnet sie, und versucht die Spaltennamen und Spaltenformate richtig anzugeben. Oder wandelt die Dateien nach dem einlesen so um, dass ihr damit weiterarbeiten kÃ¶nnt.

Die drei Dateien sind etwas unterschiedlich - vor allem unterschiedlich groÃŸ.

Empfohlene Reihenfolge nach aufsteigender Schwierigkeit:

Bierpreis, FahrgastzÃ¤hlung, Wohnungen

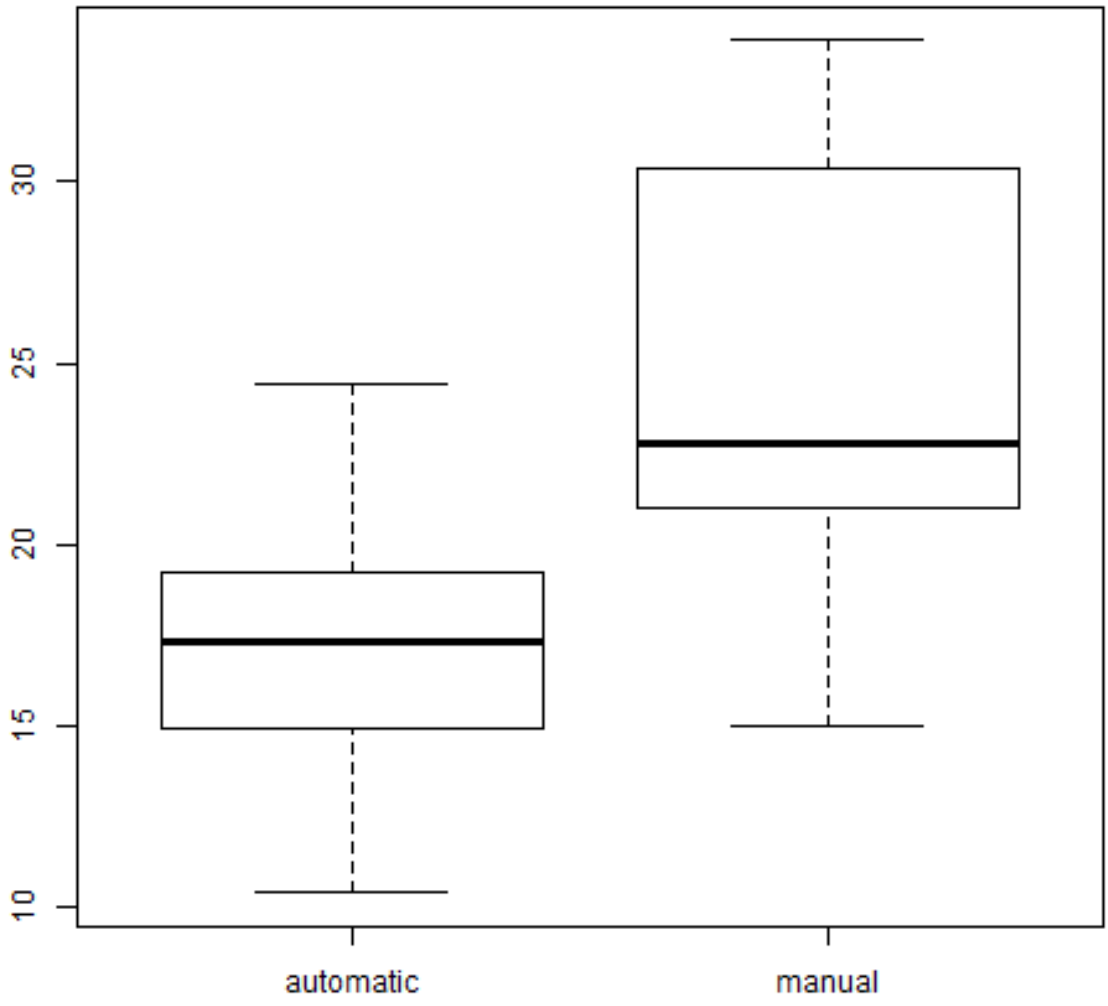

Figure 2: plot of chunk unnamed-chunk-41