

4 - Dataframes und Daten einlesen

author: Benedict Witzenberger date: 17. April 2019

Recap

Ich habe für 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 können Daten auf verschiedene Arten zuschneiden (= subsetting)

Das Blöde 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 zählt im Unterschied zu vielen anderen Programmiersprachen ab 1, nicht ab 0.

Subsetting bei einem Vektor: Positiv

Positive Zahlen geben Elemente an den angegebenen Positionen zurück:

```
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 zurück
```

```
[1] 2.1 2.1
```

```
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 zurück, die nicht angegeben wurden.

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

```
[1] 4.2 5.4
```

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

```
[1] 3.3 5.4
```

Positive und negative Integers dürfen beim Subsetting nicht gemischt werden.

Subsetting bei einem Vektor: Logical

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

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

```
[1] 2.1 3.3
```

Ein zu kurzer Vektor wird recyclet:

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

```
[1] 2.1 3.3
```

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

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

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

Subsetting bei einem Vektor: Sonstiges

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

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

Ein Subsetting mit `[0]` gibt einen leeren Vektor zurück.

Subsetting bei einem Vektor: Namen

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

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

```
  a  d
2.1 5.4
```

Subsetting bei Matrizen I

Matrizen und Arrays können recht einfach angesprochen 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 funktioniert auch leeres Subsetting - dann werden alle Zeilen oder Spalten zurückgegeben.

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, verhält 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 können. Vor allem beim Programmieren, wollen wir meistens, dass der Datentyp des Inputs gleich dem Output ist.

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

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 zurück, nicht das Element in der Liste
- Factor: Wirft alle ungenutzten Levels weg
- Matrix: Wenn nur noch eine Dimension übrig 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 übrig bleibt, wird die als Vektor ausgegeben

Subsetting bei Dataframes IV: \$

\$

`x$y` ist die Kurzschreibung für `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 existiert, bekommen wir bei `[]` meistens NA oder 0 zurück, bei `[[]]` meist direkt einen Fehler.

Subsetting und Zuweisen

Wir können 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 können wir das brauchen?

Drei Anwendungsbeispiele:

Lookup-Table

Match und Merge

Samplen

Dataframes: order, character subsetting und logical subsetting

Anwendung: Lookup-Table

Wir können mithilfe eines Vektor Werte zuordnen:

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

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

Anwendung: Lookup-Table Lösung

Wir können mithilfe eines Vektor Werte zuordnen:

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

```
      m      f      u      f      f      m
"Männlich" "Weiblich" NA "Weiblich" "Weiblich" "Männlich"
      m
"Männlich"
```

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", "ungenügend"),
  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	ungenügend	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	ungenügend	TRUE

Hinweis

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

Anwendung: Samplen

Wir wollen aus einem Dataframe zufällige 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 dafür, dass Zufallszahlen in Skripten reproduzierbar bleiben.

Samplen I

Wir ziehen drei zufällige Zeilen:

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

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

Mit Zurücklegen:

```
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 Gleichständen vorzubeugen können wir auch mehrere Variablen sortieren lassen.

Anwendung Dataframes: character subsetting

Oder: Eine Spalte entfernen

Dafür 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 wählen nur die Spalten aus, die wir gerne hätten:

```
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

Dafür brauchen wir zunächst einmal die Befehle für Vergleiche:

Befehl	Bedeutung
!x	Nicht x
& oder &&	UND
	oder
x %in% vector	Vektor enthält x
is.na(x)	NAs in x
>	größer als
<	kleiner als
>=	ist größer 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 prüfen 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 können auch die Funktion `subset()` zum Subsetting verwenden:

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

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

Übung: 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`

Lösung: Booleans I

```
TRUE == FALSE
```

```
[1] FALSE
```

```
TRUE != FALSE
```

```
[1] TRUE
```

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

```
[1] FALSE
```

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

```
[1] TRUE
```

Lösung: 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 Datensätzen.

Das Paket 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 Prüfung bei Upload
 - über 14.000 Packages verfügbar
-

Github

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

- Unfertige Versionen
- Können Fehler enthalten
- Work in Progress -> schneller verfügbar, einfacher eigene Anmerkungen und Veränderungen zu machen/vorzuschlagen
- keine zentrale Übersicht, man muss die Projekte selbst finden

Packages updaten

CRAN

```
update.packages()
```

Github

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

Bekannte Datensätze

Es gibt zahlreiche Beispieldatensätze, die mit R, bzw. Packages mitgeliefert werden:

Name in R	Beschreibung	Verfügbar 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

Name in R	Beschreibung	Verfügbar mit
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
faithful	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 Datensätze, die wir bereitstehen haben, finden wir mit `data()`

Wichtige Funktionen für Dataframes

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

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

Übung: Bekannte Datensätze ausprobieren

Nehmt euch etwas Zeit, ladet einen der Datensätze mit:

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

- Wie ist der Datensatz aufgebaut?
- Welche Variablen enthält er?
- Findet ihr bereits spannende Ergebnisse?

Daten einlesen

R kann viele verschiedene Datentypen einlesen.

Die gängigsten sind:

- TXT
- TSV
- 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 händisch einstellen können.

header: TRUE oder FALSE, falls die erste Zeile die Spaltentitel enthält

sep: Der Trenner, hier per default ein Komma

quote: Die Zeichen, die dafür sorgen, dass Sätze mit Kommas nicht getrennt werden.

dec: Das Zeichen, das für Dezimaltrennung verwendet wird

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

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

strip.white: Löscht Leerzeichen vor und nach Texten

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

fileEncoding: Setzt manuell das Encoding (oft UTF-8). Häufiges Problem beim Wechsel zwischen Win und Mac

Nutzung von read.csv()

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

Weitere read.table()-Abwandlungen

Für Text-Files

- `read.table()` flexible Überfunktion für 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 für Zeile ein und können genau steuern, wie die Zeile verarbeitet 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 große oder viele Dateien einlesen wollen.

Beispiele:

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

=====

Übrigens: 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

Standardmässig kann R keine Exceldateien lesen.

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

Wir sollten uns nur für 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 großen 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)
```

Verrücktere Formate

Eine kleine Zusammenstellung, welche Befehle und Packages bestimmte Dateiformate öffnen können

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 natürlich auch noch möglich ist: Verbindung zu Datenbanken oder Webscraping

Cheaterpackage **Data.table**: **fread()** rät 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 können 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 (1973–74 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 handeln.

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"
```

Übung: Dateien öffnen

Auf dem Github-Repo liegen drei originale Datensätze in CSV und XLSX.

Öffnet sie, und versucht die Spaltennamen und Spaltenformate richtig anzugeben. Oder wandelt die Dateien nach dem einlesen so um, dass ihr damit weiterarbeiten könnt.

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

Empfohlene Reihenfolge nach aufsteigender Schwierigkeit:

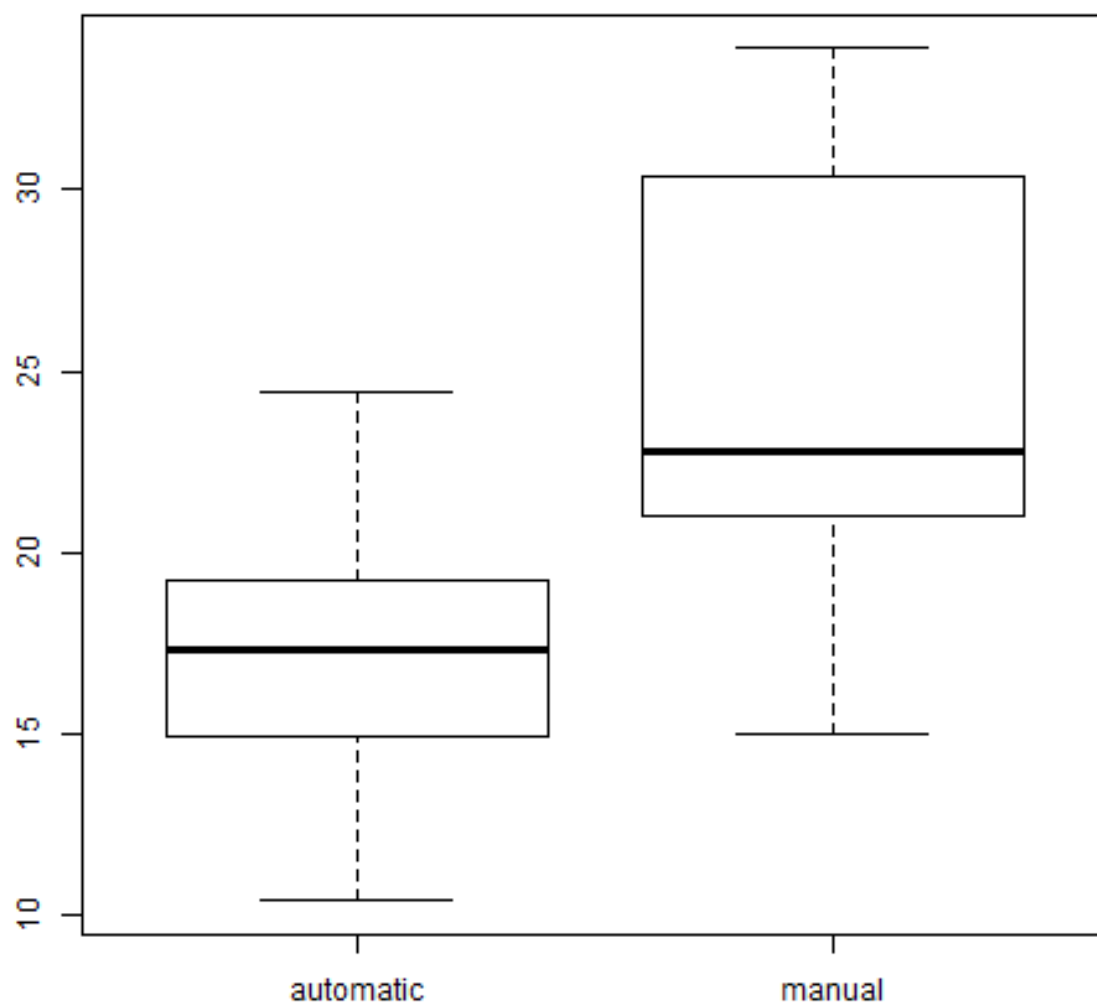


Figure 1: plot of chunk unnamed-chunk-41

Bierpreis, Fahrgastzählung, Wohnungen