DIHSÜD
Digital Innovation Hub

Der DIH SÜD wird unterstützt von:
FFG Forschung wirkt.
Das Land Steiermark
LAND KÄRNTEN

R in a Day – Einstieg in die Datenanalyse mit R

Digital Innovation Hub (DIH) Süd

Michael Melcher

FH JOANNEUM
Institut für Wirtschaftsinformatik und Data Science

6. Februar 2026

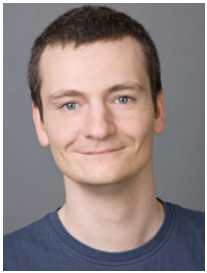
Geplante Inhalte für heute

0. Vorstellung, Kennenlernen der Teilnehmer
1. Einführung und Setup von **R**
2. Grundlagen in **R**
3. Daten importieren und verstehen
4. Daten bearbeiten, visualisieren und modellieren
5. Zusammenfassung, Ausblick und weitere Kurse im Rahmen des `DIH SÜD`

R in a Day ist ein praxisorientierter Einsteigerkurs, keine Programmierkenntnisse sind erforderlich, Fragen sind jederzeit erlaubt (bitte mich ganz unverschämt und lautstark unterbrechen).

Michael Melcher

- Studium Technische Chemie, Technische Mathematik (TU Wien)
- 2002 – 2021 Institut für Naturwissenschaften und Technologie in der Kunst, Akademie der Bildenden Künste, Wien
- 2012 – 2021: Institut für Statistik, Universität für Bodenkultur Wien
- seit 2021: Lehrender an der `IMC FH Krems`, Fach Mathematik
- seit 2021: Dozent (FH) an der `FH JOANNEUM`, Hauptstudiengänge *Data Science and Artificial Intelligence* (Master, berufsermöglichend), *Wirtschaftsinformatik* (Bachelor, Vollzeit)
- Tätigkeiten in Lehre und Forschung/Projekten
- Gebiete: Statistik, Statistical Learning, Chemometrie
- seit 2007: **R**-Enthusiast



Aber viel wichtiger: in welchem Bereich bewegen Sie sich, wofür möchten Sie **R** einsetzen (oder haben es bereits eingesetzt), was sind Ihre Erwartungen an den heutigen Kurs?

Inhalt

Ziele Modul 1

- Eine funktionierende Installation von **R** und `RStudio` am eigenen Rechner zu haben (oder zu wissen, wie man die Online-Version nutzt).
- Die Aufgabenteilung zwischen **R** und `RStudio` zu verstehen.
- Sich in `RStudio` zurechtzufinden.

R – Geschichte & Homepage

- The statistical programming language **S**: entwickelt in den $1970/80$er Jahren von John Chambers, Rick Becker und Allan Wilks in den *AT & T Bell Laboratories*.
- S-PLUS: kommerzielle Implementierung der Sprache **S** (TIBCO Software Inc.), teuer, nicht open-source.
- **R**: *free software environment for statistical computing and graphics*, Ross Ihaka & Robert Gentleman am *Department of Statistics* der University of Auckland/New Zealand, Anfang/Mitte der 1990er Jahre
- **Ziel**: statistische Programmiersprache technisch so gut wie **S**, aber frei, erweiterbar und gemeinschaftsgetrieben
- 1995: erste herunterladbare Version; 2000: **R** in der Version $1.0.0$ verfügbar
- Heute: R-core team (18 Personen mit Schreibrechten)
- **R** ist unter der GNU General Public Licence veröffentlicht (GPL-2 bzw. GPL-3). **R** darf uneingeschränkt (auch für kommerzielle Zwecke) verwendet werden, der Quellcode ist öffentlich. Wenn Code weitergegeben wird, muss er ebenso öffentlich sein (wenn er GPL-Code enthält).

R – Bedeutung

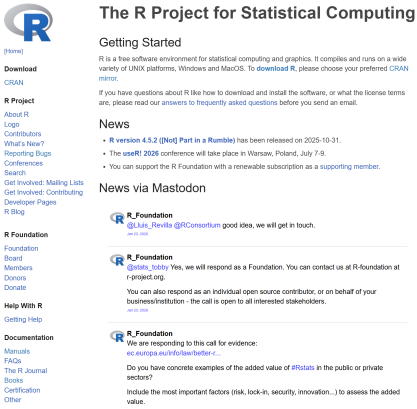
- **R** ist die führende Programmiersprache weltweit im Bereich Statistik.
- **R** ist nach `Python` in den Gebieten Data Science, Machine Learning, Artificial Intelligence unter den am häufigsten verwendeten Sprachen.
- **R** hat eine hohe Verbreitung im Sektor Bildung/Universitäten.
- Branchen/Unternehmen, die **R** verwenden:
  - **Tech-/Internetunternehmen**: Amazon, Walmart, Google, Meta, Microsoft, ...
  - **Finanzdienstleister und Banken**: JP Morgan, Accenture, ...
  - **Medien**: BBC, ...
  - **Datengetriebene Unternehmen/Plattformen**: Airbnb, Uber, Booking.com, Ebay, ...

R – Homepage

- **R**-project Homepage (einfach *r* oder *R* in google eingeben):

  http://www.r-project.org

  Besonders der Abschnitt Documentation/Manuals ist relevant – dort finden sich Dokumentationen geschrieben von u.a. Mitgliedern des **R**-core teams. Herauszustreichen ist *An Introduction to* **R**.



R installieren (eigener Rechner)

- Wir besuchen die **R** Homepage unter

  http://www.r-project.org/

- Linke Spalte oben, Link CRAN (Comprehensive R Archive Network), Wahl eines naheliegenden Servers (z.B. den der WU Wien)
- Im ersten Abschnitt Download and Install R wählen wir das entsprechende **Betriebssystem** (Linux, Windows or macOS).
- **Windows**: wir wählen das base subdirectory und dort den Link

  https://cran.r-project.org/bin/windows/base/R-4.5.2-win.exe

  (aktuellste Version mit Stand Jänner 2026). Ein Doppelklick auf das heruntergeladene .exe File, für alle weiteren Schritte können die Standardeinstellungen verwendet werden.
- Unter **macOS** wählen wir

  https://cran.r-project.org/bin/macosx/big-sur-arm64/base/R-4.5.2-arm64.pkg

  und nach einem Doppelklick können ebenso alle Standardeinstellungen beibehalten werden. Für macOS Versionen älter als 10.13 (High Sierra) findet sich ebenso auf dieser Seite das entsprechende Installationspaket.
- Auf **Linux** ist **R** in vielen Distributionen bereits vorinstalliert, ansonsten ist ein *Package Management System* ratsam, um **R** herunterzuladen und zu installieren.

R

Wir starten **R** und orientieren uns zuerst ...

RStudio (eigener Rechner)

- Statt der Eingabe einzelner Funktionen wollen wir längere Codefragmente/Skripte schreiben, wofür wir einen Editor verwenden werden. **R** liefert zwar einen mit, sein Funktionsumfang ist allerdings gering (z.B. kein *syntax-highlighting*, keine *command completion*, keine automatische Einrückung, ...).
- Es gibt viele Möglichkeiten (u.a. `vim`, `TinnR`, `Emacs`, ...), wir werden `RStudio` verwenden, das gratis unter

  `https://posit.co/`

  heruntergeladen werden kann. Wir gehen im Menüpunkt `Free & Open Source` auf `Download RStudio` und laden die `RStudio Desktop` Version für das entsprechende Betriebssystem herunter. Der direkte Link lautet

  `https://posit.co/download/rstudio-desktop/`

  Scrollt man etwas hinunter, erhält man Versionen für jedes gängige Betriebssystem. Die aktuelle Version lautet derzeit (Jänner 2026) 2026.01.0-392.

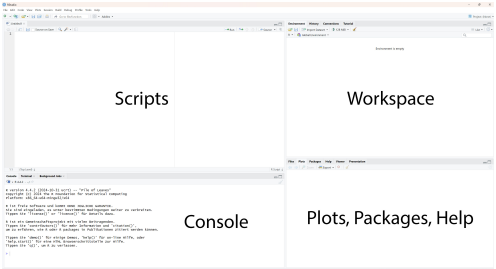
RStudio + R

Wir starten RStudio und orientieren uns auch hier zuerst ...

RStudio + **R**

- Die `RStudio` Oberfläche ist in 4 sogenannten panes (*Scheibe*) strukturiert (ggf. zu Beginn nur 3 sichtbar, mit `File` → `New File` → `R Script` bekommt man das vierte). Links unten haben wir die **R** Console, links oben das Skriptfenster, rechts oben den Workspace/die Command History und im rechten unteren pane werden Hilfeseiten, Graphiken und installierte Pakete angezeigt.

- Ein typischer Workflow in `RStudio` könnte so aussehen:
  1. Neues Skript mittels `File` → `New File` → `R Script`, in das wir unsere Abfolge von **R** Funktionen schreiben.
  2. Die Tastenkombination `Ctrl + Return` auf Windows/Linux bzw. `Command + Return` (Mac) führt die aktuelle Zeile bzw. mehrere markierte Zeilen aus. Der `Source`-button in der rechten oberen Ecke des Skriptfensters führt das gesamte Skript aus.
  3. Nach erledigter Arbeit speichern wir das Skript (Textdatei) mit der Erweiterung `.R` via `File` → `Save As`



RStudio + **R** – das Wichtigste

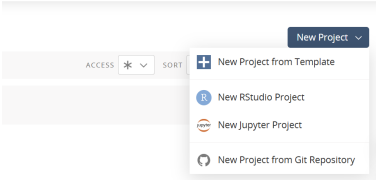
- **R** . . . die Sprache, die Engine, der Motor
- `RStudio` . . . ermöglicht uns ein bequemes Arbeiten mit **R**
- **R** ohne `RStudio` ✓ (möglich, aber anstrengend)
- `RStudio` ohne **R** ✗ (nicht möglich)
- Grundregel:
  - schnelle/kurze Rechnungen: in der **R** Console
  - alles andere (längere Skripte): im Skriptfenster, schrittweise ausführen, speichern (wesentlicher Vorteil: **Reproduzierbarkeit**)

R und RStudio in der Cloud

- Um **R** und RStudio online zu nutzen, registriert man sich in der Posit Cloud (rechts oben auf Sign Up) unter

  https://posit.cloud/

- Wir wählen den kostenlosen Plan (Cloud Free), klicken auf Learn more und hier auf Sign Up.
- Eingabe von Mail Adresse und Namen, Wahl eines Passworts. Nach einer Bestätigung per Mail (Link darin anklicken) kann mit sich rechts oben unter Log In einloggen.
- **Linke Spalte**: auf Your Workspace, dann rechts oben auf New Project → New RStudio Project



- Es erscheint die RStudio Oberfläche wie im Fall der lokalen Installation. Achtung: es bestehen Einschränkungen hinsichtlich des Speichers.

Inhalt

Ziele Modul 2

- Einfache Rechnungen (in der Console und im Skriptfenster) durchführen
- Erstellung und Manipulation von verschiedenen Objekten
- Anwenden einfacher Funktionen
- Datentypen in **R** (numerisch, logisch, character)
- Vektoren und Data Frames als wichtigste Datenstrukturen

Material & Vorbereitungen in `RStudio`

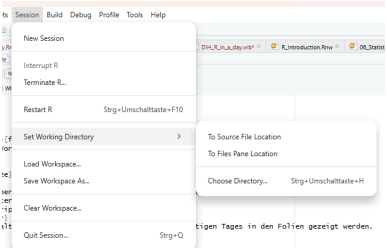
- Unter den per Mail ausgeschickten Materialien findet sich die Datei

  Skript_R_in_a_day.R

  Sie enthält alle Code-Fragmente, die im Laufe des heutigen Tages in den Folien gezeigt werden.
- Erstellen Sie am besten einen Ordner auf Ihrer SSD/Festplatte, kopieren das gezippte File dorthin und entpacken es.
- In `RStudio` wählen Sie

  File → *Open File*

  und wählen diese Datei aus.



- Obwohl einstweilen nicht unbedingt notwendig, wählen Sie in `RStudio` ferner

  Session → Set Working Directory → To Source File Location
- Wie bereits in Modul 1 besprochen, kann eine Code-Zeile durch die **Tastenkombination** `STRG + Return` oder `Cmd + Return` ausgefhrt werden. Der Cursor springt dabei immer auf die folgende Zeile.

Inhalt

R als Taschenrechner

Wir beginnen mit ein paar **einfachen Rechnungen** in der Console. In der Console werden die Rechnungen nach Eingabe und Drücken von `Enter` ausgewertet, in einem Skript `Ctrl + Return` (in macOS: `Cmd + Enter`) drücken, wenn der Cursor in auszuführender Zeile.

Addition + bzw.
Subtraktion −

```
1 + 2

[1] 3

3 - 5

[1] -2
```

Multiplikation * bzw.
Division /

```
3.5 * 5.5

[1] 19.25

3.5 / 5.5

[1] 0.6363636
```

Potenzen mit ^ oder **,
Klammersetzung stets mit runden Klammern ().

```
2^5

[1] 32

(1 - 3) * (3 + 2)^2

[1] -50
```

Mathematische R Funktionen

Funktion	Bedeutung	Beispiel
`sqrt(x)`	Quadratwurzel	`sqrt(16)`
`log(x)`	natürlicher Logarithmus	`log(5)`
`log(x, base = a)`	Logarithmus zur Basis a	`log(16, base = 4)`
`log10(x)`	Logarithmus zur Basis 10	`log10(5)`
`log2(x)`	Logarithmus zur Basis 2	`log2(5)`
`exp(x)`	Exponentialfunktion	`exp(5)`
`abs(x)`	Absolutwert	`abs(-10)`
`sin(x), cos(x), tan(x)`	Winkelfunktionen	`sin(`π`)`
`asin(x), acos(x), atan(x)`	inverse Winkelfunktionen	`asin(0.5)`
`factorial(x)`	Fakultät	`factorial(5)`

Beispiele:

```
# Argument im Bogenmass
sin(pi/2)
```

```
[1] 1
```

```
# Funktion + inverse Funktion
log(exp(3))
```

```
[1] 3
```

```
# ist 1 * 2 * 3 * 4 * 5
factorial(5)
```

```
[1] 120
```

Ein- und Ausgaben

- Das eingeblendete > ist das **command prompt**. Es signalisiert: **R** erwartet eine Eingabe . . .
- Der Dezimalpunkt in **R** ist stets (und unveränderbar) ein Punkt.
- Abstände spielen in **R** keine Rolle. Sie werden z.B. um Rechenzeichen wie $=$, $+$, $-$ etc. sowie nach Beistrichen empfohlen, um die **Lesbarkeit** zu erhöhen. In **R** sind folgende Eingaben erlaubt und führen zum selben Ergebnis.

```
3 * 4 + 2
[1] 14
```

```
3*4+2
[1] 14
```

- Das vorangestellte [1] gibt die Position des Elements rechts davon in einem (langen) Vektor an (hier bedeutungslos, siehe später).
- Bei unvollständigen Eingaben, wie beispielsweise

```
# ENTER bevor Zeile komplett
3 / (2 -
+ 4)

[1] -1.5
```

wo wir nach der ersten Zeile (versehentlich) bereits `Enter` gedrückt haben, bekommen wir die Anzeige

+

Wir können das statement durch Eingabe von $4)$ komplettieren oder wir brechen mit `Esc` die Eingabe ab und beginnen neu.

Ein- und Ausgaben

- Für sehr kleine oder sehr große Zahlen verwendet **R** die wissenschaftliche Notation.

```
# klein
3 / 6000000
```

```
[1] 5e-07
```

```
# gross
factorial(30)
```

```
[1] 2.652529e+32
```

  wobei 5e-07 für $5 \cdot 10^{-7}$ und 2.6e+32 für $2.6 \cdot 10^{32}$ stehen.

- Durch Drücken der `Pfeil nach oben` Taste können wir in der Historie der letzten Eingaben zurückblättern und eine davon ggf. wiederholen.

Zuweisungen

- Bisher haben wir volatile Ergebnisse erzeugt. Der Aufruf einer **R** Funktion erzeugt ein Ergebnis, das in der Console angezeigt wird (und dann nicht weiter verfügbar ist). Wir wollen daher unsere Ergebnisse bestimmten Objekten/Variablen zuweisen.

```
# speichern des ausgewerteten rechten Ausdrucks in Objekt mit Namen links
y <- exp(2)

# Alternative: "=" - Zeichen
y = exp(2)
```

- Der Zuweisungsoperator ist ein < (kleiner als) Zeichen und ein direkt nachfolgendes Minus -.
- Der Objektname kann dann für weitere Operationen verwendet werden bzw. der Inhalt des Objekts ausgegeben werden

```
# Inhalt von "y"
y

[1] 7.389056

# Zuweisung und Anzeige in einem Schritt
(y <- log(2))

[1] 0.6931472
```

- **R** ist case sensitive, d.h. die Objekte `y` und `Y` sind verschieden und können parallel existieren.
- Mehrere Anweisungen in einer Zeile, getrennt durch Semikola, sind möglich aber nicht empfehlenswert.

Datentypen

R kennt 6 Datentypen – 3 bis 4 davon sind für uns von Bedeutung

- Numerisch (Unterscheidung integer/float)

```
a <- 5
class(a)

[1] "numeric"

# integer (= ganze Zahl) "erzwingen"
b <- 5L
class(b)

[1] "integer"
```

- Logisch (`TRUE` oder `FALSE`)

```
e <- TRUE
class(e)

[1] "logical"
```

- Character

```
f <- "R in a day"
class(f)

[1] "character"
```

Häufiger Fehler:

```
# numerisch + character -> Fehler
3 + "7"

Error in 3 + "7": nicht-numerisches Argument für binären Operator

# R ist aber bemueht, ein Ergebnis zu erhalten ...
3 + TRUE

[1] 4
```

Inhalt

Vektoren

- Mittels der `c()`-Funktion können einzelne Zahlen/Objekte zu einem Vektor kombiniert werden. Vektorelemente können Namen tragen.

```
# Tagesumsaetze eines kleinen Cafes in Euro
umsaetze <- c(620, 790, 740, 550, 920, 1080, 1010)

# Ausgabe
umsaetze
```

```
[1]  620  790  740  550  920 1080 1010
```

```
# Umsaetze inkl. Namen
umsaetze_namen <- c(Montag = 620, Dienstag = 790, Mittwoch = 740,
                    Donnerstag = 550, Freitag = 920, Samstag = 1080,
                    Sonntag = 1010)

# Ausgabe
umsaetze_namen
```

```
   Montag   Dienstag   Mittwoch Donnerstag    Freitag    Samstag    Sonntag
      620        790        740        550        920       1080       1010
```

- Jeder Vektor hat eine Länge, entsprechend der Anzahl an Zahlen/Zeichen, die er enthält.

```
# Anzahl Elemente in Vektor
length(umsaetze)
```

```
[1] 7
```

- **Hintergrund** (Vektoren): Daten mit gleicher Bedeutung zu einem ganzen zusammenfassen. Es ist hilfreich, einen Vektor **nicht** als geometrisches Objekt zu sehen, sondern einfach als eine geordnete Sammlung von Zahlen.

Vektoroperationen

Operationen wie beispielsweise $+$, $-$, ... werden elementweise durchgeführt.

```
# Addition zweier Vektoren der Laenge 3
a <- c(1, 2, 3)
b <- c(4, 5, 6)
(sum_vec <- a + b)

[1] 5 7 9
```

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

```
# gleiche Vektoren a,b wie oben
(prod_vec <- a * b)

[1]  4 10 18
```

$$\mathbf{a} \cdot \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 \cdot 4 \\ 2 \cdot 5 \\ 3 \cdot 6 \end{pmatrix} = \begin{pmatrix} 4 \\ 10 \\ 18 \end{pmatrix}$$

Es erscheint einleuchtend, dass die beiden Vektoren für die Operationen $+$, $-$, ... dieselbe Länge haben sollten. **Was passiert, wenn sie nicht die gleiche Länge haben**? Probieren Sie es aus ...

Recycling

- Den eben beobachteten Vorgang nennt man recycling. Dabei wird der kürzere Vektor so oft wiederholt, bis seine Länge der des längeren Vektors entspricht. **R** macht daraus also folgendes

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} + \begin{pmatrix} 10 \\ 20 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} + \begin{pmatrix} 10 \\ 20 \\ 10 \\ 20 \\ 10 \end{pmatrix} = \begin{pmatrix} 11 \\ 22 \\ 13 \\ 24 \\ 15 \end{pmatrix}$$

- In dem Fall, wo die Länge des längeren Vektors ein ganzzahliges Vielfaches der Länge des kürzeren ist, nimmt **R** somit an, dass dieses Verhalten erwünscht ist, andernfalls werden wir zumindest gewarnt.

- Was lernen wir daraus? Erstens sollten wir sowohl Fehler- als auch Warnmeldungen lesen und zweitens recycling nur in Ausnahmefällen verwenden, also darauf achten, dass die Vektoren die korrekte Länge haben.

Vektorfunktionen

- Die Funktionen `sum()` bzw. `mean()` berechnen die Summe und das arithmetische Mittel (Stichprobenmittel) der Elemente eines Vektors. Diese sind wie folgt für einen Vektor $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ definiert:

$$\sum_{i=1}^{n} x_i = x_1 + \ldots + x_n \qquad \overline{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$$

```
# unsere Umsaetze
umsaetze
```

```
[1]  620  790  740  550  920 1080 1010
```

```
# Wochenumsatz (= Summe) und Mittelwert (durchschnittlicher Tagesumsatz)
sum(umsaetze)
```

```
[1] 5710
```

```
mean(umsaetze)
```

```
[1] 815.7143
```

Erstellung von Zahlenfolgen

Manchmal erweist es sich als günstig, **strukturierte Zahlenfolgen** zu erstellen. Dazu gibt es in **R** zahlreiche Möglichkeiten:

- Die Funktion `:` (ein Doppelpunkt)

```
# Abfolge von Integern
(y <- 1:10)
```

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

- Flexibler sind wir mittels der Funktion `seq()` (für *sequence*), die die Angabe eines **Inkrements** (Argument `by`) erlaubt:

```
# Inkrement im 'by' Argument
seq(from = 1, to = 4, by = 0.5)
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
# Spezifikation der Laenge des resultierenden Vektors
seq(from = 1, to = 4, length.out = 16)
```

```
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
```

- Die Abfolge der Vektorelemente kann mittels `rev()` **umgekehrt** werden:

```
# umgekehrte Reihenfolge
rev(y)
```

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

- **Wiederholung** bestimmter Muster mittels `rep()`

```
# Wiederhole den Eintrag '4' 3 mal
rep(x = 4, times = 3)
```

```
[1] 4 4 4
```

```
# etwas komplizierter
rep(x = 1:3, times = c(1, 4, 2))
```

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

Etwas Statistik ...

Einige wichtige (zumeist statistische) Funktionen. Die Argumente `x` bzw. `y` sind dabei immer Datenvektoren.

Funktion	Bedeutung
`mean(x)`	arithmetisches Mittel
`median(x)`	Median
`var(x)`	Varianz
`sd(x)`	Standardabweichung
`quantile(x, probs)`	Quantil
`IQR(x)`	Interquartilsabstand
`MAD(x)`	median absolute deviation
`min(x), max(x)`	Minimum, Maximum
`diff(x)`	Differenz $x_i - x_{i-1}$
`range(x)`	Spannweite
`cumsum(x), cumprod(x)`	kumulative Summe/Produkt
`cov(x, y)`	Kovarianz
`cor(x, y)`	Korrelation
`summary(x)`	(numerische) Zusammenfassung
`ceiling(x), floor(x)`	aufrunden, abrunden
`round(x, digits)`	runden
`which.min(x), which.max(x)`	Position des ersten Min/Max
`unique(x)`	Duplikate eliminieren

Vektortypen

- Vektoren in **R** haben einen `type` oder `mode`. Wie bereits erwähnt, kennt **R** 6 verschiedene Typen:

  `numeric`, `integer`, `complex`, `logical`, `character` und `raw`.

- **Achtung**: Alle Komponenten/Einträge eines Vektors in **R** müssen vom gleichen Typ sein (atomic structure). Gleiches gilt im übrigen für Matrizen und Arrays, jedoch **nicht** für Listen und data frames (siehe später).

integer:

```
# Integervektor
(x_n <- 1:10)

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

numeric:

```
# vektor mit reellen Zahlen
(real_vec <- c(1.5, 1.2, 3))

[1] 1.5 1.2 3.0
```

logical:

```
# logischer Vektor
(x_l <- c(TRUE, TRUE, FALSE, TRUE, FALSE))

[1]  TRUE  TRUE FALSE  TRUE FALSE
```

character:

```
# Charactervektor
(x_char <- c("Montag", "Dienstag", "Mittwoch", "Donnerstag",
             "Freitag", "Samstag", "Sonntag"))

[1] "Montag"     "Dienstag"   "Mittwoch"   "Donnerstag" "Freitag"
[6] "Samstag"    "Sonntag"
```

Warnung – Mischen von Vektortypen

- Mischen wir Einträge unterschiedlicher Typen in einem Vektor, versucht **R**, manche Einträge zu konvertieren (implicit type conversion = type coercion)

```
# Mischung character und numeric -> 2 und 6 werden characters!
(x <- c("Anna", 2, 6))
```

```
[1] "Anna" "2"    "6"
```

```
is.character(x)
```

```
[1] TRUE
```

```
# numeric/logical: TRUE -> 1; FALSE -> 0
(y <- c(TRUE, FALSE, 4))
```

```
[1] 1 0 4
```

```
# test
is.numeric(y)
```

```
[1] TRUE
```

- Mittels der Funktionen

`is.numeric()`, `is.integer()`, `is.character()` und `is.logical()`

kann der Typ abgefragt werden (Antwort: `TRUE` oder `FALSE`). Die Umwandlung (wenn möglich) erfolgt mittels

`as.numeric()`, `as.integer()`, `as.character()` und `as.logical()`

Logische Vektoren

Logische Vektoren entstehen beispielsweise, wenn (numerische) Vektoren mit anderen verglichen werden:

Funktion	Bedeutung	Beispiel
==	Gleichheit	`c(1, 2, 3) == 2`
!=	Ungleichheit	`c(1, 2, 3) != 2`
<	kleiner als	`c(1, 2, 3) < 2`
>	größer als	`c(1, 2, 3) > 2`
<=	kleiner oder gleich	`c(1, 2, 3) <= 2`
>=	größer oder gleich	`c(1, 2, 3) >= 2`

Beispiel:

```
# unser Datenvektor
x <- c(1, 2, 7, 5, 3, 8, 11, 0)

# Eintraege <= 3?
x <= 3

[1]  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

```
# Eintraege gleich 3?
x == 3

[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE

# wieviele Eintraege > 5 - was passiert hier?
sum(x > 5)

[1] 3
```

Logische Vektoren

Logische Vektoren können mittels logischer Operatoren miteinander verknüpft werden und ergeben wieder logische Vektoren.

logisches AND &

```
# ist TRUE, wenn beide TRUE sind
TRUE & TRUE

[1] TRUE

TRUE & FALSE

[1] FALSE
```

logisches OR |

```
# ist TRUE, wenn mindestens ein Argument TRUE ist
TRUE | FALSE

[1] TRUE

FALSE | FALSE

[1] FALSE
```

Die Operatoren AND & und OR | funktionieren auch mit Vektoren (Auswertung elementweise).

```
# unser Umsatz-Vektor
umsaetze

[1]  620  790  740  550  920 1080 1010

# wieviele Tage zwischen 600 und 900 Euro Umsatz?
sum((umsaetze >= 600) & (umsaetze <= 900))

[1] 3
```

Subsetting von Vektoren

Hintergrund: Oft sind wir nicht an allen Einträgen eines Vektors interessiert, sondern nur an einem Teil. Wie können wir also eine Teilemge (*subset*) aus dem Vektor extrahieren? In **R** haben wir dazu 4 Möglichkeiten, stets werden eckige Klammern verwendet:

- Wir geben die Position der interessierenden Elemente im Vektor

```
# unser Vektor - Umsaetze
umsatz <- c(Montag = 620, Dienstag = 790, Mittwoch = 740, Donnerstag = 550, Freitag = 920, Samstag = 1080, Sonntag = 1010)

# wir wollen das zweite Element (Umsatz am Dienstag)
umsatz[2]

Dienstag
     790

# wir wollen das zweite und letzte Element
umsatz[c(2, 7)]

Dienstag  Sonntag
     790     1010
```

- Mittels negativer Indices können wir bestimmte Elemente ausschließen. **Achtung**: kein Mischen positiver und negativer Indices!

```
# alle ausser dem ersten und letzten Element
umsatz[-c(1, length(umsatz))]

  Dienstag   Mittwoch Donnerstag    Freitag    Samstag
       790        740        550        920       1080
```

Subsetting von Vektoren

- Im Fall, dass die Elemente des Vektors Namen tragen, können die interessierenden Elemente mittels ihres Namens (ihrer Namen) extrahiert werden.

```
umsatz

    Montag   Dienstag   Mittwoch Donnerstag    Freitag    Samstag    Sonntag
       620        790         740        550        920       1080       1010

# Umsaetze am Mittwoch und Freitag
umsatz[c("Mittwoch", "Freitag")]

Mittwoch  Freitag
     740      920
```

- Mittels logischer Vektoren. `TRUE` bedeutet *Element behalten*, `FALSE` bedeutet *Element nicht behalten*.

```
# Elemente, die behalten werden sollen: TRUE (Mo, Do, Fr)
log_vec <- c(TRUE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE)

# verwende diesen logischen Vektor zum subsetting
umsatz[log_vec]

    Montag Donnerstag    Freitag
       620        550        920
```

Zumeist wird man den logischen Vektor nicht händisch eingeben, er wird vielmehr Ergebnis z.B. eines Vergleichs sein.

Übung

1. Extrahiere aus dem Vektor `umsatz` jene Umsätze, die zumindest 750 Euro betragen.
2. Ermittle den durchschnittlichen Umsatz an Wochentagen (Montag bis Freitag).
3. Ermittle den gesamten Wochenumsatz.

Lösung

- Extrahiere jene Umsätze, die zumindest 750 Euro betragen:

```
umsatz[umsatz >= 750]
```

```
Dienstag  Freitag  Samstag  Sonntag
     790      920     1080     1010
```

Was passiert hier im Detail? In eckigen Klammern wird ein logischer Vektor erzeugt:

```
# jeder Eintrag wird mit 750 verglichen -> jeweils TRUE oder FALSE
umsatz >= 750
```

```
    Montag   Dienstag   Mittwoch Donnerstag    Freitag    Samstag    Sonntag
     FALSE       TRUE      FALSE      FALSE       TRUE       TRUE       TRUE
```

Wird dieser logische Vektor nun zum subsetting herangezogen, werden vom Vektor `umsatz` nur jene Elemente behalten, wo im logischen Vektor ein `TRUE` zu finden ist, und das sind genau jene, wo der Umsatz zumindest 750 Euro betragen hat.

- Ermittle den durchschnittlichen Umsatz an Wochentagen (Montag bis Freitag):

```
# alternativ: mean(umsatz[-c(6, 7)])
mean(umsatz[1:5])
```

```
[1] 724
```

- Ermittle den gesamten Wochenumsatz

```
# Wochenumsatz
sum(umsatz)
```

```
[1] 5710
```

Fehlende Werte NA

- Fehlende Werte sind in **R** mittels NA (not available) codiert.

```
# fehlendes zweites Element des 5-Vektors x
x <- c(1, NA, 4, 6, 2)
```

- Die Resultate vieler Berechnungen sind dann wie erwartet. So kann beispielsweise die Summe $1 + \texttt{NA}$ nicht numerisch ausgewertet werden und man erhält als Ergebnis abermals einen fehlenden Wert NA.

```
# zweiter Vektor y
y <- 1:5
x + y
```

```
[1]  2 NA  7 10  7
```

- Das arithmetische Mittel eines Vektors, der NAs enthält, ist wieder NA, allerdings verfügt mean() wie auch viele andere Funktionen (cum(), sd(), var(), min(), ...) über ein Argument na.rm (remove NAs), das verwendet werden kann, um ein numerisches Ergebnis zu erhalten (die NAs werden weggelassen, auf die verbleibenden Werte wird die Funktion angewendet).

```
# Mittelwert
mean(x)
```

```
[1] NA
```

```
# aber
mean(x, na.rm = TRUE)
```

```
[1] 3.25
```

Fehlende Werte NA

- Will man auf das Vorhandensein fehlender Werte testen, verwendet man `is.na()`.

```
# Vektor der Laenge 6
(x <- c(NA, 1:5))
```

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

```
# fehlende Werte?
is.na(x)
```

```
[1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

- Will man zählen, wieviele fehlende Werte man in einem Vektor hat, verwendet man `sum()` in Verbindung mit `is.na()`. Erstere Funktion transformiert den durch `is.na()` generierten logischen Vektor in einen $0/1$-Vektor (1 bedeutet `TRUE`, also fehlender Wert). `sum()` zählt als die Anzahl an `TRUE`-Werten

```
# Anzahl fehlender Werte
sum(is.na(x))
```

```
[1] 1
```

- `NA`s können aus einem Vektor eliminiert werden.

```
# Behalte nur die nicht-fehlenden Werte; ! ist die Verneinung (TRUE -> FALSE; FALSE -> TRUE)
x[!is.na(x)]
```

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

Wichtige Funktionen Vektoren

Funktion	Bedeutung	Beispiel
`c()`	Erstellung eines Vektors	`c(1, 2, 3)`
`rep()`	Wiederholung	`rep(c(5, 7), c(2, 3))`
`seq()`	Erzeugt Folge	`seq(1, 5, 0.2)`
`sample()`	Zufallsstichprobe	`sample(1:10, 3)`
`names()`	extrahiere Namen	`names(c(A = 3, B = 5))`
`length()`	Länge des Vektors	`length(3:5)`
`which()`	Position mit Bedingung = `TRUE`	`which(c(1, 5, 7) < 6)`
`any()`	mind. 1 Element erfüllt Bedingung?	`any(1:5 < -1)`
`all()`	alle Elemente erfüllen Bedingung?	`all(1:5 > -1)`
`sort()`	Sortierung	`sort(c(5, 2, 9))`

Inhalt

Hilfe für/in **R**

- Im Gegensatz zu menübasierten Programmen (`Excel`, `SPSS`, ...) ist es in **R** notwendig zu wissen ...
  1. wie die **Funktion heißt**, die das tut, was man braucht (*Funktionsnamen*) und in welchem Paket sie zu finden ist und
  2. wie man die **Funktion anwendet** (*was braucht sie als Input, was gibt sie als Output?*)
- Antwort auf Frage 1: zumeist `Google` (heute eher: `ChatGPT`), z.B. mittels Suchbegriff

  R Mittelwert berechnen

  oder

  R calculate mean
- Antwort auf Frage 2: wie Frage 1 oder mittels

```r
# Hilfe fuer die Funktion namens "mean";
help(mean)

# gleicher Effekt:
?mean
```

Hilfe zu **R** Funktionen

- Sehen wir uns zunächst den einfacheren Fall 2 an – der Funktionsname ist bekannt, aber wie die Funktion verwendet werden soll, ist unklar.
- Jede Funktion (aber auch jeder Datensatz) in **R** hat eine Hilfeseite, die mittels der Funktion `help()` aufgerufen werden kann.

```
# Hilfe fuer die mean() - Funktion; Alternative: ?mean
help(mean)
```

- Alle Hilfeseiten haben eine klar definierte Struktur:
  - **Description**: kurze Beschreibung, was die Funktion tut
  - **Usage**: wie die Funktion verwendet wird
  - **Arguments**: obligatorische und optionale Argumente
  - **Details**: weitere Informationen/Details
  - **Value**: Rückgabewert der Funktion
  - **Examples**: eines oder mehrere Beispiele, die mittels copy & paste in die Console kopiert und ausgeführt werden können. Alternative: Eingabe von `example(mean)` in der Console.
- **Achtung**: so wie die Funktion selbst ist die Hilfeseite auch nur dann verfügbar, wenn das entsprechende Paket installiert ist, in dem sich die Funktion befindet. Rund 10 Pakete sind in der Basisinstallation enthalten und brauchen nicht extra geladen werden.

Hilfe für/in **R** – Paketinstallation

Beispiel (geometrisches Mittel):

- Wir wissen (oder haben gehört ...), dass das **arithmetische Mittel** nicht immer für die Ermittlung eines *typischen Werts* sinnvoll ist. Hat man es beispielsweise mit Wachstumsraten zu tun, ist das sogenannte **geometrische Mittel** die geeignete Größe.

- Formel für Beobachtungen $x_1, x_2, \ldots, x_n$:

$$\overline{x}_{\text{geo}} = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n}$$

- Eine Google-Suche oder der Aufruf von

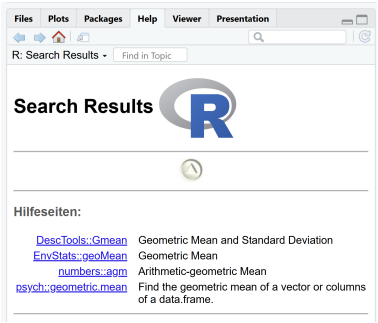
```
??geometricmean
```

liefert als Erkenntnis, dass es eine solche Funktion wohl in mehreren **R** Paketen gibt.

- Die Schreibweise

```
xx:yy
```

bedeutet, dass in einem Paket namens `xx` die Funktion `yy` vorhanden ist.



Installation von **R** Paketen (eigener Rechner)

- Unter einem Package versteht man in **R** eine kommentierte Sammlung von Funktionen und/oder Daten.
- Die Basisinstallation von **R** enthält bereits einige Pakete mit diversen Funktionen, z.B. das `stats` package, das `base` package, ...
  Früher oder später wird man Funktionen von fremden Paketen verwenden wollen. Aktuell (Stand Jänner 2026) gibt es 23.125 **R** Pakete alleine auf `CRAN` (+ Bioconductor, Github, ...). Dies bedeutet einen Zuwachs im Vergleich zu Mai 2025 von ca. 700 Paketen.
- Zwei Schritte sind notwendig, um Funktionen (oder Daten) eines Pakets verwenden zu können:
  1. Installation des Pakets (einmalig, dann ist es auf Ihrem Rechner)
  2. Laden des Pakets (in jeder **R** Session, wo das Paket verwendet werden soll)
- **Installation** (Möglichkeit 1 über die **R** Console): Eingeben der beiden Befehle in die Console (beispielhaft hier für unser Paket `DescTools`):

```r
# installiere Paket aus dem Netz - Internetverbindung erforderlich
install.packages("DescTools", dependencies = TRUE)
```

Installation von **R** Paketen in `RStudio` (eigener Rechner)

- **Installation** (Möglichkeit 2 über das Menü in `RStudio`): Als Vorbereitung wählen wir im `RStudio` Menü

  `Tools` → `Global Options`

  und in der linken Spalte `Packages` und gehen auf `Change`. Wir suchen einen nahen Server, z.B. `Austria - Wirtschaftsuniversitaet Wien`) und bestätigen 2 mal mit `Ok`.

- Wir wählen

  `Tools` → `Install Packages`

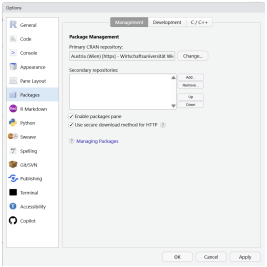
  aus dem `RStudio` Menü.

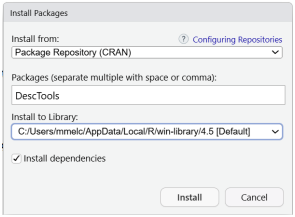
- Wir geben den Paketnamen ein, hier `DescTools` (case sensitive). Wollen wir mehr als ein Paket installieren, trennt man die Namen durch Strichpunkte oder Spaces.

- Optional kann der Pfad geändert werden, in den das Paket installiert werden soll.

- `Install dependencies`: ob auch Pakete installiert werden sollen, die für das gesuchte Paket notwendig sind (ja, das wollen wir).

- `Install` – ein paar Sekunden warten – nun ist das Paket installiert.





Installation von **R** Paketen

- Nun wollen wir das **Paket laden**, um es in der aktuellen **R** Session verwenden zu können.

```
# Laden des Pakets; Alternative: library(DescTools)
require(DescTools)
```

und wir haben nun Zugang zu allen Funktionen des Pakets `DescTools`, darunter auch `Gmean()`.

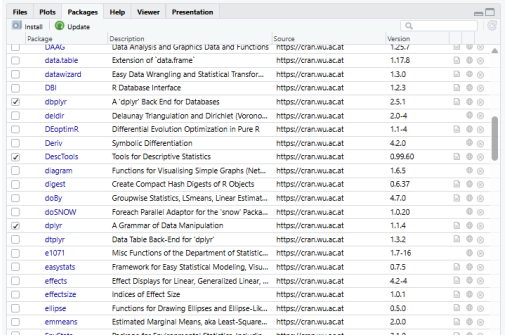
```
# Verwendung
Gmean(c(1.02, 1.04, 1.04))
```

```
[1] 1.03329
```

- Interpretation des Ergebnisses (beispielhaft): hat man in 3 aufeinanderfolgenden Jahren eine Verzinsung von 2, 4 und wieder $4\,\%$ pro Jahr für ein Guthaben auf der Bank, so entspricht dies einem **durchschnittlichen** Zinssatz von $3.33\,\%$.

Installation von **R** Paketen

- In `RStudio` kann man leicht überblicken, welche Pakete installiert sind und welche Pakete geladen sind.
- Im rechten unteren Pane auf den Reiter `Packages` wechseln.
- In der **User Library** sieht man alle installierten Pakete, angehakte Pakete sind darüberhinaus aktuell geladen.
- Durch Anhaken kann ein Paket auch geladen werden (warum ist dies eher nicht zu empfehlen?).



Installation von **R** Paketen (Posit Cloud)

- Die Installation eines **R** Pakets in der Posit Cloud läuft analog.
- Im RStudio Projekt gehen wir im **rechten unteren Fenster** auf den Reiter

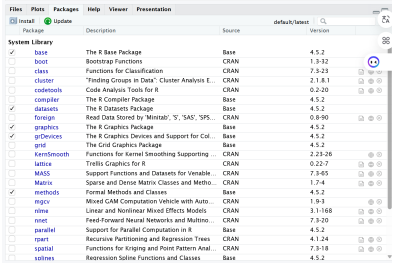
  Packages

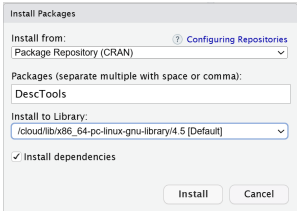
  und wählen dann gleich darunter install. Es erscheint das gleiche Fenster wie zuvor.
- Bevor wir das Paket verwenden wollen, tippen wir

```
require(DescTools)
```

oder

```
library(DescTools)
```

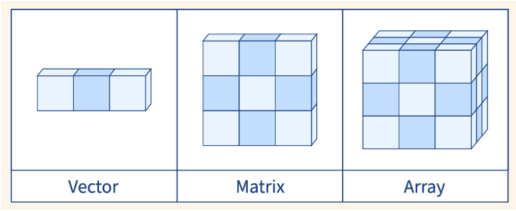




Inhalt

Matrizen und Arrays

- Eine Matrix ist eine rechteckige Anordnung von Daten (zumeist Zahlen).
- Arrays sind Verallgemeinerungen von Matrizen (z.B. Patient $\times$ Blutparameter $\times$ Zeitpunkt).
- Obwohl in der Numerik sehr wichtig, spielen Matrizen/Arrays in **R** eine untergeordnete Rolle. Grund: sie können nur Daten eines Typs enthalten.



Beispiele:

Die Erstellung erfolgt in **R** mittels der `matrix()` bzw. `array()` Funktion.

```
# Matrix mit Zahlen 1 bis 6, 3 Zeile, 2 Spalten
(mat <- matrix(1:6, nrow = 3, ncol = 2))
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

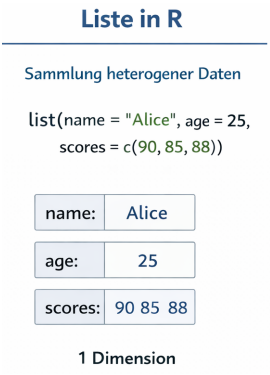
```
# Array der Dimension 3 x 2 x 2
(ar <- array(1:8, dim = c(2, 2, 2)))
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Listen

- **Motivation**: Oft will man Objekte unterschiedlichen Typs in einem größeren Objekt speichern (wie in eine Container will man mehrere, oft völlig unterschiedliche Dinge zusammenwerfen).
- In **R** heissen solche Objekte Listen und können mit der `list()` Funktion erstellt werden.
- **Beispiel**:

```
# erstelle EINE Liste mit 3 unterschiedlichen Objekten
(beispiel_liste <- list(Name = "Alice",
                        Alter = 25,
                        Testergebnisse = c(90, 85, 88)))

$Name
[1] "Alice"

$Alter
[1] 25

$Testergebnisse
[1] 90 85 88
```



Bedeutung: **R** Funktionen können lediglich ein Objekt zurückgeben. Will man mehrere Rückgabeobjekte, fasst man sie zu einer Liste zusammen (dies ist dann 1 Objekt) . . .

Listen

- Listen können sehr umfangreich sein – auch deshalb, weil eine Liste als ein Element wiederum eine Liste enthalten kann (rekursive Struktur). Einen schnellcne Überblick erhält man mittels `str()`

```
str(beispiel_liste)
```

```
List of 3
 $ Name          : chr "Alice"
 $ Alter         : num 25
 $ Testergebnisse: num [1:3] 90 85 88
```

- Der Zugriff auf eine Liste erfolgt mittels doppelter eckiger Klammern (wenn **ein** Listenelement von Interesse ist) oder einfacher Klammern (wenn man eine Teil-Liste erhalten möchte):

```
# nur das erste Listenelement -> das Ergebnis ist (hier) ein Vektor der Laenge 1
beispiel_liste[[1]]
```

```
[1] "Alice"
```

```
# Listenelemente 2 und 3 -> das Ergebnis ist wiederum eine Liste
beispiel_liste[c(2, 3)]
```

```
$Alter
[1] 25

$Testergebnisse
[1] 90 85 88
```

Inhalt

Data Frames vs. Matrizen

- Oft will man Variablen verschiedenen Typs (z.B. `numeric` und `character`) in ein matrixähnliches Objekt stecken. Das Problem: alle Spalten einer Matrix müssen vom gleichen Typ sein.
- In einem sogenannten `data frame` sind Spalten (Variablen) unterschiedlichen Typs gestattet. Zur Erstellung kleiner Datensätze verwendet man die Funktion `data.frame()`:

```
# Erstellung eines data frames
df <- data.frame(Groesse = c(1.72, 1.65, 1.85),
                 Gewicht = c(70, 59, 80),
                 Geschlecht = c("m", "w", "m"),
                 row.names = c("Stefan", "Anna", "Fritz")) # rownames optional

# Ausgabe
df
```

```
       Groesse Gewicht Geschlecht
Stefan    1.72      70          m
Anna      1.65      59          w
Fritz     1.85      80          m
```

Data Frame in R

2D Tabelle heterogener Daten

data.frame(Name = c("Alice","Bob"), Age = c(25, 32), Score = c(90, 88)

Name	Age	Score
Alice	25	90
Bob	32	88

2 Dimensionen

Data frames (oder die etwas modernere Variante, sogenannte *tibbles*) sind **der** Datentyp für **Datensätze** in **R**.

Data Frames – Subsetting

- Einzelne Elemente, ganze Zeilen oder Spalten erhält man – wie schon bei Vektoren – **mit eckigen Klammern** `[]`. **Achtung**: data frames sind 2D Strukturen, daher müssen 2 Indices – getrennt durch ein Komma – angegeben werden.

```
# 2. und 3. Zeile, keine Einschraenkung bei den Spalten
df[2:3, ]
```

```
      Groesse Gewicht Geschlecht
Anna     1.65      59          w
Fritz    1.85      80          m
```

- Will man ganze Spalten extrahieren, kann auch der $ Operator verwendet werden:

```
# Groessen-Spalte
df$Groesse
```

```
[1] 1.72 1.65 1.85
```

```
# Alternative mit gleichem Ergebnis:
df[, "Groesse"]
```

```
[1] 1.72 1.65 1.85
```

- Ebenso können logische Vektoren zum subsetting verwendet werden.

```
# alle Zeilen mit Geschlecht = weiblich
df[df$Geschlecht == "w", ]
```

```
     Groesse Gewicht Geschlecht
Anna    1.65      59          w
```

Data Frame Funktionen

In den untenstehenden Beispielen ist `df` ein data frame, `x` eine Vektor mit einer Länge gleich `nrow(df)`, `y` ein Vektor mit einer Länge gleich `ncol(df)` und `fun` eine Funktion.

Funktion	Bedeutung
`dim(df)`	Dimension des data frames
`nrow(df), ncol(df)`	Anzahl Zeilen/Spalten in data frame
`summary(df)`	spaltenweise Zusammenfassung
`subset(df, ...)`	wähle Teilmenge aus `df` aus
`na.omit(df)`	entfernt Zeilen mit `NA`s
`head(df), tail(df)`	zeige erste (letzte) Zeilen des data frames
`cbind(df, x), rbind(df, y)`	spalten- oder zeilenweises erweitern
`colMeans(df), rowMeans(df)`	Spalten- und Zeilenmittel
`scale(df)`	spaltenweises Skalieren
`cov(df), cor(df)`	Kovarianz- und Korrelationsmatrix
`apply(df, 2, fun)`	Funktion `fun` auf Spalten angewendet
`apply(df, 1, fun)`	Funktion `fun` auf Zeilen angewendet
`lapply(df, fun), sapply(df, fun)`	Funktion auf Spalten angewendet

In Modul 4 werden wir zahlreiche dieser (und weiterer Funktionen) in Aktion sehen ...

Inhalt

Inhalt

Einleitung/Arbeitsverzeichnis

- In **R** ist es – wie auch in anderen Anwendungen – üblich, für ein neues Projekt (groß oder klein), ein neues lokales Verzeichnis (auf seiner SSD/Festplatte) anzulegen, besonders dann, wenn auf viele Daten (Abbildungen, Ergebnisse, . . .) zugegriffen wird oder solche erzeugt werden.
- Dieses neu angelegte Verzeichnis definiert man am Beginn seiner Arbeit als working directory (Arbeitsverzeichnis) mittels

```
# Setzen des Arbeitsverzeichnisses -- ACHTUNG: muss angepasst werden
setwd("E:/Projekte/Projekt_XYZ")
```

- Das aktuelle Arbeitsverzeichnis kann abgefragt werden

```
# aktuelles Arbeitsverzeichnis?
getwd()

[1] "E:/FH_Joanneum/DIH_neu/R_in_a_Day"
```

- Jeglicher Austausch (Laden, Speichern) von Dateien erfolgt nun in diesem Verzeichnis (oder Unterverzeichnissen davon).

Inhalt

R Objekte

- Im Laufe einer **R** Session erzeugen und verwenden wir verschiedenste Objekte, zumeist Funktionen und Datensätze. Da nach Beendigen einer **R** Session alle diese verloren wären, wollen wir sie (manchmal) **speichern**, um sie in einer zukünftigen **R** Session zu laden. Beispielsweise haben wir zuvor den kleinen data frame `df` erzeugt.

```
# unser data frame
df
```

```
       Groesse Gewicht Geschlecht
Stefan    1.72      70          m
Anna      1.65      59          w
Fritz     1.85      80          m
```

- **Anmerkung**: Wir können mittels der Funktionen `ls()` alle Objekte der laufenden **R** Session auflisten und ggf. einzelne davon mit `rm()` löschen.

```
# Speicherinhalt
ls()
```

```
 [1] "a"              "ar"             "b"              "beispiel_liste" "df"             "e"              "f"              "log_vec"
 [9] "mat"            "prod_vec"       "real_vec"       "sum_vec"        "umsaetze"       "umsaetze_namen" "umsatz"         "x"
[17] "x_char"         "x_l"            "x_n"            "y"
```

```
# entferne beispiel_liste
rm(beispiel_liste)
```

R Objekte

- **R** kennt zwei unterschiedliche Dateiformate, um Daten zu speichern: `.RData` und `.RDS`. In einem `.RDS` file kann nur ein einzelnes Objekt (z.B. ein data frame) gespeichert werden, in einem `.RData` file auch mehrere Objekte. Wir werden uns auf `.RData` Objekte konzentrieren.
- Wir benötigen die Funktion `save()`, um `.RData` Objekte im Arbeitsverzeichnis zu speichern und `load()`, um sie ggf. von dort wieder zu laden (z.B. in einer neuen **R** Session).

```
# speichern des data frames
save(df, file = "Personen.RData")
```

```
# laden des data frames (z.B. in einer zukuenftigen R Session)
load(file = "Personen.RData")
```

- Man beachte, dass der **R** Objektname (`df`) und der Dateiname (`Personen`) beim Speichervorgang nicht übereinstimmen müssen. Wird ein `.RData` file wieder geladen, hat es in der **R** Session dann aber wieder den ursprünglichen Namen (`df`).
- Will man den gesamten Speicherinhalt (*workspace*) speichern, verwendet man `save.image()`. Mittels `load()` kann in einer neuen **R** session dieser Workspace wieder geladen werden.

```
save.image("image_2026_01_30.RData")
```

Auch bei der Beendigung einer **R** Session wird man gefragt, ob man den Workspace speichern will . . .

Inhalt

Text Dateien

- Die Funktion `read.table()` importiert eine Tabelle/Matrix im `ASCII Format`. Es ist zu empfehlen, Daten aus `Excel` (oder einer anderen Anwendung) als Textdatei (`.csv` oder `.txt`) zu speichern (Name im `Excel`: `.csv Trennzeichen getrennt`) und diese dann als solche in **R** zu laden.
- Es ist ratsam, sich die Datei in einem externen Texteditor (z.b. `Notepad++`) anzusehen: hat sie einen header (Kopfzeile mit Spaltenüberschriften), welches Zeichen wird als Dezimaltrennzeichen verwendet, welches Zeichen wird als Separator zwischen den Zellen verwendet und wie sind fehlende Werte codiert?
- Wir lesen die Datei `Auto.csv` ein, die einen header hat, und ein Komma als Trennzeichen verwendet.

```
# Auto Datensatz aus .csv Datei
data_Auto <- read.table(file = "Auto.csv", header = TRUE, sep = ",")

# Ansehen der ersten Zeilen
head(data_Auto)
```

```
  mpg cylinders displacement horsepower weight acceleration year origin                  name
1  18         8          307        130   3504         12.0   70      1 chevrolet chevelle malibu
2  15         8          350        165   3693         11.5   70      1         buick skylark 320
3  18         8          318        150   3436         11.0   70      1        plymouth satellite
4  16         8          304        150   3433         12.0   70      1             amc rebel sst
5  17         8          302        140   3449         10.5   70      1               ford torino
6  15         8          429        198   4341         10.0   70      1          ford galaxie 500
```

```
# Zeilen- und Spaltenanzahl
dim(data_Auto)
```

```
[1] 392   9
```

Import und Export von Daten

Wichtige `read.table()` Argumente

- `file` ... Dateiname (ggf. inklusive Pfad) in Anführungszeichen
- `header` ... `TRUE` oder `FALSE` – haben die Daten eine Kopfzeile mit Spaltenüberschriften?
- `sep` ... Trennzeichen. Default Wert ist *whitespace* (space oder tab).
- `dec` ... Zeichen, das als Dezimaltrennzeichen verwendet wird (bei Daten, die aus deutschsprachigen `Excel`-Sheets stammen, oft ein Beistrich)
- `na.strings` ... Coding von fehlenden Werten im Datensatz (default ist `NA`).
- `skip` ... Anzahl der Zeilen, die am Beginn des files übersprungen werden sollen.

Im `Environment` pane in **RStudio** (rechts oben) kann der Datenimport mittels `Import Dataset` auch menügestützt durchgeführt werden. Analog können wir mit der Funktion `write.table()` den Inhalt eines **R** Objekts (zumeist ein data frame) als Textdatei speichern.

Text Dateien – wie soll es **nicht** sein?

Machen wir einen Fehler in der Spezifikation der Argumente im `read.table()` Aufruf, so merkt man dies (meistens) beim Ansehen der Daten.

```
# anstatt sep = "," wird sep = ";" gewaehlt
data_Auto_false <- read.table(file = "Auto.csv", header = TRUE, sep = ";")

# Ansehen
head(data_Auto_false, n = 3)
```

```
  mpg.cylinders.displacement.horsepower.weight.acceleration.year.origin.name
1                       18,8,307,130,3504,12,70,1,chevrolet chevelle malibu
2                             15,8,350,165,3693,11.5,70,1,buick skylark 320
3                              18,8,318,150,3436,11,70,1,plymouth satellite
```

oder

```
# anstatt header = TRUE wird header = FALSE gewaehlt
data_Auto_false2 <- read.table(file = "Auto.csv", header = FALSE, sep = ",")

# Ansehen
head(data_Auto_false2, n = 3)
```

```
   V1        V2           V3         V4     V5           V6   V7     V8                        V9
1 mpg cylinders displacement horsepower weight acceleration year origin                      name
2  18         8          307        130   3504           12   70      1 chevrolet chevelle malibu
3  15         8          350        165   3693         11.5   70      1         buick skylark 320
```

Alternativen zu `read.table()`

- Im Normalfall importiert `read.table()` eine Textdatei in akzeptabler Zeit.
- **Beispiel**: Datei `FTIR.csv` mit knapp 100 MB (3600 Zeilen, 1427 Spalten, entsprechend 5.1 Millionen Zahlen) in rund 5 Sekunden.

```
# read.table (utils/base R)
system.time(readtable_time <- read.table("FTIR.csv", header = TRUE,
                                         row.names = NULL, sep = ";",
                                         stringsAsFactors = FALSE))

      User      System verstrichen
      5.17        0.11        5.28
```

- Alternative Funktionen (`read_delim()` im `readr` Package bzw. `fread()` im `data.table` Package) benötigen etwas bzw. deutlich kürzer.

```
# read_delim (readr)
system.time(readcsv_time <- readr::read_delim(file = "FTIR.csv", delim = ";"))

      User      System verstrichen
      4.17        2.03        3.22

# fread (data.table)
system.time(fread_time <- data.table::fread("FTIR.csv"))

      User      System verstrichen
      0.42        0.02        0.33
```

Inhalt

The readxl package

- Das readxl Package erlaubt den Import von Dateien im Excel Format (.xls und .xlsx). Die Schwierigkeit besteht darin, dass ein Excel file aus mehr als einem sheet bestehen kann.
- Die 2 wichtigsten Funktionen sind excel_sheets() und read_excel().

```
# Laden des Pakets (zuerst installieren, wenn notwendig)
require(readxl)
```

Wir importieren die Datei Population_Austria.xlsx mit ein paar Daten zur Einwohnerzahl der österreichischen Landeshauptstädte.

- Die Funktion excel_sheets() zeigt uns die Namen der sheets, die die Datei enthält.

```
# Liste Namen der Sheets auf
excel_sheets("Population_Austria.xlsx")
```

```
[1] "Year_1981" "Year_1991" "Year_2001" "Year_2011" "Year_2018"
```

The readxl package

- Jetzt können wir ein bestimmtes sheet mittels `read_excel()` importieren.

```
# read_excel mit Datei- und sheet-Name
read_excel("Population_Austria.xlsx", sheet = "Year_1981")

# A tibble: 9 x 3
  Bundesland        Hauptstadt    Bevoelkerung
  <chr>             <chr>                <dbl>
1 Burgenland        Eisenstadt          269771
2 Kaernten          Klagenfurt          536179
3 Niederoesterreich Sankt Poelten      1427849
4 Oberoesterreich   Linz               1269540
5 Salzburg          Salzburg            442301
6 Steiermark        Graz               1186525
7 Tirol             Innsbruck           586139
8 Vorarlberg        Bregenz             305164
9 Wien              Wien               1531346
```

The readxl package

- Gibt es mehrere sheets in einem Excel file und **alle** sollen importiert werden, kann das mit folgendem Code geschehen. Die einzelnen sheets werden als Elemente in einer Liste gespeichert.

```
# importiere alle sheets auf einmal
excel_file <- lapply(excel_sheets("Population_Austria.xlsx"),
                     read_xlsx,
                     path = "Population_Austria.xlsx")
str(excel_file, max.level = 1)
```

```
List of 5
 $ : tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
 $ : tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
 $ : tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
 $ : tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
 $ : tibble [9 x 3] (S3: tbl_df/tbl/data.frame)
```

Erläuterung: Die Funktion `lapply()` wendet auf jedes sheet (`Year_1981`, `Year_1991`, ..., `Year_2018`) die Funktion `read_xlsx()` an und speichert die Tabelle als ein Listenelement.

Inhalt

R Zugriff auf Datenbanken

- In den meisten Fällen erfolgt der Datenimport in **R** aus **Textdateien** oder aus `Excel`-sheets, insbesondere dann, wenn die Daten *klein* sind, d.h. (problemlos) in den RAM passen.
- Oft liegen in Unternehmen die (großen) Daten(mengen) auf **zentralen Servern** (Datenbankserver). Verwaltet wird die Datenbank von einem **Datenbankmanagementsystem, DBMS** wie `PostgreSQL`, `MySQL`, `Oracle Database` oder `Microsoft SQL Server`.
- Wir als client wollen **R** dazu verwenden, auf solche Daten über Abfragen (*query*) zuzugreifen – wir bekommen also nur die benötigten Daten zurück. In **R** formulieren wir Abfragen, `SQL` läuft auf der Datenbank und wir erhalten die Ergebnisse.

R Zugriff auf Datenbanken

Der Zugriff erfolgt in 3 Schritten

- Laden des **R** Pakets `DBI` (Standardschnittstelle für Datenbanken in **R**) – erlaubt die Kommunikation zwischen **R** und (fast) allen Datenbanksystemen.

```
require(DBI)
```

- Der eigentliche Zugriff erfolgt über den passenden Treiber bzw. das passende **R**-Paket, den/das es für (so gut wie) jede Datenbank gibt.

Datenbank	Empfohlenes Paket/Treiber
PostgreSQL	`RPostgres`
MySQL	`RMariaDB`
MariaDB	`RMariaDB`
SQLite	`RSQLite`
Microsoft SQL Server	`odbc`
Oracle Database	`odbc`
Google Big Query	`bigrquery`

Lässt sich kein passender Treiber finden, funktioniert in der Regel `odbc`.

R Zugriff auf Datenbanken

Der Treiber erlaubt nun die Kommunikation von **R** mit dem DBMS. Abhängig vom DBMS wären folgende Aufrufe realistisch:

```
con <- DBI::dbConnect(
  RMariaDB::MariaDB(),
  username = "foo"
)
```

oder

```
con <- DBI::dbConnect(
  RPostgres::Postgres(),
  hostname = "databases.mycompany.com",
  port = 1234
)
```

Wir erstellen also eine **Verbindung zur Datenbank**. Das erste Argument der Funktion `dbConnect()` aus dem `DBI` pakage ist stets der Treiber (jeweiliges Paket (siehe vorige Tabelle) muß installiert sein), alle folgenden Argumente sind credentials (Username, Passwort etc.).

R Zugriff auf Datenbanken

Die Datenabfrage erfolgt mittels der `dbGetQuery()` Funktion (ebenfalls aus dem `DBI` Paket):

```
data <- dbGetQuery(con,
  "SELECT * FROM customers LIMIT 100"
)
```

Im vorliegenden Statement würden nun aus der Tabelle `customers` alle Spalten geholt werden, maximal jedoch 100 Zeilen.

R – MVE Datenbanken

Als Beispiel erstellen wir uns zuerst eine SQLite Datenbank (was wir normalerweise nicht tun . . .)

1. Wir laden die notwendigen Pakete:

```
# Pakete muessen installiert sein
require(DBI)
require(RSQLite)
```

```
Lade nötiges Paket:  RSQLite
```

2. Wir verbinden uns zur (noch leeren) Datenbank:

```
# Verbindung
con <- dbConnect(SQLite(), "demo_shop.sqlite")
```

3. Wir erstellen eine (ultrakleine) Datenbank mit 3 Tabellen – `customers`, `products` und `orders` – zuerst als data frames in **R**:

```
# Kunden
customers <- data.frame(
  customer_id = 1:5,
  name = c("Anna Berger", "Lukas Huber",
           "Maria Gruber", "Thomas Wagner",
           "Julia Bauer"),
  city = c("Graz", "Wien", "Salzburg",
           "Linz", "Innsbruck"),
  stringsAsFactors = FALSE
)
```

```
# Produkte
products <- data.frame(
  product_id = 1:4,
  product_name = c("Laptop", "Monitor",
                   "Keyboard", "Mouse"),
  price = c(1200, 300, 80, 25)
)
```

```
# Bestellungen
orders <- data.frame(
  order_id = 1:8,
  customer_id = c(1, 2, 1, 4, 3, 5, 2, 1),
  product_id = c(1, 2, 4, 3, 1, 2, 3, 4),
  quantity = c(1, 2, 1, 1, 3, 1, 2, 5)
)
```

R Zugriff auf Datenbanken

Unsere Daten haben in **R** nun folgendes Aussehen:

```
# Kunden
customers

  customer_id           name      city
1           1   Anna Berger      Graz
2           2   Lukas Huber      Wien
3           3  Maria Gruber  Salzburg
4           4 Thomas Wagner      Linz
5           5   Julia Bauer Innsbruck
```

```
# Produkte
products

  product_id product_name price
1          1       Laptop  1200
2          2      Monitor   300
3          3     Keyboard    80
4          4        Mouse    25
```

```
# Bestellungen
orders

  order_id customer_id product_id quantity
1        1           1          1        1
2        2           2          2        2
3        3           1          4        1
4        4           4          3        1
5        5           3          1        3
6        6           5          2        1
7        7           2          3        2
8        8           1          4        5
```

4. Nun **speichern** wir die 3 Tabellen in der Datenbank:

```
dbWriteTable(con, "customers", customers, overwrite = TRUE)
dbWriteTable(con, "products", products, overwrite = TRUE)
dbWriteTable(con, "orders", orders, overwrite = TRUE)
```

und **trennen** die Verbindung mittels

```
dbDisconnect(con)
```

Nach Ausführen dieser Code-Zeilen sollten Sie nun in Ihrem Arbeitsverzeichnis eine Datei namens `demo_shop.sqlite` haben – unsere Datenbank, auf die wir im folgenden zugreifen wollen. Der Unterschied zur realen Situation ist lediglich, dass sie in einer lokalen Datei vorliegt.

R Zugriff auf Datenbanken

Wollen wir nun (sehr einfache) Berechnungen machen, führen wir folgende Zeilen aus

```
# falls noch nicht geschehen, laden wir die Pakete
require(DBI)
require(RSQLite)

# Herstellen einer Verbindung
con <- dbConnect(SQLite(), "demo_shop.sqlite")
```

Wir möchten nun ein paar einfache Abfragen machen – **R** schickt selbige, wir erhalten ein data frame retour.

```
# welche Tabellen existieren?
dbListTables(con)
```

```
[1] "customers" "orders"    "products"
```

```
# gib mir alle Spalten der customers Tabelle
customers <- dbGetQuery(con, "
  SELECT *
  FROM customers
")

customers
```

```
  customer_id          name      city
1           1   Anna Berger      Graz
2           2   Lukas Huber      Wien
3           3  Maria Gruber  Salzburg
4           4 Thomas Wagner      Linz
5           5   Julia Bauer Innsbruck
```

R Zugriff auf Datenbanken

. . . oder vielleicht wollen wir alle Kunden aus Graz . . .

```
graz_customers <- dbGetQuery(con, "
  SELECT name
  FROM customers
  WHERE city = 'Graz'
")

graz_customers
```

```
        name
1 Anna Berger
```

oder wir wollen wissen, welche Produktmenge im Durchschnitt bestellt wird

```
orders <- dbGetQuery(con, "
  SELECT quantity
  FROM orders
")

mean(orders$quantity)
```

```
[1] 2
```

R Zugriff auf Datenbanken

. . . oder zu guter letzt wollen wir die Tabellen verknüpfen, um zu sehen, wer was in welcher Menge bestellt hat (durch einen *inner join*)

```
order_details <- dbGetQuery(con, "
  SELECT
      c.name,
      p.product_name,
      o.quantity
  FROM orders o
  JOIN customers c
      ON o.customer_id = c.customer_id
  JOIN products p
      ON o.product_id = p.product_id
")

order_details
```

```
          name product_name quantity
1   Anna Berger       Laptop        1
2   Lukas Huber      Monitor        2
3   Anna Berger        Mouse        1
4 Thomas Wagner     Keyboard        1
5  Maria Gruber       Laptop        3
6   Julia Bauer      Monitor        1
7   Lukas Huber     Keyboard        2
8   Anna Berger        Mouse        5
```

Inhalt

Weitere Dateiformate

Dateiformat	Ursprung	Funktion	Paket
JSON	-	fromJSON()	jsonlite
XML	-	read_xml()	xlm2
SQL	SQLite, MySQL, ...	fromJSON()	DBI + RSQLite, MariaDB, ...
.sas7bdat	SAS	read_sas()	haven
.dta	Stata	read_dta()	haven
Parquet	-	read_parquet()	arrow
HDF5	-	h5read()	rhdf5
NetCDF	-	nc_open()	ncdf4
shapefile	-	st_read()	sf
.mat	Matlab	readMat()	R.matlab
Google Sheets	-	read_sheet()	googlesheets4

Inhalt

Auto Datensatz – erste Schritte

- Wir starten mit dem Auto Datensatz und installieren dazu das ISLR2 package.

```
# installieren des Pakets mittels
## install.packages("ISLR2", dependencies = TRUE)

# laden des Pakets
require(ISLR2)
```

- Datensätze in Paketen stehen entweder direkt nach Laden des Pakets mittels `require()` oder `library()` zur Verfügung oder erfordern ein

```
# lade Datensatz
data(Auto)
```

- Man startet gewöhnlich mit einem schnellen Überblick, indem man sich die ersten Zeilen ansieht:

```
# erste (6) Zeilen
head(Auto)
```

```
  mpg cylinders displacement horsepower weight acceleration year origin                 name
1  18         8          307        130   3504         12.0   70      1 chevrolet chevelle malibu
2  15         8          350        165   3693         11.5   70      1         buick skylark 320
3  18         8          318        150   3436         11.0   70      1        plymouth satellite
4  16         8          304        150   3433         12.0   70      1             amc rebel sst
5  17         8          302        140   3449         10.5   70      1               ford torino
6  15         8          429        198   4341         10.0   70      1          ford galaxie 500
```

Auto Datensatz – Erkundung

- Wir wollen zuerst den Datensatz verstehen – wie jede Funktion hat auch jeder Datensatz eine Hilfe-Seite

```
help(Auto)
```

- Wir erfahren dadurch, dass es sich um technische Daten (9 Variablen) zu 392 Automodellen handelt, was sich leicht verifizieren lässt:

```
# erste Zahl: Zeilenanzahl (Beobachtungen), zweite Zahl: Spaltenanzahl (Variablen)
dim(Auto)
```

```
[1] 392   9
```

Bedeutung der einzelnen Spalten:

- `mpg` ... miles per gallon, Benzinverbrauch
- `cylinders` ... Anzahl der Zylinder
- `displacement` ... Hubraum (in *Kubik-Inch* in^3)
- `horsepower` ... Leistung (PS)
- `weight` ... Gewicht in *lbs* (Pfund)
- `acceleration` ... Zeit, um vom 0 auf 60 mph zu beschleunigen (Sekunden)
- `year` ... Modelljahr (Jahrhundert weggelassen)
- `origin` ... Herkunft (1 Amerika, 2 Europa, 3 Japan)
- `name` ... Modellname

Auto Datensatz – Erkundung

- Eine wichtige Frage zu Beginn – gibt es fehlende Werte im Datensatz?

```
# Anzahl fehlender Werte im gesamten Datensatz
sum(is.na(Auto))
```

```
[1] 0
```

```
# Alternative - gibt es fehlende Werte?
anyNA(Auto)
```

```
[1] FALSE
```

- Stellt man die gleiche Frage, möchte aber eine Antwort getrennt nach Spalten (Variablen), so wäre

```
# wende die Funktion 'anyNA' spaltenweise (2) an -- zeilenweise w\"are '1'
apply(Auto, 2, anyNA)
```

```
         mpg    cylinders displacement   horsepower       weight acceleration         year       origin         name
       FALSE        FALSE        FALSE        FALSE        FALSE        FALSE        FALSE        FALSE        FALSE
```

- **Schlussfolgerung**: wir haben die angenehme Situation, dass es **keine** fehlenden Werte im Datensatz gibt.

Auto Datensatz – Erkundung

- Bei diesem (sowie bei den meisten anderen) Datensätzen handelt es sich um ein `data frame`:

```
class(Auto)
```

```
[1] "data.frame"
```

- Der Zugriff auf einzelne Zeilen bzw. Spalten erfolgt durch **eckige Klammern** bzw. ist auch $ zur Extraktion von Spalten möglich.

```
# Zeilen 20 bis 30, Spalten mpg und name
Auto[20:30, c("mpg", "name")]
```

```
   mpg                          name
20  26 volkswagen 1131 deluxe sedan
21  25                  peugeot 504
22  24                  audi 100 ls
23  25                     saab 99e
24  26                     bmw 2002
25  21                  amc gremlin
26  10                    ford f250
27  10                    chevy c20
28  11                   dodge d200
29   9                     hi 1200d
30  27                 datsun pl510
```

Um z.B. den **mittleren Verbrauch** (in `mpg`) aller Autos zu erhalten, könnte man folgendermaßen vorgehen

```
# ist das fachlich richtig?
mean(Auto$mpg)
```

```
[1] 23.44592
```

Auto Datensatz – Erkundung

- Zusammen mit Plots (siehe später) lässt sich die Frage nach atypischen Werten (Ausreißer) oft mittels der `summary()` Funktion beantworten

```
# Zusammenfassung der Daten
summary(Auto)
```

```
      mpg          cylinders      displacement     horsepower        weight      acceleration        year           origin
 Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0   Min.   :1613   Min.   : 8.00   Min.   :70.00   Min.   :1.000
 1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2225   1st Qu.:13.78   1st Qu.:73.00   1st Qu.:1.000
 Median :22.75   Median :4.000   Median :151.0   Median : 93.5   Median :2804   Median :15.50   Median :76.00   Median :1.000
 Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5   Mean   :2978   Mean   :15.54   Mean   :75.98   Mean   :1.577
 3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0   3rd Qu.:3615   3rd Qu.:17.02   3rd Qu.:79.00   3rd Qu.:2.000
 Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0   Max.   :5140   Max.   :24.80   Max.   :82.00   Max.   :3.000

                 name
 amc matador       :  5
 ford pinto        :  5
 toyota corolla    :  5
 amc gremlin       :  4
 amc hornet        :  4
 chevrolet chevette:  4
 (Other)           :365
```

Es wird eine – abhängig vom Skalenniveau – zumeist (aber nicht immer!) sinnvolle Zusammenfassung der Daten **pro Spalte** ausgegeben.

- Generell zu **Skalenniveaus**: 4 verschiedene: **nominal** (Ausprägungen sind Namen, keine Ordnung), **ordinal** (geordnete Kategorien), **intervallskaliert**, **absolutskaliert** (Unterscheidung der letzten beiden Kategorien oft nicht notwendig).

`Auto` Datensatz – Zusammenfassung, to do's

1. Spaltennamen auf **Englisch** $\Longrightarrow$ Deutsch.
2. Damit verbunden sind auch die **Einheiten** (z.B. *Miles per Gallon* anstatt *Liter pro 100 Kilometer*, in^3 anstatt cm^3, *lbs* statt *kilogramm*). Ferner sollte die Variable `year` (70, 71, ...) auf die tatsächlichen Werte (1970, 1971, ...) korrigiert werden.
3. Die ersten 6 Variablen sind numerisch und auch als solche im data frame. Die Variable `origin` ist eigentlich eine **kategorielle Variable**, die im Datensatz aber numerisch hinterlegt ist (sehr schlecht und irreführend). **Muss** korrigiert werden.

Bevor wir zu den eigentlichen Fragestellungen kommen werden, müssen zunächst diese **Korrekturen** vorgenommen werden. Oft: $80\,\%$ Datenvorbereitung (*data cleaning*), $20\,\%$ tatsächliche Datenanalyse.

Auto Datensatz – Datenbereinigung

1. **Änderung** der **Spaltennamen**:

```
# aktuelle Spaltennamen
names(Auto)

[1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"       "acceleration" "year"         "origin"       "name"

# Aenderung durch Zuweisung neuer Namen
names(Auto) <- c("Verbrauch", "Zylinder", "Hubraum", "Leistung", "Gewicht", "Beschleunigung", "Modelljahr", "Herkunft", "Name")
```

2. **Änderung der Einheiten**: Variablen `Verbrauch`, `Hubraum` und `Gewicht`: folgende Zusammenhänge gelten

$$\text{Liter pro 100 km} = \frac{235.15}{\text{mpg}} \qquad \text{cm}^3 = 0.061023 \cdot \text{in}^3 \qquad \text{kg} = \frac{1}{2.20462} \cdot \text{lbs}$$

```
# Verbrauch nun in l/100 km
Auto$Verbrauch <- 235.15 / Auto$Verbrauch

# Hubraum nun in cm^3
Auto$Hubraum <- 2.54^3 * Auto$Hubraum

# Gewicht in kg
Auto$Gewicht <- (1 / 2.20462) * Auto$Gewicht

# Modelljahr im Format 1970, ...
Auto$Modelljahr <- 1900 + Auto$Modelljahr
```

Auto Datensatz – Datenbereinigung

3. **Änderung** der Variable `origin` in eine kategorielle Variable (Faktor):

```
Auto$Herkunft <- factor(Auto$Herkunft,                          # betroffene Spalte
                        levels = c(1, 2, 3),                    # alte Bezeichnungen
                        labels = c("USA", "Europa", "Japan"))   # neue Bezeichnungen
```

4. Nun sind wir fertig, wir sehen uns das Ergebnis an . . .

```
head(Auto)
```

```
  Verbrauch Zylinder  Hubraum Leistung  Gewicht Beschleunigung Modelljahr Herkunft                       Name
1  13.06389        8 5030.829      130 1589.390           12.0       1970      USA chevrolet chevelle malibu
2  15.67667        8 5735.472      165 1675.119           11.5       1970      USA         buick skylark 320
3  13.06389        8 5211.086      150 1558.545           11.0       1970      USA        plymouth satellite
4  14.69688        8 4981.667      150 1557.184           12.0       1970      USA             amc rebel sst
5  13.83235        8 4948.893      140 1564.442           10.5       1970      USA               ford torino
6  15.67667        8 7030.050      198 1969.047           10.0       1970      USA          ford galaxie 500
```

Auto Datensatz – Datenanalyse

Wir möchten im Rahmen einer kleinen **Datenanalyse** die **folgenden Fragen** beantworten:

1. Wir suchen das Modelljahr, den Hubraum und den Namen des Autos mit dem größten Hubraum.
2. Den durchschnittlichen Verbrauch der Autos, getrennt nach Herkunft.
3. Finden Sie den Namen des Autos mit dem größten Verhältnis $\frac{\text{Leistung}}{\text{Gewicht}}$.
4. (Etwas fortgeschritten): Wie sieht die Verteilung der Automarken (nicht der Automodelle) aus?
5. Den Zusammenhang zwischen der Variable `Verbrauch` und den anderen Variablen (beschrieben durch numerische Kenngrößen und Visualisierungen).
6. (Etwas fortgeschritten): können wir den Zusammenhang zwischen dem Verbrauch und den anderen Variablen statistisch beschreiben (mit anderen Worten: gesucht ist ein Vorhersagemodell, wieviel ein Auto verbraucht, wenn die anderen Größen wie Gewicht, Leistung etc. bekannt sind).

Auto Datensatz – Datenanalyse

1. Modelljahr, Hubraum und Namen des Autos mit dem größten Hubraum.

```
Auto[Auto$Hubraum == max(Auto$Hubraum), c("Modelljahr", "Hubraum", "Name")]
```

```
   Modelljahr  Hubraum                    Name
9        1970 7456.114          pontiac catalina
14       1970 7456.114  buick estate wagon (sw)
96       1973 7456.114 buick electra 225 custom
```

Alternative 1: durch Sortierung

```
head(Auto[order(Auto$Hubraum, decreasing = TRUE), c("Modelljahr", "Hubraum", "Name")])
```

```
   Modelljahr  Hubraum                       Name
9        1970 7456.114             pontiac catalina
14       1970 7456.114      buick estate wagon (sw)
96       1973 7456.114     buick electra 225 custom
7        1970 7439.727              chevrolet impala
8        1970 7210.308            plymouth fury iii
95       1973 7210.308 chrysler new yorker brougham
```

Alternative 2: mittels der Funktion `subset()`

```
subset(Auto, subset = Auto$Hubraum == max(Auto$Hubraum), select = c(Modelljahr, Hubraum, Name))
```

```
   Modelljahr  Hubraum                    Name
9        1970 7456.114          pontiac catalina
14       1970 7456.114  buick estate wagon (sw)
96       1973 7456.114 buick electra 225 custom
```

Auto Datensatz – Datenanalyse

2. Durchschnittlicher Verbrauch der Autos, getrennt nach Herkunft.

```
# mit bereits bekannten Mitteln
mean(Auto[Auto$Herkunft == "USA", "Verbrauch"])
```

```
[1] 12.89926
```

```
mean(Auto[Auto$Herkunft == "Europa", "Verbrauch"])
```

```
[1] 8.986467
```

```
mean(Auto[Auto$Herkunft == "Japan", "Verbrauch"])
```

```
[1] 8.060947
```

Eleganter mittels der Funktion `by()`

```
by(data = Auto$Verbrauch, INDICES = Auto$Herkunft, FUN = mean)
```

```
Auto$Herkunft: USA
[1] 12.89926
------------------------------------------------------------------------------------------------------------
Auto$Herkunft: Europa
[1] 8.986467
------------------------------------------------------------------------------------------------------------
Auto$Herkunft: Japan
[1] 8.060947
```

Auto Datensatz – Datenanalyse

2. (Fortsetzung) Ebenso elegant mittels der Funktion `aggregate()`

```
aggregate(Verbrauch ~ Herkunft, data = Auto, FUN = mean)
```

```
  Herkunft Verbrauch
1      USA 12.899263
2   Europa  8.986467
3    Japan  8.060947
```

Auto Datensatz – Datenanalyse

3. Den Namen des Autos (oder der Autos) mit dem größten Verhältnis von Leistung und Gewicht.

Auto Datensatz – Datenanalyse

3. Den Namen des Autos (oder der Autos) mit dem größten Verhältnis von Leistung und Gewicht.

```
# liefert potentiell auch mehrere "Sieger"
(Gewinner <- Auto[Auto$Leistung / Auto$Gewicht == max(Auto$Leistung / Auto$Gewicht), "Name"])
```

```
[1] buick estate wagon (sw)
304 Levels: amc ambassador brougham amc ambassador dpl amc ambassador sst amc concord amc concord d/l amc concord dl 6 ... vw rabbit custom
```

```
# kuerzer, aber eventuell problematisch, hier egal ...
Auto[which.max(Auto$Leistung / Auto$Gewicht), "Name"]
```

```
[1] buick estate wagon (sw)
304 Levels: amc ambassador brougham amc ambassador dpl amc ambassador sst amc concord amc concord d/l amc concord dl 6 ... vw rabbit custom
```

```
# Daten dieses Autos
Auto[which.max(Auto$Leistung / Auto$Gewicht), ]
```

```
   Verbrauch Zylinder  Hubraum Leistung  Gewicht Beschleunigung Modelljahr Herkunft                   Name
14  16.79643        8 7456.114      225 1399.788             10       1970      USA buick estate wagon (sw)
```

Auto Datensatz – Datenanalyse

4. **Verteilung der Automarken**: **R** verfügt ebenso über zahllose Methoden und Funktionen zur Bearbeitung von **Text**-Daten. Wir sehen uns zunächst einen Teil der Spalte `Name` genauer an:

```
# zeige erste 20 Namen
Auto$Name[1:20]
```

```
 [1] chevrolet chevelle malibu   buick skylark 320           plymouth satellite          amc rebel sst
 [5] ford torino                 ford galaxie 500            chevrolet impala            plymouth fury iii
 [9] pontiac catalina            amc ambassador dpl          dodge challenger se         plymouth 'cuda 340
[13] chevrolet monte carlo       buick estate wagon (sw)     toyota corona mark ii       plymouth duster
[17] amc hornet                  ford maverick               datsun pl510                volkswagen 1131 deluxe sedan
304 Levels: amc ambassador brougham amc ambassador dpl amc ambassador sst amc concord amc concord d/l amc concord dl 6 ... vw rabbit custom
```

Offenbar folgt die **Nomenklatur** folgendem Schema:

`Automarke – Modellbezeichnung`

Strategie: wir trennen durch die Verwendung der Funktion `strsplit()` mit Trennungszeichen `'Leerzeichen'` den Namen und behalten nur den ersten Teil. Wendet man diese Strategie auf den ersten Namen an

```
strsplit(x = as.character(Auto$Name[1]), split = " ", fixed = TRUE)
```

```
[[1]]
[1] "chevrolet" "chevelle"  "malibu"
```

Wir erhalten eine Liste mit einem Element (`[[1]]`), das einen Vektor mit den (in diesem Fall) 3 Namensbestandteilen enthält. Die Funktion `strsplit()` verlangt als Eingabe einen Character-Vektor, daher muss der Faktor mittels `as.character()` in einen solchen konvertiert werden.

Auto Datensatz – Datenanalyse

4. (Fortsetzung) Wir wenden nun die Funktion auf den gesamten Namen Vektor im Datensatz an.

```
# teile bei jedem Lehrzeichen
split_temp <- strsplit(x = as.character(Auto$Name), split = " ", fixed = TRUE)
```

Nun wenden wir auf jedes Listenelement (d.h. auf jeden Vektor) die Funktion [an (ja, dies ist auch eine Funktion in **R**).

```
# extrahiere nur das erste Element (=Automarke)
split_temp2 <- lapply(split_temp, "[", 1)
```

Das Ergebnis ist eine Liste, die wir in einen Vektor verwandeln und uns ansehen

```
# Liste -> Vektor
result <- unlist(split_temp2)

# zeige Teil des Ergebnisses
result[1:20]
```

```
 [1] "chevrolet"  "buick"      "plymouth"   "amc"        "ford"       "ford"       "chevrolet"  "plymouth"   "pontiac"    "amc"
[11] "dodge"      "plymouth"   "chevrolet"  "buick"      "toyota"     "plymouth"   "amc"        "ford"       "datsun"     "volkswagen"
```

Auto Datensatz – Datenanalyse

4. (Fortsetzung) Welche und wieviele verschiedene Automarken sind vertreten? Fällt jemandem etwas auf?

```
# eliminiere Duplikate
unique(result)
```

```
 [1] "chevrolet"     "buick"         "plymouth"      "amc"           "ford"          "pontiac"       "dodge"         "toyota"
 [9] "datsun"        "volkswagen"    "peugeot"       "audi"          "saab"          "bmw"           "chevy"         "hi"
[17] "mercury"       "opel"          "fiat"          "oldsmobile"    "chrysler"      "mazda"         "volvo"         "renault"
[25] "toyouta"       "maxda"         "honda"         "subaru"        "chevroelt"     "capri"         "vw"            "mercedes-benz"
[33] "cadillac"      "mercedes"      "vokswagen"     "triumph"       "nissan"
```

```
# wieviele sind dies insgesamt
length(unique(result))
```

```
[1] 37
```

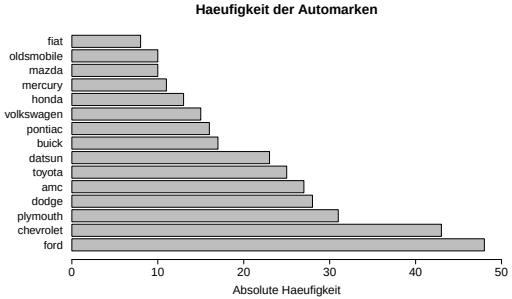
```
# wieviele von welcher Marke vertreten, sortieren nach Haeufigkeit
(tb <- sort(table(result), decreasing = TRUE))
```

```
result
        ford    chevrolet     plymouth        dodge          amc       toyota       datsun        buick      pontiac   volkswagen
          48           43           31           28           27           25           23           17           16           15
       honda      mercury        mazda   oldsmobile         fiat      peugeot         audi     chrysler        volvo           vw
          13           11           10            8            8            7            6            6            6
       opel         saab       subaru        chevy      renault          bmw     cadillac        maxda mercedes-benz        capri
           4            4            4            3            3            2            2            2            2            1
  chevroelt           hi     mercedes       nissan     toyouta      triumph    vokswagen
           1            1            1            1            1            1            1
```

Auto Datensatz – Datenanalyse

4. (Fortsetzung) Zur graphischen Darstellung verwenden wir ein Balkendiagramm, beschränken und aber auf die 15 häufigsten Marken.

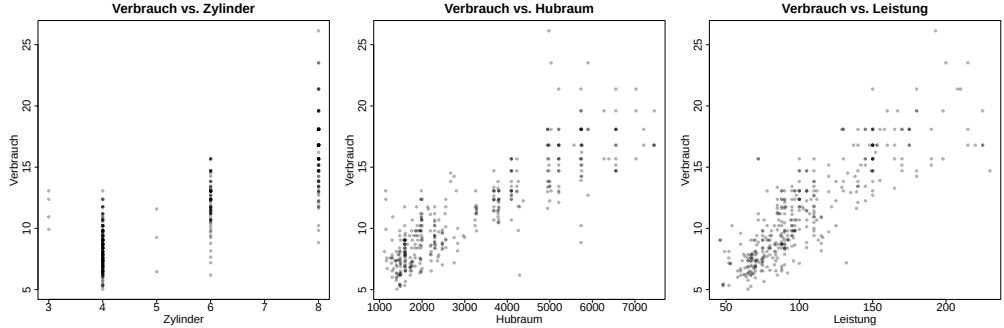
```
# Vorbereitung: groessere Raender links
par(mar = c(5, 10, 4, 2))

# horizontale Balken
barplot(tb[1:15], horiz = TRUE, las = 1, xlim = c(0, 50),
        main = "Haeufigkeit der Automarken", cex.names = 1.5, xlab = "Absolute Haeufigkeit")

# reset der Einstellungen
par(mar = c(5, 4, 4, 2) + 0.1)
```



Auto Datensatz – Datenanalyse

5. Zusammenhang Verbrauch mit den anderen Variablen – wir betrachten zunächst die numerischen Variablen

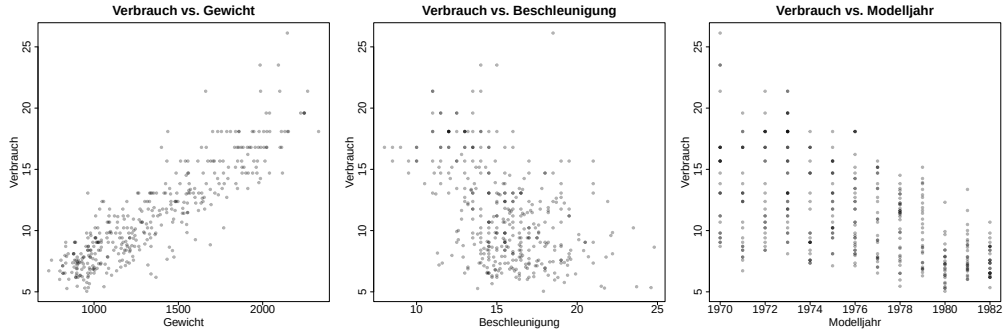
```
# Streudiagramme (scatterplots) - Variablen Zylinder, Hubraum, Leistung
par(mfrow = c(1, 3))  # 3 plots nebeneinander
plot(x = Auto$Zylinder, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Zylinder", ylab = "Verbrauch", main = "Verbrauch vs. Zylinder")
plot(x = Auto$Hubraum, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Hubraum", ylab = "Verbrauch", main = "Verbrauch vs. Hubraum")
plot(x = Auto$Leistung, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Leistung", ylab = "Verbrauch", main = "Verbrauch vs. Leistung")
par(mfrow = c(1, 1)) # reset
```



Auto Datensatz – Datenanalyse

5. (Fortsetzung) Zusammenhang Verbrauch mit den anderen Variablen – wir betrachten zunächst nur numerische Variablen.

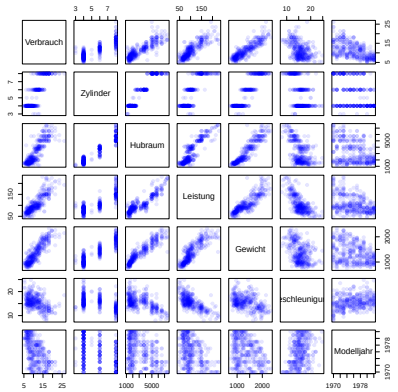
```
# Streudiagramme (scatterplots) - Variablen Gewicht, Beschleunigung und Modelljahr
par(mfrow = c(1, 3))  # 3 plots nebeneinander
plot(x = Auto$Gewicht, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Gewicht", ylab = "Verbrauch", main = "Verbrauch vs. Gewicht")
plot(x = Auto$Beschleunigung, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Beschleunigung", ylab = "Verbrauch",
     main = "Verbrauch vs. Beschleunigung")
plot(x = Auto$Modelljahr, y = Auto$Verbrauch, pch = 19, col = rgb(0, 0, 0, 0.3), xlab = "Modelljahr", ylab = "Verbrauch", main = "Verbrauch vs. Modelljahr")
par(mfrow = c(1, 1)) # reset
```



Auto Datensatz – Datenanalyse

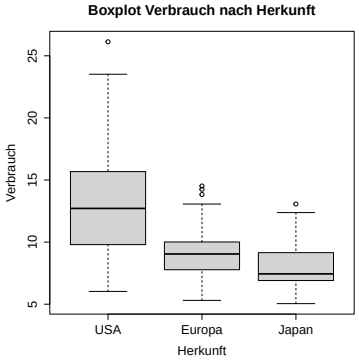
5. (Fortsetzung) Ein Pairs-Plot zeigt die Zusammenhänge zwischen allen Variablen (nicht nur mit der Variable `Verbrauch`) – zu empfehlen nur bei kleiner Variablenanzahl.

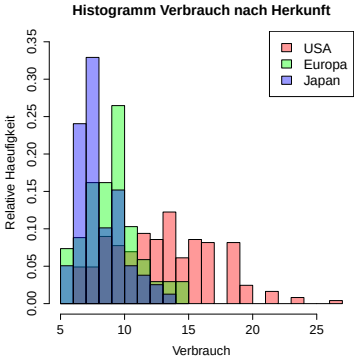
```
# pairs-Plot
pairs(Auto[, 1:7], pch = 19, col = rgb(0, 0, 1, 0.1), cex.labels = 1.3)
```



Auto Datensatz – Datenanalyse

5. (Fortsetzung) Beschreibung des Zusammenhangs zwischen `Herkunft` (kategorielle Variable) und `Verbrauch` (numerisch) – oft werden dazu Boxplots oder (überlagerte) Histogramme (Häufigkeitsverteilungen) angewendet.





Auto Datensatz – Datenanalyse

6. **Vorhersagemodell** für den Verbrauch – ein Einstieg ins Machine Learning . . . Wir betrachten ein sehr einfaches **lineares Modell**:

$$\begin{aligned}\text{Verbrauch} = \beta_0 + \beta_1 \cdot \text{Zylinder} + \beta_2 \cdot \text{Hubraum} + \beta_3 \cdot \text{Leistung} + \\ + \beta_4 \cdot \text{Gewicht} + \beta_5 \cdot \text{Beschleunigung} + \beta_6 \cdot \text{Modelljahr} + \beta_7 \cdot \text{Herkunft}\end{aligned}$$

```
# lineares Modell
lin_mod <- lm(Verbrauch ~ Zylinder + Hubraum + Leistung + Gewicht + Beschleunigung + Modelljahr + Herkunft, data = Auto)
```

In der **Formelnotation** (sehr gebräuchlich und praktisch in **R**, erfordert etwas Gewöhnung) steht die vorherzusagende Größe links vom $\sim$, die Größen, anhand derer eine Vorhersage getätigt werden soll, rechts vom $\sim$ (getrennt durch ein $+$).

Auto Datensatz – Datenanalyse

6. (Fortsetzung) Wir sehen uns das **Ergebnis** an und bemerken aufgrund des hohen R^2 Werts (abzulesen bei `Multiple R-squared`) von 0.89, dass es sich um ein **gutes Modell** handelt. Die Koeffizienten β_j (genauer: Schätzwerte davon) können in der Spalte `Estimate` abgelesen werden.

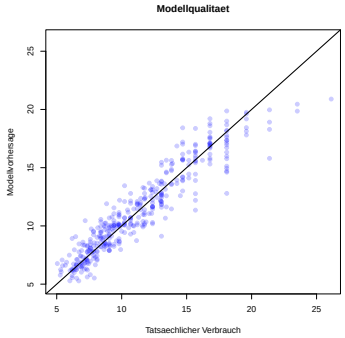
```
# Zusammenfassung des Modells
summary(lin_mod)
```

```
Call:
lm(formula = Verbrauch ~ Zylinder + Hubraum + Leistung + Gewicht +
    Beschleunigung + Modelljahr + Herkunft, data = Auto)

Residuals:
    Min      1Q  Median      3Q     Max
-3.7252 -0.8236 -0.0738  0.7048  5.5766

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     6.058e+02  4.127e+01  14.681  < 2e-16 ***
Zylinder        3.532e-01  1.292e-01   2.734  0.00655 **
Hubraum        -4.635e-04  1.879e-04  -2.467  0.01406 *
Leistung        2.998e-02  5.514e-03   5.437 9.67e-08 ***
Gewicht         5.830e-03  5.810e-04  10.035  < 2e-16 ***
Beschleunigung  8.101e-02  3.951e-02   2.051  0.04099 *
Modelljahr     -3.072e-01  2.083e-02 -14.750  < 2e-16 ***
HerkunftEuropa -6.956e-01  2.278e-01  -3.053  0.00242 **
HerkunftJapan  -4.763e-01  2.223e-01  -2.142  0.03282 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.33 on 383 degrees of freedom
Multiple R-squared:  0.8868, Adjusted R-squared:  0.8845
F-statistic: 375.1 on 8 and 383 DF,  p-value: < 2.2e-16
```

Auto Datensatz – Datenanalyse

6. (Fortsetzung) Wie gut ist das Modell? **Mit welcher Genauigkeit** kann der Verbrauch eines Autos vorhergesagt werden bei Kenntnis der anderen Größen? Wir vergleichen die tatsächlichen Verbräuche mit den durch das Modell vorhergesagten Verbräuchen . . .

```
# vom Modell vorhergesagt
vorhersagen <- predict(lin_mod)

plot(x = Auto$Verbrauch, y = vorhersagen, pch = 19, col = rgb(0,0,1,0.2),
     xlab = "Tatsaechlicher Verbrauch", ylab = "Modellvorhersage",
     main = "Modellqualitaet", xlim = c(5, 26), ylim = c(5, 26))
abline(a = 0, b = 1, lwd = 2)
```



Auto Datensatz – Datenanalyse

6. (Fortsetzung) Mit welchem **durchschnittlichen Fehler** in der Vorhersage des Verbrauchs hat man zu rechnen bei Verwendung des Modells?

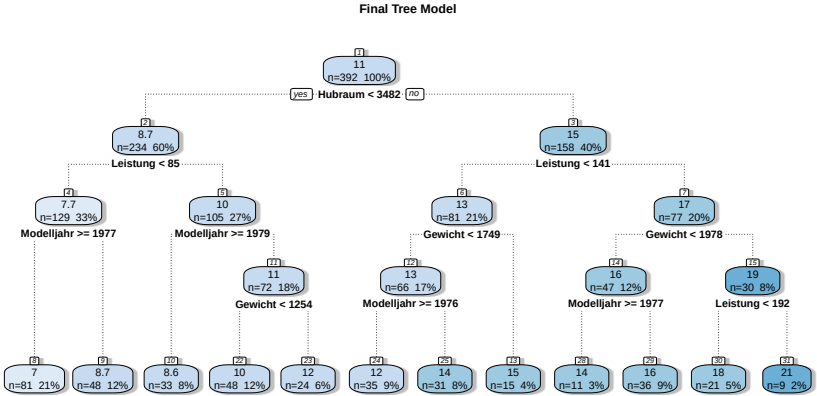
```
# MAE = mean absolute error
(MAE <- mean(abs(vorhersagen - Auto$Verbrauch)))
```

```
[1] 0.9806444
```

Wie sehen, dass wir – im Durchschnitt – um etwa 1 l/100km danebenliegen. Ein akzeptabler Fehler? Wenn ja → fertig; wenn nein → andere Methode, bessere oder mehr Features (x-Variablen), ...

Auto Datensatz – Datenanalyse

6. (Fortsetzung) Wir testen noch eine andere Modellklasse zur Vorhersage des Verbrauchs – sogenannte Regressionsbäume. Wir erstellen zunächst das Modell, optimieren es, geben – graphisch – das finale Modell aus und versuchen, es zu verstehen . . .



Auto Datensatz – Datenanalyse

Baum-Modelle (*trees* oder eine Weiterentwicklung davon, sogenannte *Random Forests*) haben eine Reihe von Vorteilen:

- Sie können für Regressions- (vorherzusagende Größe ist numerisch) sowie auch Klassifikationsaufgaben (vorherzusagende Größe ist kategoriell) verwendet werden.
- Auch können Sie für sogenannte unsupervised learning Aufgaben verwendet werden (es gibt gar keine vorherzusagende Größe).
- Sie können – im Gegensatz zu den meisten anderen Methoden – mit fehlenden Werten umgehen.
- Die x-Variablen (auf denen eine Vorhersage basiert) können **beliebiges Skalenniveau** haben (nominal, ordinal, numerisch).
- Sie liefern in vielen Fällen interpretierbare Ergebnisse.
- Sie führen eine automatische Variablenselektion durch. So werden beispielsweise die Variablen `Zylinder`, `Beschleunigung` und `Herkunft` **nicht** im Modell verwendet $\rightarrow$ weniger Anforderungen an die Daten (einfachere Messungen), einfachere/interpretierbarere Modelle.

Auto Datensatz – Datenanalyse

Die erforderlichen **R**-Code Zeilen hinter dem Modell der vorigen Folie:

```r
# wir laden zunaechst ein paar Pakete ... (muessen zuerst installiert werden)
require(RColorBrewer); require(rattle); require(rpart.plot)

# wir erstellen das Modell
set.seed(123)
reg_tree <- rpart(Verbrauch ~ Zylinder + Hubraum + Leistung + Gewicht + Beschleunigung + Modelljahr + Herkunft, data = Auto,
                  control = rpart.control(cp = 0))

# wir ermitteln die optimale Modellkomplexitaet
row_with_min <- which.min(reg_tree[["cptable"]][, "xerror"])
opt_cp1 <- reg_tree[["cptable"]][row_with_min, "CP"]
se1 <- reg_tree[["cptable"]][row_with_min, "xstd"]
fin_row <- min(which(reg_tree[["cptable"]][, "xerror"] <=  reg_tree[["cptable"]][row_with_min, "xerror"]+ se1))
opt_cp2 <- reg_tree[["cptable"]][fin_row, "CP"]

# Optimierung hinsichtlich Komplexitaet
final_tree_model <- prune(reg_tree, cp = opt_cp2)

# Plotten des Modells
fancyRpartPlot(final_tree_model, sub = "", palettes = c("Blues"),
               main = "Final Tree Model", mar = c(2, 1, 4, 1))
```

Inhalt

Zusammenfassung – Ziele erreicht

1. Einführung und Setup
2. Grundlagen in **R**
3. Daten importieren und verstehen
4. Daten bearbeiten, visualisieren und einfache Modelle

Weitere Kurse im Rahmen des DIH

Folgende weitere Kurse werden von uns bis Sommer 2026 angeboten (weitere werden folgen):

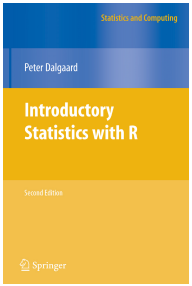
Titel	Dauer (Tage)	Datum
R in a Day – Einstieg in die Datenanalyse mit **R**	1	31.3., 3.6.
Objekte finden, verfolgen, verstehen: Computer Vision praktisch	0.5	9.2
Python kompakt: Einführung für EinsteigerInnen	0.5	24.2., 10.3.
Python in a Day – Programmieren leicht gemacht	1	2.3., 16.3.
Von Tabellen zur Automatisierung – Einstieg in Python für Excel User	0.5	6.3., 20.3.
Von Daten zu Prognosen – Machine Learning mit Python	0.5	31.3., 7.4.
Work smarter, not harder: Workflow-Management in der Datenanalyse	0.5	3.4., 10.4.
Von Zahlen zu Bildern: Interaktive Dashboards mit Python	0.5	20.4., 27.4.
Daten sichtbar machen – Visualisierung mit **R**	0.5	12.5., 19.5.
Mehr als nur `ChatGPT`: Large Language Models für KMUs	0.5	2.6., 9.6.
Machine Learning Summer School	5.0	13. – 17.7.

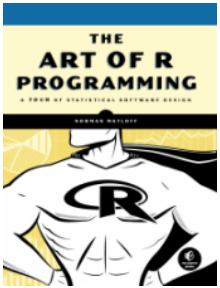
Einladung: Gerne uns Themen bekanntgeben, die für Sie von Interesse sind (michael.melcher@fh-joanneum.at).

Inhalt

Literatur zu **R** – Klassiker

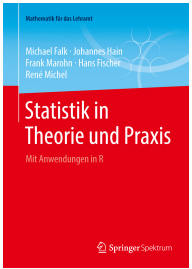
- **Peter Dalgaard**: *Introductory Statistics with R*, Springer 2008
- **Uwe Ligges**: *Programmieren mit R*, Springer 2009
- **Normal Matloff**: *The Art of R Programming*, no starch press, 2011.

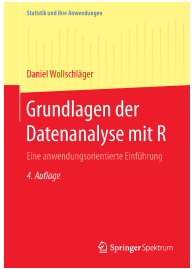




Literatur zu **R**

- **F. Kronthaler**: *Statistik angewandt. Datenanalyse mit dem* **R** *Commander*, Springer 2016.
- **M. Falk et al.**: *Statistik in Theorie und Praxis mit Anwendungen in R*, Springer 2014.
- **D. Wollschläger**: *Grundlagen der Datenanalyse mit R*, Springer 2017.





R Serien

- Use R! Serie: seit 2006, bisher (Stand J"anner 2026) 92 Bücher

  https://www.springer.com/series/6991/books

- The R Series (Chapman & Hall/CRC): bisher 74 Titel seit 2007

  https://www.routledge.com/Chapman--HallCRC-The-R-Series/book-series/CRCTHERSER

- O'Reilly Book Series, Manning Publications & no starch press, 168 Bücher, Hörbücher, Videos, Kurse

  https://www.oreilly.com/search/skills/r/?rows=100&order_by=created_at&language=en&language=de

Online Kurse

- Online Kurse bei coursera

  https://www.coursera.org/

- HarvardX – Online Kurse der Harvard University

  https://www.edx.org/school/harvardx

- Kostenlose online Kurse des Departments of Statistics des PennState Eberly College of Science unter

  https://newonlinecourses.science.psu.edu/statprogram/

- Mehr als 300 online Kurse bei datacamp.com

  https://www.datacamp.com/courses/tech:r

  Jahresbeitrag ~ 130 Euro; Kurse über **R**, Python, SQL, Git, Shell Programming, ..., Videos, Übungen, Datensätze, Folien; **R** Kurse über Gebiete wie Graphik (ggplot2, ggvis, lattice), RStudio, RMarkdown, Zeitreihenanalyse, Daten Import und Export, Machine Learning, ...

Wir bedanken uns für Ihre Teilnahme!