Inhalt

[1 Tibbles als Datensheets 3](#_Toc164239393)

[1.1 Import von Daten 3](#_Toc164239394)

[1.2 Export von Daten 6](#_Toc164239395)

[2 Daten auswählen: filter & select 7](#_Toc164239396)

[2.1 Filtern 7](#_Toc164239397)

[2.2 Select 9](#_Toc164239398)

[2.3 Arrange 11](#_Toc164239399)

[3 Data Wrangling: Mutate, rename, separate, unite 12](#_Toc164239400)

[3.1 Anlegen einer Sequenz 12](#_Toc164239401)

[3.2 Umbenennen von Variablen 12](#_Toc164239402)

[3.3 Modus und Klasse rekodieren 12](#_Toc164239403)

[3.4 Numerische Variablen 1: Werte rekodieren 13](#_Toc164239404)

[3.5 Numerische Variablen 2: Kategorisieren von kontinuierlichen Variablen 13](#_Toc164239405)

[3.6 Numerische Variablen 3: Index bilden 14](#_Toc164239406)

[3.7 Kategoriale Variablen 1: Dummies bilden 15](#_Toc164239407)

[3.8 Kategoriale Variablen 2: Kategorien zusammenlegen 15](#_Toc164239408)

[3.9 Kategoriale Variablen 3: Kategorien neu ordnen 16](#_Toc164239409)

[3.10 Kategoriale Variablen 4: Kategorien umbenennen / rekodieren 17](#_Toc164239410)

[3.11 Bedingtes Berechnen mit ifelse() und case\_when() 17](#_Toc164239411)

[3.12 Eigene Funtionen in mutate einbinden 18](#_Toc164239412)

[3.13 Scientific notation ändern 18](#_Toc164239413)

[3.14 Variablen zerlegen und zusammenfügen mit separate und unite 18](#_Toc164239414)

[3.15 Datensätze joinen 20](#_Toc164239415)

[3.16 pivot\_longer() und pivot\_wider: Format der Daten ändern 22](#_Toc164239416)

[3.17 Programming und Funktionen 24](#_Toc164239417)

[4 Explorative Datenanalyse 35](#_Toc164239418)

[4.1 Globale Analysen 35](#_Toc164239419)

[4.2 Häufigkeiten (uni- und bivariat) 36](#_Toc164239420)

[4.3 Univariate deskriptive Statistiken 40](#_Toc164239421)

[4.4 Kreuztablen mit Mittelwerten 41](#_Toc164239422)

[4.5 Korrelationen 41](#_Toc164239423)

[4.6 Missing data: Analyse und treatment 42](#_Toc164239424)

[5 Zeit 47](#_Toc164239425)

[5.1 Parsing: Umwandlung in das Format date oder datetime 47](#_Toc164239426)

[5.2 Extraktion von Teilelementen 49](#_Toc164239427)

[5.3 Runden 51](#_Toc164239428)

[6 Text 53](#_Toc164239429)

[6.1 Grundprozeduren in der Manipulation von Text 53](#_Toc164239430)

[6.2 str\_c() und unite: Zusammenfügen von strings 54](#_Toc164239431)

[6.3 str\_split() & separate(): Zerteilen eines strings / Tokenizing 55](#_Toc164239432)

[6.4 str\_detect: Pattern suchen und zählen 56](#_Toc164239433)

[6.5 str\_extract() und str\_sub(): Extrahieren von Zeichen 57](#_Toc164239434)

[6.6 str\_remove(): Löschen von Zeichen 60](#_Toc164239435)

[6.7 str\_replace(): Replace / Ersetze 60](#_Toc164239436)

[6.8 Regular expressions 60](#_Toc164239437)

[6.9 Beispiele für regexes 65](#_Toc164239438)

[6.10 Textfeatures (GIBT’S WOHL NICH MEHR) 67](#_Toc164239439)

[6.11 Part-of-speech (POS) tagging 68](#_Toc164239440)

[7 Grafiken 70](#_Toc164239441)

[7.1 Scatterplot 71](#_Toc164239442)

[7.2 Linien und Kurven 77](#_Toc164239443)

[7.3 Darstellen von Statistiken 79](#_Toc164239444)

[7.4 Barplots 83](#_Toc164239445)

[7.5 Histogramme und Polygone 88](#_Toc164239446)

[7.6 Density plots 91](#_Toc164239447)

[7.7 Ridges plots 92](#_Toc164239448)

[7.8 Boxplots 93](#_Toc164239449)

[8 Grafikparameter und Generelles 95](#_Toc164239450)

[8.1 Zwei Grafiken überlagern 95](#_Toc164239451)

[8.2 Farben 95](#_Toc164239452)

[8.3 Manuelle Färbung 95](#_Toc164239453)

[8.4 Achsen 96](#_Toc164239454)

[8.5 Text / labels 98](#_Toc164239455)

[8.6 Legenden 100](#_Toc164239456)

[8.7 Themes 101](#_Toc164239457)

[8.8 Grafiken speichern 102](#_Toc164239458)

[9 Broom 103](#_Toc164239459)

[9.1 Videos 103](#_Toc164239460)

[9.2 Funktionen 103](#_Toc164239461)

[9.3 Genestete Daten 104](#_Toc164239462)

[9.4 Nutzen 105](#_Toc164239463)

# Tibbles als Datensheets

## Import von Daten

### Aus csv files

* csv ist das üblichste oder häufigste Format. Es enthält Daten die durch Kommata oder Semikola abgetrennt sind

Age;Attrition;DailyRate;DistanceFromHome;Education;EmployeeCount;EmployeeNumber;EnvironmentS…¹

41;Yes;1102;1;2;1;1;2;Female;94;3;2;4;5993;19479;8;Yes;11;3;1;80;0;8;0;1;6;4;0;5

49;No;279;8;1;1;2;3;Male;61;2;2;2;5130;24907;1;No;23;4;4;80;1;10;3;3;10;7;1;7

37;Yes;1373;2;2;1;4;4;Male;92;2;1;3;2090;2396;6;Yes;15;3;2;80;0;7;3;3;0;0;0;0

* Je nachdem, ob die Daten Kommata als Trennzeichen haben oder Semikola verwendt man die Funktionen
  + read\_csv (Kommata)
  + read\_csv2 (Semikola)
* Hier der Import eines HR-Datensatzes von meiner github-Seite.

hrdata <- read\_csv2("https://raw.githubusercontent.com/IcarusAE/BusinessAnalytics/main/Ressourcen/hr\_data.csv")

Diese Daten werden die Grundlage für die weiteren Kapitel sein

### Import aus Excel

* readxl package (von Wickham & Bryan)
* Funktion read\_excel

read\_excel("dataset.xlsx")

* Schreibt man die Funktion in dieser Form, wird das erste sheet eingelesen--mit dem sheet-Argument kann/muss man hintere sheets auswählen:

read\_excel("test.xlxs", sheet = "es level")

* Welche sheets es gibt, erfährt man über

"test.xlsx" %>%  
 excel\_sheets()

Voraussetzung ist, dass der Pfad durch setwd() gesetzt wurde oder dadurch, dass es ein Projekt ist, klar ist. Wenn nicht muss er explizt seint, z.B. „C://data//test.xlsx")

### Import aus der Zwischenablage

library("clipr")

topic\_labels <- read\_clip\_tbl()

### Import größerer Datensätze

* Anstatt mit read\_csv2() kann man mit dem vroom package Daten einlesen. Das ist v.a. bei großen Datensätzen deutlich schneller

library(vroom)

data = vroom("data.csv")

* Man kann mit dem col\_select-Argument (siehe unterer link) Spalten auswählen. Das ist bei großen Datensätzen schneller, als erst alles einzulesen und dann in select() zu pipen. Bei mehreren Variablen diese mit c() bündeln
* Mehr info: <https://www.tidyverse.org/blog/2019/05/vroom-1-0-0/>
* Mit dem **n\_max-**Argument kann man sowohl mit readr als auch vroom die Anzahl von Zeilen/Fällen bestimmen, die eingelesen werden sollen. Auch das macht bei großen files Sinn, wenn man nur mal probieren will
* **Real Big Data**

Wenn Daten seeehr groß sind, macht es Sinn, nur bestimmte Variablen einzulesen. Welche Variablen es gibt, bekommt man hier mit heraus:

read.table("C:\\...\data.csv",   
 head = TRUE,   
 nrows = 5,  
 sep = ",")[- 1, ] %>%   
 as\_tibble()

Das liest die header und die ersten 5 Fälle ein. Dann die Variable auswählen und mit dem folgenden Code einlesen (Hinweis: read\_csv2("…") %>% select() geht nicht, weil der select-Befehl erst nachgeschaltet ist).

library(data.table)

selected\_predictors <- c("pid","pnr", "syear", "pab0004")

data <- fread("pl.csv", select = selected\_predictors) %>%   
 as\_tibble()

### Import von googlesheets Daten

* Das geht mit dem googlesheets4 package.
* API-Zugriff. Beim ersten mal muss eine Verbindung zum sheet über den eigenen google-account hergestellt sein (und man muss natürlich für das sheet from Urheber freigeschaltet sein).

gs4\_auth()

* Dieser command prompted die API. Hier muss man den google-account auswählen und tidyverse Zugriff gestatten
* **Zugriff auf sheets:** Grundfunktion ist

data <- read\_sheet(URL)

* **Auswahl bestimmter sheets:** Die o.g. Funktion liest automatisch das erste Sheet ein. Wenn man ein anderes will, kann man das mit dem sheets-Argument wählen

df <- read\_sheet("....", sheet = "ES level")

* **Unklare Variablenformate:**
  + Eine Schwierigkeit kann bestehen, dass Rstudio die Modi nicht durgängig versteht und dann eine odere mehrere Variablen als Liste mit gemischen Modi erstellt. Dies kann man mit „col\_types“ umgehen. Nachteil ist , mann muss das für alle durchdeklinieren

df <- read\_sheet("....",   
 col\_types = 'ccccdccccddddcddddddccccccc')

* + Wie man sieht kommt nach col\_tyes die Liste aller Variablentypen (hier waren das 27 Variablen = 27 explizite Typen!). Wie man sieht, ist das sehr umständlich, wenn der Datensatz viele Variablen hat
  + Möglichkeiten für col\_types=
    - c = character
    - d = double
    - i = integer
    - ? = educated guess (was default ist)

### Umwandlung von Dataframes in Tibbles

* Manchmal liegen die Daten als dataframes vor; sie lassen sich aber einfach in ein tibble umwandeln. Hier mal am Beispiel des mtcars-Datensatzes, der beim Starten von R automatisch mitgeladen wird.

mtcars <- as\_tibble(mtcars)

* Achtung: Manchmal haben datafraames sogenannte **Rownames**. Dies sind labels oder Namen für Zeilen. In mtcars sind das Autotypen. Wenn man mit dem o.g. Befehl den dataframe umwandelt, gehen die verloren. Hier mal mtcars asl dataframe:

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

(Man sieht in der ersten Spalte den Autotyp—der ist aber keine Variable sondern der rowname (→ kein Spaltenkopf)

* Mit dem „rownames“-Argument werden sie als eine explizite Variablen umgewandelt

mtcars <- as\_tibble(mtcars, rownames="car\_type")

# A tibble: 32 × 12

car\_type mpg cyl disp hp drat wt qsec vs am gear carb

<chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 Mazda RX4 21 6 160 110 3.9 2.62 16.5 0 1 4 4

2 Mazda RX4 Wag 21 6 160 110 3.9 2.88 17.0 0 1 4 4

3 Datsun 710 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1

4 Hornet 4 Drive 21.4 6 258 110 3.08 3.22 19.4 1 0 3 1

* + Nun ist car\_type eine richtige Variable
  + Anm. Wenn du den reinen as\_tibble(…) Befehl ausgeführt hast (und die rownames schon weg sind), lässt sich durch data(mtcars) der ursprüngliche dataframe wiederherstellen.

### Variablennamen aufräumen mit clean\_names()

* Sobald man seine Daten eingelesen hat, sollte man die Funktion clean\_names() aus dem janitor Paket anwenden. Dies säubert—wie der Name sagt—die Variablennamen. Beispiele
  + Ersetzen von Leerzeichen durch underscores ('firm size' → firm\_size)
  + Ersetzen von Großbuchtstaben in Kleinbuchstaben (Attrition → attrition)
  + Erkennen von Teilnamen und Standardisierung (FirmSize → firm\_size)

library(janitor)

hrdata <- hrdata %>%   
 clean\_names()

## Export von Daten

Hinweis: Da alles im Tidyverse ein Tibble ist, gilt das folgende für alle Tibbles—und damit Rohdaten als auch Ergebnisse von Analysen

### Als csv speichern

write\_csv2(semdata, "semdata\_neu.csv")

### Als xlsx speichern

library(writexl)

mtcars %>%   
 write\_xlsx("test.xlsx")

### Tibbles über die Zwischenablage (z.B. in Excel) kopieren

write.table(data,"clipboard",sep="\t",col.names=NA)

### Umlaute bewahren und csv als UTF8 speichern

* Wenn man Umlaute etc. in den Variablennamen oder strings hat, werden die in R unschön eingelesen. Das kann man beheben, in dem man die csv als UTF8 speichert. Das geht aber nicht in Excel. Stattdessen:
  + Die Daten mit Excel laden und dort als csv speichern
  + Mit der rechten mousetaste auf das file klicken → mit dem Editor öffnen
  + Dann „speichern unter“ wählen und UTF8 auswählen. Die Endung .txt durch „alle Dateien“ ersetzen.

# Daten auswählen: filter & select

## Filtern

### Positive Auswahl

* **Grundschema**

hrdata %>%  
 filter(education == 2)

* Hierbei sind alle logischen Verknüpfungen möglich
  + == ist gleich
  + != ist nicht
  + > größer
  + > kleiner
  + >= größer/gleich
  + >= kleiner/gleich
* **Nach zwei Bedingungen filtern**

hrdata %>%   
 filter(hourly\_rate > 50 & age < 40)

→ Filtert die Personen, die mehr als 50h arbeiten *und* jünger als 40 sind

hrdata %>%   
 filter(hourly\_rate > 50 | age < 40)

→ Filtert die Personen, die mehr als 50h arbeiten *oder* jünger als 40 sind

* **Level eines Faktor** **oder Characters filtern**.

hrdata %>%   
 filter(gender == "Female")

* **Mehrere Werte / Kategorien einer Variable filtern**

hrdata %>%   
 filter(Education %in% c(1,2))

Sind die level strings (“low“, „high”, “medium”) kommen diese in die Funkton:

hrdata %>%   
 filter(education %in% c("low","high"))

* **Mehrere Werte ausschließen** durch ! ("nicht")

hrdata %>%   
 filter(education %in% c(1,2))

→ Nimmt alle *außer* 1 und 2

* **Bereich einer Variable filtern**

hrdata %>%

filter(education %in% c(2:4))

→ Nimm alle von 2-4

* **Bestimmte Fälle (Zeilennummern) rausziehen**

hrdata %>%   
 slice(10:15)

* + Zieht die Zeilen 10 bis 15
  + Kann interessant sein, um sich bestimmte Tiele eines tibbles anzuschauen
  + Kann v.a. interessant sein, innerhalb von Gruppen (v.a. wenn sie groß sind)
* **Jeweils die ersten x Zeilen in mehreren Gruppen rausziehen**

hrdata %>%   
 group\_by(gender) %>%   
 slice(1:3)

🡪 Zeigt immer nur die ersten 3 Fälle jeweils für Männer und Frauen an.

* Alternativ kann man aus jeder Gruppe eine **Zufallsauswahl** treffen:

hrdata %>%   
 group\_by(gender) %>%   
 slice(1:3)

* Und man kann sich die **höchsten n Werte** in jeder Gruppe ausgeben lassen. Hier die Top 3

hrdata %>%   
 group\_by(gender) %>%   
 top\_n(3, monthly\_income)

"Zeige für Männer und Frauen jeweils die top 3 im monthly income"

* **Missings anzeigen/filtern** (mcht in dem Beispiel keinen Sinn, weil keine Missings existieren)

hrdata %>%  
 filter(is.na(age))

### Negativ-Auswahl

*Anm.* das ist nichts anderes, als Fälle nach einer Bedingung zu **löschen**

* **Negativ filtern:**

hrdata %>%  
 filter(age != 18)

oder

hrdata %>%  
 filter(! age == 18)

* **Mehrere Werte löschen**

hrdata %>%  
 filter(! age %in% c(18,19,20))

* **Einen Bereich elimineren**

hrdata %>%  
 filter(! age %in% c(18:25))

* **Missings eliminieren**

hrdata %>%  
 filter(is.na(age))

### Zufallsstichprobe aus dem Datensatz ziehen

hrdata %>%   
 slice\_sample(n=10) #Achtung, „n“ ist essentiell, weil es ja auch prop sein kann

hrdata %>%   
 slice\_sample(prop = .5) #Ziehung von 50%

Geht auch pro Gruppe (→ zieht 5% aus jeder Gruppe)

hrdata %>%   
 group\_by(gender) %>%   
 slice\_sample(prop = .05) %>%   
 ungroup()

## Select

### Basis

* select klappt mit Variablennamen, als auch Nummern (und Kombi) und Bereichen von

hrdata %>%   
 select(1, 3:5, monthly\_income, total\_working\_years:work\_life\_balance)

* + Hier mal alle Möglichkeiten in einem (Anm. Nummern sollt man vermeiden
  + Mittels select kann man die Reihenfolge der Variablen ändern. Dazu müssen aber alle enthalten sein. Will man das nicht, ist die Sub-Funktion „everything()“ extrem hilfreich:

hrdata %>%   
 select(monthly\_income, over\_time, everything())

Holt MonthlyIncome und Overtime nach vorne—danach kommt der Rest wie gehabt

* **Negativ-Auswahl**

hrdata %>%   
 select(-age, -gender)

oder

hrdata %>%  
 select(-c(age, gender) )

* + Eliminert die Variablen AG und Gender
  + Auch das geht mit Bereichsoperator „:“ und concatenate („c“)

hrdata %>%   
 select(-c(age:gender))

Wirft die Variablen Age bis Gender raus

### Auswahl aufgrund von Namens-Elementen

* **Selektion auf Basis von Namenselementen**

hrdata %>%   
 select(contains("job"))

Wählt alle Variablen, die im Namen "Delay" haben

* **Ausschluss** von Variablen, die ein bestimmtes Wort im Variablennamen haben

hrdata %>%   
 select(-contains("job"))

Schmeißt alle raus, die "Job" enthalten. Mehrere Begriffe gehen wieder mit c()

hrdata %>%   
 select(contains(c("job", "years")))

* Mit starts\_with und ends\_with kann das spezifischer geschehen:

hrdata %>%   
 select(ends\_with("year"))

Wählt Vaiablen, die mit year enden

Analog: Anfangs-Teile eines Namens selektieren

hrdata %>%  
 select(starts\_with("year"))

### Nach Variableneigenschaften selektieren

* Geht mit select\_if()
* Z.B. Variablen nach Klasse selektieren

hrdata %>%   
 select\_if(is.numeric)

Analog is.factor, is.character

### Sonstiges

* **Anzeige aller Werte oder Levels** einer Variable oder der Kombination

hrdata %>%   
 select(education) %>%   
 distinct()

* + Zeigt alle einziartigen level von Education an. Nimmt in select() 2 Vairablen, werden die Kombinationen angezeigt.
  + Anm. In der Regel ist count() besser weil es dasselbe zeigt plus die Häufigkeiten.
* **Identifizieren von Duplicates:** Geht mit get\_dupes() aus dem janitor package. **Dies listet alle duplicates auf.** *Hier mach ich erst mal künstlich 5 dublicates, in dem ich aus mtcars 5 zeilen ziehe und die den Daten neu hinzufüge. Der einzig relevante Befehl sist aber get\_dupes()*

hrdata %>%   
 janitor::get\_dupes()

* + Ohne weitere Argumente werden so identische Fälle über alle Variablen der Daten angezeigt.
  + Will man das einschränken, kann man Zielvariablen als Argumente angeben:

hrdata %>%   
 janitor::get\_dupes(age, gender, education)

* **Duplicates löschen:**
  + distinct() löscht alle identischen Zeilen (bzl. aller Variablen im Datensatz)

hrdata %>%   
 distinct()

* + Will man die Variablen bestimmen, hinsichtlich derer die Duplicates gelöscht werden, dann muss „.keep\_all“ rein, sonst löscht es nicht nur die Dups sonst auch die nicht-benannten Variablen)

hrdata %>%   
 distinct(age, gender, .keep\_all = TRUE)

* + Nimmt man nur distinct() ohne Argument, schmeißt R **“dublicate rows”** raus!

## Arrange

### Einfaches Sortieren nach einer Variable

* **Aufwärts**

hrdata %>%   
 arrange(age)

* Abwärts

hrdata %>%   
 arrange(-age)

### Sortieren nach 2 Variablen

hrdata %>%   
 arrange(education, age)

* + Sortiert zuerst nach Bildung, dann nach Alter

# Data Wrangling: Mutate, rename, separate, unite

## Anlegen einer Sequenz

* Z.B. hilfreich, um eine ID zu erzeugen

hrdata %>%  
 mutate(id = row\_number())

## Umbenennen von Variablen

hrdata <- data %>%   
 rename(alter = age, geschlecht = gender)

Links stehen die Zielvariablen, rechts die Ursprungsvariablen. Alternativ geht das in der select-Funktion

hrdata <- data %>%   
 select(alter = age, geschlecht = gender)

Dabei werden halt dann auch nur diese ausgewählt. Bietet sich halt an, wenn man die sowieso machen und sie gleichzeitig umbenennen möchte.

## Modus und Klasse rekodieren

* Background
  + Modus: Datentypus (numeric, character, logical)
  + Klasse: Higher-Ordner typus der Variable (z.B. dataframe, factor, date, matrix, numeric, character).

Wie man sieht, kann numeric und character sowohl ein Modus eines Datums sein auch die Klasse der Variable.

* **Für eine Variable**

hrdata %>%   
 mutate(education = as.factor(education))

* + Analog geht as.numeric und as.character
  + **Achtung:** Wenn man eine Character-Variable direkt in eine numerische umwandelt, werden alle Einträge gelöscht. Auch wenn charater-Variablen sehr handlich sind (man kann sie direkt in die Regression packen), empfiehlt es sich, sie am Anfang alle in Faktoren zumwandeln.
* **Für mehrere Variablen**

hrdata %>%   
 mutate\_at(.vars = c("education", "attrition", "gender"), .funs = factor)

bzw. in numerisch

mutate\_at(.vars = c("x","y","z"),   
 .funs = **as.**numeric )

(Hier mal generisch anhand der Variablen x, y und z.

* **Modus rekodieren für alle Variablen eines Typs**

hrdata %>%  
 mutate\_if(is.character, factor)

* + Erstes ist Bedingungsprüfung
  + Anm. Macht für hrdata keinen Sinn, weil es keine character gibt
  + Achtung: Wenn man in numeric umwandeln möchte muss da as.numeric hin!

data %>%  
 mutate\_if(is.integer, as.numeric)

## Numerische Variablen 1: Werte rekodieren

* Häufig hat man kategoriale Variablen, die man rekodieren möchte. HÄufigster Fall: Man hat eine item-Batterie in einem Fragebogen, und inige items sind invertiert (z.B. “ich bin unzufrieden” anstatt “ich bin zufrieden”). D.h. man möchte die Skala quasi umdrehen.
* Das geht auf zwei Wege. Die erste ist die dafür geeignete recode()-funktion.

hrdata %>%   
 mutate(env\_dissat = recode(environment\_satisfaction,  
 "1" = 4,  
 "2" = 3,  
 "3" = 2,  
 "4" = 1)  
 )

* + Wichtig, die neuen Werte (immer links müssen seltsamerwiese in Anführungszeichen stehen.
  + Die zweite Art läuft über die case\_when() Subfunktion, die später noch ausführlicher kommen wird:

hrdata %>%   
 mutate(env\_dissat = case\_when(  
 environment\_satisfaction == 1 ~ 4,  
 environment\_satisfaction == 2 ~ 3,  
 environment\_satisfaction == 3 ~ 2,  
 environment\_satisfaction == 4 ~ 1  
 ))

* + Die links-rechts-Orientierung ist anders rum, weil der erste Wert die Bedingung ist un der zweite der Ziel-Wert
  + “~” anstatt “=”
  + case\_when ist generischer und kann für viele andere Rekodierungsaufgaben benutzt werden. Daher ist sie mein Favorit, auch wenn sie etwas umständlicher ist (auch wenn man das eh mit copy&paste macht).

## Numerische Variablen 2: Kategorisieren von kontinuierlichen Variablen

* **Einfacher dummy-split**

hrdata %>%  
 mutate(age\_group = factor(age > 50,  
 labels = c("young", "old")))

Ab 50 soll eine Person „old“ sein, darunter „young“

* Ein Sonderfall ist der sog. Mediansplit. Wie der Name sagt, wird hier die Vairable am Median in zwei große Teile gesplitet. Das geht komfortabel, in dem man die median()-Funktion einbettet

hrdata %>%  
 mutate(age\_group = factor(age > median(age, na.rm = TRUE),   
 labels = c("young", "old")))

Das ist nru für deskriptive Analysen sinnvoll. Für statistische Verfahren sollten kontinuierliche Variablen nicht gesplittet werden

* **Multiple Splits**

hrdata %>%

mutate(age\_group = cut(age,   
 breaks = c(18, 25, 35, 45, 55),  
 labels = c("18-25", "26-35", "36-45", "46-55", ">56"),  
 include.lowest = TRUE))

* + Die breaks sind die Alters-Werte, mit der eine Kategorie *endet*
  + Die labels zeigen das ja (d.h. wie die Kateogien starten und enden)
  + Das Argument include.lowest führt dazu, dass 18 noch in die erste Kategorie eingeschlossen wird

## Numerische Variablen 3: Index bilden

* Composites sind Summen oder Mittlwerte anderer Variablen. Meist generiert man sie wenn man Variablen hat, die ein Konstrukt messen oder im Fall der Bildung von Indexvariablen

hrdata %>%   
 rowwise() %>%   
 mutate(satisfaction\_comp = mean(c(  
 environment\_satisfaction,   
 job\_satisfaction,   
 work\_life\_balance),  
 na.rm=TRUE)  
 )%>%  
 ungroup()

* + Sieht kompliziert aus—zentral ist
  + rowwise()
  + darunter der die Berechnung als Mittelwert (alternativ: sum() )
  + na.rm=TRUE muss rein, wenn es missings gibt
  + ungroup hebt die reihenweise Berechnung auf (sonst werden nach folgende Prozeduren auch reihenweise angeordnet)
* Man kann mit c\_across() auch über einen Bereich von Variablen mitteln, die nebeneinander stehen

data %>%   
 rowwise() %>%   
 mutate(Xsum = mean(c\_across(x1:x7), na.rm=TRUE)) %>%  
 ungroup()

(Anm. Generisches Beispiel)

## Kategoriale Variablen 1: Dummies bilden

hrdata %>%  
 mutate(gender\_dummy = as.numeric(gender == "Female"))

* + Das heißt, man benennt die 1er Kategorie und sonst nix. Andere Kategorien werden 0; missings bleiben missings
  + Wenn man mehrere Kategorien hat, muss man dies halt k-1 mal machen. Angenommen die Variable K hat 3 Kategorien (A,B , C) , dann folgen 2 dummies (z.B. B und C daraus), wenn man eine Regression machen möchte

dataset %>%  
 mutate(K\_b = as.numeric(K == "B"),  
 K\_c = as.numeric(K == "C")  
 )

(Generisches Beispiel)

## Kategoriale Variablen 2: Kategorien zusammenlegen

* Mit fct\_lump() aus dem **forcats** package kann man gering besetzt Faktorlevel zusammenlegen zu einer „others“-Kategorie. Sehr hilfreich bei barplots, die ansonsten eine Masse von uninteressanten Balken zeigen würden.
* Hiermal als Beipsiel die Augenfarbe der Charaktäre im Starwars data set (lauffähig ohne irgend ein package)

starwars %>%  
 count(skin\_color, sort = TRUE)

# A tibble: 31 x 2

skin\_color n

<chr> <int>

1 fair 17

2 light 11

3 dark 6

4 green 6

5 grey 6

6 pale 5

7 brown 4

8 blue 2

9 blue, grey 2

10 orange 2

# ... with 21 more rows

* Dies soll nun auf die breitesten 5 Kategorien plus eine Reste-Kategorie reduziert werden:

starwars %>%  
 mutate(skin\_color = fct\_lump(skin\_color, n = 5)) %>%  
 count(skin\_color, sort = TRUE)

* Das n-Argument legt die Anzahl der gewünschten Kategorien fest. Dazu kommt dann noch eine „others“-Kategorie

# A tibble: 6 x 2

skin\_color n

<fct> <int>

1 Other 41

2 fair 17

3 light 11

4 dark 6

5 green 6

6 grey 6

Wie man sieht, ist die „ohters“ Kategorie natürlich im ranking weiter oben, weil sie mehr Fälle umfasst

## Kategoriale Variablen 3: Kategorien neu ordnen

* Faktor-levels bzw. level können angeordet werden. Das ist bei einem Balkenplot wichtig, damit der plot nicht chaotisch ist
  + mit **fct\_reorder** entlang eines Statistik (wie count, mean) etc.
  + mit **fct\_relevel** manuell
* **Numerisches Anordnen** (noch mal mit der Hautfarbe der Starwars Figuren)

starwars %>%  
 count(skin\_color, sort = TRUE) %>%   
 mutate(skin\_color = fct\_reorder(skin\_color, n)) %>%   
 ggplot(aes(x=skin\_color, n)) +  
 geom\_col()+  
 coord\_flip()

Ein Bild, das Text, Screenshot, Diagramm enthält.

Automatisch generierte Beschreibung

* **Manuelles Anordnen**

dataset %>%   
 mutate(educ = fct\_relevel(educ, "Secondary school certificate",  
 "High School Degree",   
 "Certificate for university entrance",  
 "Bachelor",  
 "Master",  
 "Magister",  
 "Diplom",  
 "Promotion"))

* (Generisches Beispiel mit einer fiktiven Bildungsvariable)

## Kategoriale Variablen 4: Kategorien umbenennen / rekodieren

hrdata %>%  
 mutate(  
 Education = as.factor(Education),   
 Education = fct\_recode(Education,   
 "High" = "4",  
 "Med" = "3",  
 "Low" = "2",  
 "None" = "1")  
 )

Anm.

* + Die as.factor-Zeile ist hier ausnahmsweise nötig, weil Eduation numerisch ist.
  + Es ist egal, wie die Ausgangskategorien benannt sind (hier sind es Zahlen; sie können aber auch Begriffe sein.

## Bedingtes Berechnen mit ifelse() und case\_when()

* Es gibt zum einen die if\_else()-Funktion. So wie ich das sehe, ist die v.a. für simple Berechnungen optimal:

x <- c(-5:5, NA)

if\_else(x < 0, "negative", "positive", "missing")

Erstes Argument ist die Bedingung. Wenn die wahr ist (x <0), dann wird das zweite Argument eingesetzt, wenn nicht, die dritte. Wenn keine gelten, ist es missing

* Für komplexere könnte case\_when besser sein:

sim <- tibble(X = c(1,2,3,4,5,6), Y = c(0,1,1,1,0,1))

sim %>%   
 mutate(Z = case\_when(X == 1 ~ 0,   
 X >= 2 ~ 1,  
 TRUE ~ X))

"or" (|) und "not" (!=) gehen natürlich auch.

* "TRUE ~ X" ist "else". D.h. wenn es noch zig andere Wertkategorien gibt, bedeutet dies, dass R die originalen Werte der originalen X-Variable übernehmen soll. Das ist handlich, wenn man nur ein paar Kategorien ändern will.
* Alternative kann man über TRUE ~ einen alterantiven Wert für alle Restkategorien festlegen—z.B. für TRUE ~ 3 dazu, dass alle anderen Kategorien = 3 sind.
* Beispiel:

df = tibble(old = c("a","a","a", "b","b","b", "c","c","c","d","d","d"))

# Alte Kategorien werden übernommen

df %>%   
 mutate(new = case\_when(old == "a" ~ "A",  
 old == "b" ~ "B",  
 TRUE ~ old))

# Alte Kategorien werden zusammengelegt

df %>%   
 mutate(new = case\_when(old == "a" ~ "A",  
 old == "b" ~ "B",  
 TRUE ~ "Rest"))

* Achtung: Wenn die Original-Werte **missings** hat, werden sie ohne TRUE übernommen

df = tibble(x = c("a","b","b", NA))

df %>%   
 mutate(x\_new = case\_when(x == "a" ~ 0,  
 x == "b" ~ 1))

* + Der unter 3.4 angesprochene Weg ist der direktere Weg für die Dummy-Bildung
  + Wenn man sich für case\_when() entscheidet msüsen—wie oben die 1 und die 0 beide explizit gebildet werden, sonst werden auch die „a“-level = NA (anstatt 0)!

## Eigene Funtionen in mutate einbinden

* Hier auch gleich ein Beispiel dafür, wie man das für alle Variablen eines Typs macht. Hier am Beispiel Reskalierung auf den 0-1-Bereich

1. Funktion schreiben

rscl <- function(x, na.rm = FALSE) {  
 (x - min(x)) / (max(x) - min(x))   
 }

1. Auf die dann in mutate\_if verwiesen werden

feat\_scld <- features %>%   
 mutate\_if(is.numeric, rscl)

## Scientific notation ändern

Entweder durch options(scipen = 10) (was nicht immer geht) oder mit

mutate\_if(is.numeric, round, 5)

## Variablen zerlegen und zusammenfügen mit separate und unite

### Trennen von Variablen in mehrere separate Variablen

* **Trennung von Variablen, die aus abgetrennten Komponenten** besteht, in in separate Variablen, die die Komponenten einzeln enthalten, z.B. beim Datum: 2019-01-05 wird z.B. in 3 Variablen getrennt (2019, 01, 05)
  + Achtung: sep="-" bezieht sich auf das Original und fragt, durch was beide getrennt sind. So würde bei „Montag, 01.02.18“ der Separator zwischen Tag und Datum "," sein.
  + Hier am Beispiel des (automatisch geladenen) economics-Datensatzes

economics %>%   
 separate(date, c("year","month","day"), sep="-")

* + Achtung: R *ersetzt* die Datum durch die Aufsplittung
  + Daher besser die Originalvariable erst kopieren dann trennen:
* **Trennung von Variablen ohne Separator**
  + Was tun bei, wenn es keinen separator gibt? Angenommen, es gibt eine Varible die sowohol das Geschlecht als auch das Alter in einer Zelle enthält („m23“ für eine männliche, 23jährige Person). Lösung: Man trennt nach dem ersten Zeichen

data %<%  
 separate(gender\_age, sep=1)

* **Trennen von mehreren Angaben in einer Zelle**
  + Ausgangspnkt ist diese unschöne Datentabelle. Kern des Problems ist, dass die Variable visid\_id mehrere Ids enthält, die mit Kommata oder anderen Trennzeichen (|) abgetrennt sind

Ein Bild, das Text, Quittung, Schrift, weiß enthält.

Automatisch generierte Beschreibung

* + Diese Variable (bzw. ihre Inhalte) wird nun getrennt—Folge ist ein longformat-tibble, wo die IDs eines subjects untereinander stehen

data %>%  
 separate\_rows(visit\_id, measured, convert=TRUE)

Ein Bild, das Text, Quittung, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

* + Geht mit mehreren Variablen gleichzeitig
* Mit complete() kann man die Variablen-Einträge der fehlenden Zielen auf NA setzen

complete(subject\_id,   
 nesting(visit\_id)

Dadurch bekommen jetzt alle subjects eine visid\_id von 1, 2 und 3 und NA in der measured-Variable (d.h. R orientiert sch am Fall mit der maximalen Anzahl)

Ein Bild, das Text, Quittung, Schrift enthält.

Automatisch generierte Beschreibung

### Zusammenfügen mehrerer Variablen zu einer neuen

* **Unite** macht das Gegenteil von separate() und verbindet Variablen zu einer neuen

economics %>%   
 unite(date, c("year","month","day"), sep="-")

* + Hier wrüden die 3 Varialben year, month und day zu einer neuen (date) kombiniert; die Einträge werden jeweils mit dem Bindestricht getrenn

## Datensätze joinen

* Hinweis: Wenn man einfach Spalten in gleicher Reihenfolge zusammenfügen will, geht das am einfachsten mit bind\_cols()
* Beispiel:

data1 %>%>  
 bind\_cols(data2)

* Das ist im Grunde analog zu cbind()—generiert aber automatisch ein tibble

Ein Bild, das Text, Screenshot, Schrift, Kreis enthält.

Automatisch generierte Beschreibung

Hintergrund ist dieses Video: <https://www.youtube.com/watch?v=2W5-WrBEnEA>

Sie hat 3 Datensätze:

**Studentenfile**

Ein Bild, das Text, Schrift, Zahl, Screenshot enthält.

Automatisch generierte Beschreibung

**Location fille**

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

**Ethnicity file**

Ein Bild, das Text, Schrift, Zahl, Reihe enthält.

Automatisch generierte Beschreibung

Wichtig: Die Venn-Diagramme beziehen sicha auf Fälle, nicht auf Variablen

* **left join:** Aggregiert das linke file (Studi-Profile) mit dem ausgewählten rechten, also alle Studi-Daten mit der ethnicity-description. Left\_join bewirkt dass Studies ohne ethnicity erhalten bleiben; sie bekommen nur ein missing

left\_join(Student\_profile, Ethnicity\_Data, by „Ethnic\_Code“)

* **right\_join** passt sich an die rechte Zieldatei an. Hier ist die enthaltende Ziel-Variable und ihre Fälle massgebend. Ergebnis werden Zeilen sein, in denen alle Studenten-Infos missing sind, aber die einen Eintrag in der location haben

right\_join(Student\_Profile, Location\_Data, by = „Student\_ID“)

* **Inner\_join:** Enthält nur die Zeilen, die in beiden files vorhanden sind.

inner\_join(Student\_Profile, Location\_Data, by=“Student\_ID“)

* **Full\_join:** Aggregiert plump alles

Full\_join(Student\_Profile, Location\_Data, by=“Student\_ID“)

* **Anti\_join:** Zeigt die Komplemente beider Datensätze

Zwei Besonderheiten

1. Es braucht keine Schlüsselvariable (by=)
2. Die Reihenfolge ist wichtig:

|  |  |
| --- | --- |
| t1 | t2 |
| A |  |
| B | B |
| C | C |
| D | D |
|  | E |
|  | F |

anti\_join(t2, t1)

<chr>

1 E

2 F

🡪 Welche sind in t2, die nicht in t1 sind?

anti\_join(t1, t2)

<chr>

1 A

🡪 Welche sind in t1, die nicht in t2 sind?

## pivot\_longer() und pivot\_wider: Format der Daten ändern

### Pivot\_longer

* Pivot\_longer transformiert einen wideformat-Datensatz in einen longformat
* **Generierung von Beispiel-Daten**

df <- data.frame(  
 subj = c(1,2,3,4),  
 gender = c('f', 'f', 'm', 'm'),  
 t1 = c(4, 5, 7, 6),  
 t2 = c(6, 5, 9, 8)  
)

df

subj gender t1 t2

1 1 f 4 6

2 2 f 5 5

3 3 m 7 9

4 4 m 6 8

* **Pivotieren**

df %>%   
 pivot\_longer(3:4, names\_to = "time", values\_to = "value")

subj gender time dv

<dbl> <chr> <chr> <dbl>

1 1 f t1 4

2 1 f t2 6

3 2 f t1 5

4 2 f t2 5

5 3 m t1 7

6 3 m t2 9

7 4 m t1 6

8 4 m t2 8

* + Das erste Argument (3:4) benennt die Variablen, die umgebrochen werden sollen
  + names\_to wird die Spalte mit dem Variablen-Namen
  + values\_to wird die Benennung der Y-Variable

### Pivot\_wider

* Macht das Gegenteil und überführt einen longformat-Datensatz in wide format, so das ein Fall = eine Zeile wird
* Beispiel: Mittelwerte und Varianzen eines Cluster-Analyse; diese stehen für jeden Clsuter (Klasse) untereinander. Ziel soll sein, dass Mittelwert und Vairanz zwei nebeneinander stehende Variablen sind

Class Category Parameter Est

<int> <chr> <chr> <dbl>

1 1 Means int\_red 1.37

2 1 Means ext\_red 0.806

3 1 Means compens 0.669

4 1 Variances int\_red 0.419

5 1 Variances ext\_red 0.685

6 1 Variances compens 0.538

7 2 Means int\_red 1.02

8 2 Means ext\_red -0.338

9 2 Means compens -1.45

10 2 Variances int\_red 0.419

pivot\_wider(c(Class,Parameter), names\_from = Category, values\_from = Est)

* Grundprinzip:
  + Das erste Argument definiert die Variablen, die jeden Fall identifizieren. Hier müssen es 2 sein, da die „Parameter“ (=Geclusterte Variablen) genested in der jeweiligen Klasse sind. Jeder „Fall“ (Zeile) ist daher die Kombi einer Klasse und der jeweiligen Variable.
  + names\_from(): Geplante neue Spalten/Variablen. Hier kommen die aus der Category (means vs. variances)
  + values\_from(): Worau werden die Werte der neuen Spalten entnommen? → Est(imate)

Class Parameter Means Variances

<int> <chr> <dbl> <dbl>

1 1 int\_red 1.37 0.419

2 1 ext\_red 0.806 0.685

3 1 compens 0.669 0.538

4 2 int\_red 1.02 0.419

5 2 ext\_red -0.338 0.685

6 2 compens -1.45 0.538

7 3 int\_red 1.59 0.419

8 3 ext\_red 0.825 0.685

9 3 compens -1.46 0.538

10 4 int\_red -1.12 0.419

11 4 ext\_red -0.361 0.685

* **Komplexerer Fall:** Hier hat man nicht nur einen Estimate, sondern mehrere (estimate + se)

Class Category Parameter Estimate se p

<int> <chr> <chr> <dbl> <dbl> <dbl>

1 1 Means int\_red 1.37 0.120 1.78e-30

2 1 Means ext\_red 0.806 0.0782 6.07e-25

3 1 Means compens 0.669 0.0883 3.51e-14

4 1 Variances int\_red 0.419 0.0829 4.36e- 7

5 1 Variances ext\_red 0.685 0.0748 5.69e-20

...

pivot\_wider(c(Class,Parameter),

names\_from = Category,

values\_from = c(Estimate, se))

# A tibble: 12 × 6

Class Parameter Estimate\_Means Estimate\_Variances se\_Means se\_Variances

<int> <chr> <dbl> <dbl> <dbl> <dbl>

1 1 int\_red 1.37 0.419 0.120 0.0829

2 1 ext\_red 0.806 0.685 0.0782 0.0748

3 1 compens 0.669 0.538 0.0883 0.101

4 2 int\_red 1.02 0.419 0.221 0.0829

5 2 ext\_red -0.338 0.685 0.468 0.0748

6 2 compens -1.45 0.538 0.257 0.101

## Programming und Funktionen

**(Hinweis: Das ist für Fortgeschrittene)**

### Funktionen: Tutorials und background

* Einführungsvideo von Hefin Rhys: <https://www.youtube.com/watch?v=ffPeac3BigM>
* Cran-Einführung: <https://cran.r-project.org/doc/manuals/R-intro.html#Writing-your-own-functions>
* Chapter in einem tidyerse online book: <https://b-rodrigues.github.io/modern_R/defining-your-own-functions.html>
* **Zentrale Keywords bzgl. Funktionen:**
  + **Essential arguments:** Dies sind die Argumente in der Klammer (einzelne Begriffe ohne Komma). *Positional matching:* Z.B. (data….): An erster Stelle müssen die Daten kommen. Daher braucht man nicht data = df schreiben sondern es reicht df.
  + **Default arguments:** Haben ein **Komma** und einen Wert. Bedeutet, wenn man das ignoriert, wird automatisch dieser Wert genommen, man kann ihn aber optional ändern
  + **Unnamed arguments** sind die 3 Punkte "**. . .**". Diese beziehen sich auf ein default argument in einer der benutzten (Sub)funktionen im body der Funktion. Z.B. wenn eine eigene Funktion u.a. ein Histogramm generiert, dass als default argument „breaks“ hat. Dann kann man mit dem unnamed Argument dieses default argument quasi auf die eigene Funktionsebene „heben“.

MixedStuff <- function(data, x, ...) {

hist(data, x, ...)

}

Im call kann kann „breaks=50“ direkt angesprochen werden, obwohl es eigentlich ein Argument der Sub-Funktion ist (z.B. MixedStuff(data, var1, breaks=50) )

Wenn man Sub-Funktionen benutzt, besser das package mit nennen mittels "::"

* + **Logical arguments:** Sind default arguments, z.B. expo(x, power =1, hist=FALSE)

Das ist ein Beispiel aus Rhys‘ Video. Der default ist, dass das Histogramm (das im body als Funktion kommt, nicht generiert wird. Das kann man aber durch TRUE quasi anschalten. Achtung: Ein logisches Argument macht eine *if-else-Bedingungsprüfung* im body notwendig (wenn hist=TRUE dann histogram(), wenn else, dann was anders/kein histogram). Siehe unten ein Beispiel.

*Alternative* ist ein unnamed Argument dass dann TRUE oder FALSE sein kann (hier remove\_na)

data = c(1, 8, 1, NA, 8)

my\_func <- function(data, func, remove\_na){

mean(data, na.rm = remove\_na)

}

my\_func(data, func, remove\_na=TRUE)

🡪 In der Funktion wurde die Bedingungsprüfung bei na.rm selbst zum Argument

* **Tidy evaluation: Variablen in tibbles**
  + **Curly-Curly:** Wenn man eine Funktion auf Variablen anwendet, sucht R im environment. Sind sie aber in den tibble, müssen sie **enquoted** werden und der Variablen in der Sub-Funktion (hier summarise) durch doppel-curly brackets eingebunden werden (ausgesprochen „curly-curly). Das wird **tidy evaluation** genannt.

simple\_function <- function(dataset, variable){

dataset %>%

summarise(mean( {{variable}} ))

}

simple\_function(mtcars, mpg)

* + **Filter** funktioniert mittels Doppelklammern

simple\_function <- function(dataset, filter\_variable, value, variable){

dataset %>%

filter(({{filter\_variable}}) == value) %>%

summarise(mean = mean({{variable}}))

}

simple\_function(mtcars, am, 1, mpg)

* Leider konnte ich nicht rausfinden, wie das läuft, wenn die zu filternde Variable ein Faktor war auf dessen faktor level ich Bezug nehmen wollte (Masem-Beispiel): Also musste es doch base R sein:

biv\_meta <-function(mdata, model, value) {

es\_data <- mdata[mdata$model== value, ]

* Im call konnte ich dann biv\_meta(mdata, model, "gender.attitude") schreiben
* **Statistiken multipler Variablen:** Will man eine Funktion auf mehrere Variablen anwenden will, geht das mit across() ([Quelle](https://www.bryanshalloway.com/2020/06/25/using-across-to-build-functions-with-dplyr-with-notes-on-legacy-approaches/)):

sum\_vars <- function(data, vars){

summarise(data, across({{vars}}, list(sum = sum, mean = mean)))

}

* + Function call ist dann

sum\_vars(mtcars, c(mpg, disp))

* + Wie man sieht, braucht man allerdings c().

mpg\_sum mpg\_mean disp\_sum disp\_mean

1 642.9 20.09062 7383.1 230.7219

* + Selektieren multipler Variablen geht auch so

data\_select <- function(mdata, vars){

mdata %>%

select( {{vars}})

}

data\_select(mdata, c(studyID, n, pbc\_type, authors))

# A tibble: 733 x 4

studyID n pbc\_type authors

<dbl> <dbl> <fct> <fct>

1 1 345 NA Gupta et al. (2009)

2 2 308 SEF Carr & Sequeira (2007)

3 2 308 NA Carr & Sequeira (2007)

4 2 308 SEF Carr & Sequeira (2007)

5 4 4292 SEF Wilson et al. (2007)

...

### Pragmatischer workflow bei der Entwicklung einer Funktion

Eigene Funktionen kann man aufbauen, in dem man

1. erst mal den relevanten dplyR code schreibt
2. den code dann in die Funktion schreibt und
3. abschließend den code abstrahiert und relevante Parameter in die Funktion als Argumente schreibt

Hier macht das jemand um einen ggraph eines word cloud zu programmieren

<https://www.youtube.com/watch?v=ae_XVhjHd_o&t=44s> (15:45)

Hier mal an simplen deskriptiven Stats von 4 Variablen

x1 = rnorm(100); x2 = rnorm(100);x3 = rnorm(100);y = .4\*x1 + .5\*x2 + .8\*x3 + rnorm(100)

data = tibble(x1,x2,x3,y)

**#Step 1: Base code generieren**

data %>%

summarise(mean\_x1 = mean(x1), sd\_x1 = sd(x1),

mean\_y = mean(y), sd\_y = sd(y))

Hier hat man noch konkrete Variablen (x1 und y). (Ich lass na.rm=TRUE mal der Übersicht halber weg)

**#Step 2: Funktions-Skelett anlegen und base code stumpf reinkopieren**

base\_stats = function() {

data %>%

summarise(mean\_x1 = mean(x1), sd\_x1 = sd(x1),

mean\_y = mean(y), sd\_y = sd(y))

}

#Step 3: Abstrahieren (inkl. curly brackets)

base\_stats = function(dataframe, x, y) {

dataframe %>%

summarise(mean\_x = mean({{x}}, na.rm=TRUE), sd\_x = sd({{x}}),

mean\_y = mean({{y}}, na.rm=TRUE), sd\_y = sd({{y}}))

}

Wie man sieht wurden x1 und y durch allgemeine slot-Namen ersetzt (dataframe, x, y)

Den kann man dann laufen lassen mit beliebigen Variablen in data

base\_stats(data, x2, x1)

### Potentielle cases

* **Kombination von if-else-Bedingungsprüfungen**

my\_function <- function(argument1, argument2, method = "foo"){

x <- argument1 + argument2

if(method == "foo"){

1/sqrt(x)

}

else if (method == "bar"){

"this is a string"

}

else { "this is not defined"

}

}

🡪 Wenn die method = foo ist, rechne 1/sqrt(x). Wenn sie stattdessen (elfe if) „bar“ ist, dann gib den string aus. Wenn nix zutrifft (else = ansonsten), dann gibt „not defined“ aus

my\_function(2,5, method="xy")

[1] "this is not defined"

* **For-loops in Funktionen (siehe unten einfacheres Beispiel für ne loop):** hier über die Fibronacci-Zahlen

Zentral ist die generierung der temporären temp-Variable die zu Beginn jeder loop überschriebne wird (während a und b erhalten bleiben)

my\_fibo <- function(n){

a <- 0

b <- 1

for (i in 1:n){

temp <- b

b <- a

a <- a + temp

}

a

}

🡪 Auch genannt „iterative Funktion)

**Exkurs:** Einfache for-loop (Monte-Carlo Simulation einer Korrelation

V = 1:100000 #Laufindex, der durchgenudelt wird

result <- list() #Zielliste

# For-loop laufen lassen

for (i in V){

x = rnorm(28)

y = .5\*x + rnorm(28,0,sqrt(1-.5^2))

rxy = cor(x,y)

result[[i]] <- rxy

}

#Das ganze in ein genestetes Tibble

(data <- tibble(y = result))

#...und unnesten

(data <- data %>%

unnest(y))

* **Labeln von Objekten, die in einer Funktion erzeugt wurden**
  + Geht mit as\_label.
  + Beispiel ist die simple Mittelwertsfunktion, die bei der Einführung von enquo und bang-bang generiert wurde. Jetzt soll jetzt der output "mean\_..." sich nach der jeweiligen Variable richten

simple\_function <- function(dataset, mean\_col){

mean\_col <- enquo(mean\_col)

mean\_name <- paste0("mean\_", as\_label(mean\_col))

dataset %>%

summarise(!!(mean\_name) := mean((!!mean\_col)))

}

* + Man beachte den assignment operator und die beiden bang-bangs

simple\_function(mtcars, am)

mean\_am

1 0.40625

Wie man sieht ist die Variable "am" Teil des Objektnamens. Wenn man da gar nix pasted, wird alles als Name der simple Variablenname benutzt (der für mean\_col später verwendet wird).

* **In der Funktion auf extrahierte Elemente eines outputs verweisen** (#Extraktion, #extrahieren)
  + Erst muss man rausfinden, wie das Element heißt. Dies geht durch str(model) wo model = model object. Ist die Listenstruktur zu komplex, kann man das durch str(model, max.level=2) veränderen
  + ACHTUNG: Manchmal ist es sinnvoller den str()-Befehl auf die summary des Models anzuwenden (v.a. wenn man Teile davon benutzen möchte
  + Auch names(LinReg)kann helfen bei der Identifikation
  + Manchmal ist es simpel—manchmal etwas rumgezocke, bis man den Parameter hat. Dann wird er einfach durch $ an das Objekt angehängt. Wenn der Zielparameter noch eine Ebene drunter ist, dann eben zweimal hintereinander (z.B metafor\_temp$mf.g$inner). Hier ein Beipiel aus einer MASEM-Funktion

b = tibble(metafor\_temp$b[,1]) %>%

rename(estimate = `metafor\_temp$b[, 1]`) %>%

round(., 2)

* + Wie man sieht musste ich das Teil umbenennen, weil durmh die Umwandlung in ein tibble der Parameter diesen hässliche column name bekommt
  + Mittels print() wird das Ergebnis zwischendurch ausgegegen (return scheint zu bewirken, dass danach nix mehr ausgegeben wird)

### Map und anonyme Funktionen

#### Map vs. base R

* Der Begriff **functional programming** heißt erstmal lediglich, dass man Funktionen schreibt. Dazu bietet base R Sub-Funktionen wie die **while-loop** („führe so lange eine Operation aus, bis ein Kriterium erreicht ist“), die **for-loop** („Führe für jedes Dlement von i eine Operation aus“) oder rekursive Funktionen, die auf einen in der Funktion generierten Wert Bezug nehmen. Letztere sind rechenaufwändiger als loops (ALLES NOCH MAL CHECKEN)
* Die Stuktur des map-Befehls ist map(.x, .f, ...)
  + **.x** ist ein Laufindex (Elemente eines Index, Listenelemente oder Variablen im tibble)
  + **.f** ist eine Funktion: Das kann eine anonyme Funktion sein, oder eine Standardfunktion. Wickhkam bezeichnet die Funktion als ein Rezept, dass auf jeden Eintrag in .x angewendet wird (eine anonyme Funktion erkennt man an dem „(x)“ –also function(x){}
  + **"..."** ist ein anonymes Argument (nicht dasselbe wie eine anonyme Funktion), mit der man auf Argumente von .f zurückgreifen kann.
* Map ist erstmal ein Prinzip, dass man auch in den Funktionen der \*apply-Familie hat!
* Das purrr package bringt einige neue Funktionen. Eine davon ist die map\*()-family, die die \*apply Funktionen ersetzt (beachte die zwei verschiedenenen Bedeutungen)
* Mittels map() cycled man durch jedes Element des Tibbles/Vektors, wendet dort entweder eine anonyme Funktion (.f) oder eine andere Funktion (z.B.mean etc.) an und gibt als Ergebnis eine Liste aus. Nimmt man stattdessen map\_df() wird ein tibble ausgegeben. Im Grunde macht sie damit den selben job wie die for-loop. Vorteil von map() ist eine größere Einfachheit bei der Schreibweise und auch, dass man den results-Vektor (in den das Ergebnis gechrieben wird) nicht vordefinieren muss. Verglichen mist lapply ist sie gleich komplex, man kann aber das Ausgabeformat frei bestimmen.
* Beispiel: Hier wird eine anonyme Funktion angewendet, die eine Variable zentriert

data = tibble(z = rnorm(10))

map(data, function(x){

x\_centered = x - mean(x)

round(x\_centered, 2)

})

$z

[1] 1.71 -1.77 1.62 0.80 -0.21 -0.94 -1.62 -0.21 -0.48 1.09

Sieht aus wie ein Vektor ist aber eine Liste. Wählt man stattdessen map\_df() bekommt man ein tibble

# A tibble: 10 x 1

z

<dbl>

1 1.71

2 -1.77

3 1.62

4 0.8

...

* **Vergleich map mit for-loops und lapply**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **for-loop** | **map()** | **lapply()** |
| Daten / Vorbereitung | results = numeric(10) | V = c(1:10) | V = c(1:10) |
| Funktion | for(i in 1:10) {  results[i] = i\*2  }  print(results) | map(V, function(x){  x\*2  })  Alternative  map(V, ~{. \*2}) | lapply(V, function(x){  x\*2  })  Identisch! |

* + Wie man sieht, benötigt die loop einen vordefinierten results-Vektor (map nicht). Andererseits benötigt map eine vorliegende Datenstruktur (hier V) und sei es nur als Laufindex (wie im Beispiel). In den meisten Fällen liegt die aber sowieso vor, weil man map ja auf Daten anwenden möchte
  + In der Roh-Form gibt map eine Liste aus. Die Wahl eines anderen Formats (z.B. tibble mittels map\_df erfordert, dass V einen header hat.
  + In map kann man V pipen (V %>% function(x).... oder V %>% map(~{. \*2}) )
  + Weiterhin sieht man dass map und lapply identisch sind. Die 2 Vorteile von map sind 1) die formula-Schreibwiese (~) und 2) die Flexibilität beim output (lapply kann nur Listen)
* Wendet man map auf nicht auf einen Vektor sondern eine multivariate Datenstruktur an (dataframe, matrix, Liste), führt map die Funktion für jede Variable / Listenelement aus:

my\_list = list(matrix(1:9,ncol=3), matrix(11:25,ncol=3))

map(my\_list, function(x){x \* 2})

[[1]]

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

[1,] 2 8 14

[2,] 4 10 16

[3,] 6 12 18

[[2]]

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

[1,] 22 32 42

[2,] 24 34 44

...

#### Formel-Schreibweise

* Während man mit map diestandardgemäße Funktionsschreibweise benutzen kann (d.h. function(x) {}, gibt es eine sparsamere Alternative. Diese **ersetzt**
  + function(x) durch eine **Formel**, die mit ~ eingeleitet wird und
  + "x" durch "."

Hier der direkte Vergleich

map(data, function(x){ 1/sqrt(x)})

map(data, ~{ 1/sqrt(**.**)})

Oder angewendet auf das Listenbeispiel („multipliziere jedes Element mit 2“):

map(my\_list, ~{. \* 2}) anstelle map(my\_list, function(x){x \* 2})

#### Anwendung von Standard-Funktionen

* Das schicke ist, dass man map auf übliche Funktionen (wie mean etc. und deren Argumente anwenden kann und so die Funktion auf alle Variablen des Datensatzes anwenden kann.

map\_df(data, mean, na.rm=TRUE)

x1 x2 x3 y

<dbl> <dbl> <dbl> <dbl>

1 -0.0778 -0.00283 0.114 0.0153

* Das ganze geht natürlich auch, wenn man data piped: data %>% map\_df(mean, na.rm=TRUE)
* Wie man sieht, geht das ohne jegliche Formelschreibweise
* Anwendung auf das Listenbeispiel (dabei wird über alle Elemente eines Listenelements gemittelt):

map(my\_list, mean, na.rm=TRUE)

[[1]]

[1] 5

[[2]]

[1] 18

#### Weiteres

* **map\_if()** führt eine Funktion aus, wenn eine Bedingung erfüllt ist [ist das eine Alternative für ifelse?]

x = round(rnorm(10),2)

z = c("a", "a","a","b","b","a","b","b","a","b")

data = tibble(x,z)

🡪 Wenn die Daten numerisch sind, dann wandel sie in character um

map\_if(data, is.numeric, as.character)

$x

[1] "0.98" "1.16" "1.24" "0.92" "0.41" "1.15" "-1.1”...

$z

[1] "a" "a" "a" "b" "b" "a" "b" "b" "a" "b"

Wenn der input ein tibble ist, geht er durch jede Variable (was für stats sehr hilfreich ist). Das lässt sich durch select() natürlich eingrenzen.

Eine „else“-Komponente lässt sich auch bestimmen

map\_if(data, is.numeric, as.character, .else = as.integer)

(in dem o.g. Fall jetzt unsinnnig)

* **map2()** verarbeitet 2 Inputs für die Funktion

map2(data3$z, data3$w, function(x, y){

(x\*\*2)/y

})

(Keine Ahnung, was ein Anwendungsfall sein könnte

* **Einlesen multipler dataframes:** Das wird im „the joy of functional programming erklärt): wenn man zig datafiles in einer directory hat ist der Pfad ein Objekt, über dessen Elemente (=files) man die files einlesen kann

*Achtung:* Das ganze fängt an mit dem unzippen der files. Liegen die schon unzipped vor, kannes direkt mit dem Einlesen beginnen.

Ich hab als letzten Punkt eine Kurversion aus lediglich 3 commands, die funktionieren müsste!

1. **Liste mit filenamen anlegen.** Grundlage war in dem Fall ein Ordner, der 4 separate Datensätze als csv enthielt. Die hatten alle einen Stammnamen („Test Nummer 1.csv“, „Test Nummer 2.csv“ etc. Jeder dieser Datensätze hatte 4 Variablen (gender, stress, workload, depression), was in der Realität nicht der Fall sein muss. Daher muss man das vor dem Einlesen checken. Das schicke daran: Man muss das nicht manuell machen.  
     
   Dazu muss der Pfad angegeben werden als auch ein charakteristiches Element des Namens (hier war das der Begriff „Nummer“).

header <- fs::dir\_ls("C:\\Users\\Holger\\Dropbox\\test", regexp="Nummer")

[1] "gender; stress; workload; depression"

[2] "gender; stress; workload; depression"

[3] "gender; stress; workload; depression"

[4] "gender; stress; workload; depression"

Wie man sieht, enthalten alle Datensätze die selben Variablen.

1. **Files einlesen**

data <- vroom::vroom(header, id="path")

1. **ID aus den filenamen extrahieren**

Dies geht natürlich nur, wenn der filenamen eine Zahl enthält. Es muss halt irgendwas idiosynkratisches sein.

data <- data %>% extract(path, "id", "(\\d{1})")

# A tibble: 12 x 5

id gender stress workload depression

<chr> <dbl> <dbl> <dbl> <dbl>

1 1 1 0 2 0

2 1 2 1 1 2

3 1 3 1 2 2

4 2 1 0 2 0

5 2 2 1 1 2

6 2 3 1 2 2

7 3 1 0 2 0

8 3 2 1 1 2

* **Unzippen von files**

Wenn die files gezippt sind, kann man mit einem Schlag alle files unzippen und dann die o.g. Schritte ausführen

1. Generieren einer Liste, die die filenamen enthält

paths <- fs::dir\_ls("C:\\Users\\Holger\\Desktop\\test", glob= "\*.zip")

1. Auf einen Schlag alle unzippen

map(paths, ~ unzip(.x, exdir="test"))

„exdir“ ist der Zielfolder. Ich hatte den test-folder auf dem Desktop. Lässt man exdir komplett weg, unzippt er die auf dem Desktop. Mit exdir macht er das im test-Ordner

Anschließend können die Schritte 1-3 aus dem o.g. Einlesenteil durchgeführt werden

# Explorative Datenanalyse

## Globale Analysen

Globale Analysen bestehen aus Funktionen, die über eine größere Anzahl Variablen grundlegende Verteilungsstatistiken berichten. Sie sind daher effizienter als sie mit summarise() einzeln anzufordern (letzteres macht man eher spontan bei Interesse bzgl. einer Variable)

* **Summary table aller Variablen** mittels der describe() Funktion aus dem psych-Paket

hrdata %>%   
 psych::describe()

Lässt sich noch über as\_tibble zurechtschneiden nach Belieben. Wichtig: das rownames-Argument (sonst sind die Variablennamen weg)

hrdata %>%   
 psych::describe() %>%   
 as\_tibble(rownames = "variable") %>%   
 select(variable, n, mean, sd, min, max)

# A tibble: 29 × 6

variable n mean sd min max

<chr> <dbl> <dbl> <dbl> <dbl> <dbl>

1 Age 1470 36.9 9.14 18 60

2 Attrition\* 1470 1.16 0.368 1 2

3 DailyRate 1470 802. 404. 102 1499

4 DistanceFromHome 1470 9.19 8.11 1 29

5 Education 1470 2.91 1.02 1 5

6 EmployeeCount 1470 1 0 1 1

7 EmployeeNumber 1470 1025. 602. 1 2068

8 EnvironmentSatisfaction 1470 2.72 1.09 1 4

9 Gender\* 1470 1.6 0.490 1 2

10 HourlyRate 1470 65.9 20.3 30 100

# ℹ 19 more rows

* **Übersicht über Datensatz und Berücksichtigung aller Klassen**

data %>%   
 skimr::skim(mtcars)

* + Kann man mit Vorschaltung von select\_if(is.numeric) auch nur für bestimmte Klassen anzeigen
  + oder mit select() auf bestimmte Variablen

── Data Summary ────────────────────────

Values

Name Piped data

Number of rows 1470

Number of columns 29

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Column type frequency:

character 3

numeric 26

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Group variables None

── Variable type: character ───────────────────────────────────────────────

skim\_variable n\_missing complete\_rate min max empty n\_unique whitespace

1 Attrition 0 1 2 3 0 2 0

2 Gender 0 1 4 6 0 2 0

3 OverTime 0 1 2 3 0 2 0

── Variable type: numeric ─────────────────────────────────────────────────

skim\_variable n\_missing complete\_rate mean sd p0 p25 p50 p75 p100 hist

1 Age 0 1 36.9 9.14 18 30 36 43 60 ▂▇▇▃▂

2 DailyRate 0 1 802. 404. 102 465 802 1157 1499 ▇▇▇▇▇

3 DistanceFromHome 0 1 9.19 8.11 1 2 7 14 29 ▇▅▂▂▂

4 Education 0 1 2.91 1.02 1 2 3 4 5 ▂▃▇▆▁

5 EmployeeCount 0 1 1 0 1 1 1 1 1 ▁▁▇▁▁

6 EmployeeNumber 0 1 1025. 602. 1 491. 1020. 1556. 2068 ▇▇▇▇▇

7 EnvironmentSatisfaction 0 1 2.72 1.09 1 2 3 4 4 ▅▅▁▇▇

8 HourlyRate 0 1 65.9 20.3 30 48 66 83.8 100 ▇▇▇▇▇

9 JobInvolvement 0 1 2.73 0.712 1 2 3 3 4 ▁▃▁▇▁

10 JobLevel 0 1 2.06 1.11 1 1 2 3 5 ▇▇▃▂▁

11 JobSatisfaction 0 1 2.73 1.10 1 2 3 4 4 ▅▅▁▇▇

* Darstellung der Variablenklassen

str(hrdata)

$ Age : num [1:1470] 41 49 37 33 27 32 59 30 38 36 ...

$ Attrition : chr [1:1470] "Yes" "No" "Yes" "No" ...

$ DailyRate : num [1:1470] 1102 279 1373 1392 591 ...

$ DistanceFromHome : num [1:1470] 1 8 2 3 2 2 3 24 23 27 ...

$ Education : num [1:1470] 2 1 2 4 1 2 3 1 3 3 ...

$ EmployeeCount : num [1:1470] 1 1 1 1 1 1 1 1 1 1 ...

$ EmployeeNumber : num [1:1470] 1 2 4 5 7 8 10 11 12 13 ...

$ EnvironmentSatisfaction : num [1:1470] 2 3 4 4 1 4 3 4 4 3 ...

glimpse(hrdata)

Rows: 1,470

Columns: 29

$ Age <dbl> 41, 49, 37, 33, 27, 32, 59, 30, 38, 36, 35, 29, …

$ Attrition <chr> "Yes", "No", "Yes", "No", "No", "No", "No", "No"…

$ DailyRate <dbl> 1102, 279, 1373, 1392, 591, 1005, 1324, 1358, 21…

$ DistanceFromHome <dbl> 1, 8, 2, 3, 2, 2, 3, 24, 23, 27, 16, 15, 26, 19,…

$ Education <dbl> 2, 1, 2, 4, 1, 2, 3, 1, 3, 3, 3, 2, 1, 2, 3, 4, …

$ EmployeeCount <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …

$ EmployeeNumber <dbl> 1, 2, 4, 5, 7, 8, 10, 11, 12, 13, 14, 15, 16, 18…

$ EnvironmentSatisfaction <dbl> 2, 3, 4, 4, 1, 4, 3, 4, 4, 3, 1, 4, 1, 2, 3, 2, …

$ Gender <chr> "Female", "Male", "Male", "Female", "Male", "Mal…

Vorteil: Es werden soviele tatsächliche Werte gezeigt, wie auf den Bildschirm passen

## Häufigkeiten (uni- und bivariat)

### Einfache Häufigkeitstabelle

hrdata %>%   
 count(education, sort= TRUE)

Education n

<dbl> <int>

1 3 572

2 4 398

3 2 282

4 1 170

5 5 48 7

Das name-Argument ist v.a. dann extrem hilfreich, wenn man dies als neuen, aggregierten, Datensatz weiterverwendet.

### Häufigkeiten in der count-Funktion aggregieren

Man kann das Ausmaß der Aggregierung innerhalb des count-Befehls vergröbern, was v.a. bei vielen differenzierten Werten sinnvoll ist. Beispiel monthly\_income:

hrdata %>%   
 count(monthly\_income)

monthly\_income n

<dbl> <int>

1 1009 1

2 1051 1

3 1052 1

4 1081 1

5 1091 1

6 1102 1

7 1118 1

8 1129 1

...

Wie man sieht sehr feingranuliert und nicht hilfreich. So werden die Häufigkeiten in 1000er-

Schritten aggregiert:

hrdata %>%   
 count(monthly\_income = 1000\*(monthly\_income %/% 1000))

MonthlyIncome n

<dbl> <int>

1 1000 33

2 2000 362

3 3000 147

4 4000 207

5 5000 165

6 6000 121

7 7000 54

...

### Orginal-Datensatz bewahren mit add\_count()

Count() liefert Häufigkeiten. Eine andere Sichtweise ist, dass es einen neuen Datensatz liefert mit aggregierten Anzahlen. Alle fallbezogenen Informationen/Werte sind dann weg (weil aggregiert.

* + **Mit count():**

hrdata %>%  
 count(education)

Education n

<dbl> <int>

1 1 170

2 2 282

3 3 572

4 4 398

5 5 48

* + Dies will man oft nicht—stattdessen möche man die Häufigkeit einer Variable einfach als Information den Rohdaten hinzufügen. Dies geht mit add\_count()

hrdata %>%   
 add\_count(education) %>%   
 select(education, n, everything()) #Nur zur Veranschaulichung

# A tibble: 1,470 × 30

**Education n** Age Attrition DailyRate DistanceFromHome EmployeeCount

**<dbl> <int>** <dbl> <chr> <dbl> <dbl> <dbl>

1 2 **282** 41 Yes 1102 1 1

2 1 **170** 49 No 279 8 1

3 2 **282** 37 Yes 1373 2 1

4 4 **398** 33 No 1392 3 1

5 1 **170** 27 No 591 2 1

6 2 **282** 32 No 1005 2 1

7 3 **572** 59 No 1324 3 1

8 1 **170** 30 No 1358 24 1

9 3 **572** 38 No 216 23 1

10 3 **572** 36 No 1299 27 1

→ Wie zu sehen wird die Häufigkeit des Wertes (z.B. Education = 2 → 282) hinter jeden Fall eingefügt.

### Bivariate Häufigkeits- oder Kreuztabelle

* **Einfache Kreuzabelle (nur mit Häufigkeiten**

Hier: Kreuzung von Education und Geschlecht

hrdata %>%   
 count(education, gender) %>%   
 spread(gender, n)

* + Da count eine Auflistung aller Kombinationen lifert, ist die spread-Funktion sehr hilfreich, da sie die Kreuzung darstellt. Die in spread zuerst genannte Variable bestimmt die Spalte:

Education Female Male

<dbl> <int> <int>

1 1 60 110

2 2 117 165

3 3 235 337

4 4 154 244

5 5 22 26

Es empfiehlt sich, die Variable mit der geringeren Anzahl von Kategorien als Spaltenvariable zu wählen.

* **Prozentwerte in Kreuztabellen**

Will man Prozentwerte haben (Spalten oder Zeilen) kann man das auf 2 Arten machen

* + sehr einfach mit der tabyl-Funktion aus dem janitor package
  + manuell mittels count und mutate. Das ist vorzuziehen, wenn man einen plot anhängen will zur Visualisiserung

Schicker (inkl. Prozentwerten) geht es mit der tabyl()-Funktion aus dem janitor package ([hier](https://cran.r-project.org/web/packages/janitor/vignettes/tabyls.html) ein kleines tutorial)

1. **Einfach mit tabyl**

library(janitor)

hrdata %>%   
 tabyl(education, gender) %>%   
 adorn\_percentages("col") %>% #Berechnet Spaltenprozente  
 adorn\_pct\_formatting(digits = 2) %>% #Formatierung mit 2 Dezimalstellen  
 adorn\_ns() #Hinzufügen der einfachen N‘s

Education Female Male

1 10.20% (60) 12.47% (110)

2 19.90% (117) 18.71% (165)

3 39.97% (235) 38.21% (337)

4 26.19% (154) 27.66% (244)

5 3.74% (22) 2.95% (26)

Hinweis: “adorn” bedeutet “verzieren”

1. **Manuell über count & Barplot**

hrdata %>%   
 count(education, gender) %>%   
 group\_by(education) %>%   
 mutate(percent = n / sum(n) )

cyl wird vorher noch zum Faktor gemacht (braucht man nicht, sieht aber im plot besser aus)

am cyl n percent

<dbl> <fct> <int> <dbl>

1 0 4 3 0.158

2 0 6 4 0.211

3 0 8 12 0.632

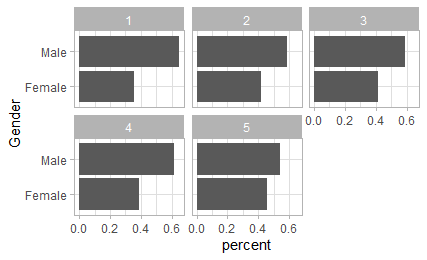
4 1 4 8 0.615

5 1 6 3 0.231

6 1 8 2 0.154

Für eine Visualisierung würde man einfach an den obere pipeline die folgende anschließen

%>%   
 ggplot(aes(gender, percent)) + #Auswahl Variablen aus der Tabelle  
 geom\_col() + #Barplot  
 facet\_wrap(~education) + #Aufsplittung in separate Fenster  
 coord\_flip() + #Balken um 90° drehen (→ Horizontal)  
 theme\_light() #Anderes layout (optional)



Wichig: Diese Variable in facet\_wrap muss dem vorherigen group-Befehl entsprechen, weil die Prozentwerte ja für die Gruppe gebildet werden (also wie ist das %Verhältnis von Männern und Frauen in einen Bildungstyp?)

## Univariate deskriptive Statistiken

Zentrale Funktion ist summarise()

dataset %>%   
 summarise(mean(X, na.rm=TRUE))

* Anstatt mean geht auch
  + sd()
  + var()
  + min()
  + max()
  + quantile(x, .25) oder ein anderes Perzentil
  + sum()
* Es macht Sinn, dem „Mikrotibble“ (das nur den Mean enthält) eine Spaltenüberschriftzuweisen:

hrdata %>%   
 summarise(distance\_mean = mean(distance\_from\_home, na.rm = TRUE))

distance\_mean

<dbl>

9.19

* In die summarise()-Funktion kann man natürlich weitere Statistiken und / oder weitere Variablen) einfügen. Aber einer bestimmten ANazhl lohnt sich aber psych::describe() mit vorheriger select() Funktion

Alternative ist summarise\_at über einen Bereich von Variablen

hrdata %>%   
 summarise\_at(vars(distance\_from\_home:environment\_satisfaction), mean,   
 na.rm = TRUE)

* + Aufpassen auf die vars() Subfunktion
  + Anstatt eines Bereichs von Variablen geht auch eine Auflistung (x, y, z)
* Schließlich kann man mit summarise\_if() eine Bedingungsprüfung machen, z.B. nach dem Modus:

hrdata %>%   
 summarize\_if(is.numeric, mean, na.rm=TRUE) %>%

→ Berechnet für alle numerischen Variblen den Mittelwert

## Kreuztablen mit Mittelwerten

* **Gruppenvergleich I: Ein Gruppierungsfaktor**

hrdata %>%   
 group\_by(gender) %>%   
 summarise(mean\_inc = mean(monthly\_income, na.rm = TRUE) )

* **Gruppenvergleich II: Zwei gekreuzte Gruppierungsfakoren**

hrdata %>%   
 group\_by(education, gender) %>%   
 summarise(mean\_inc = mean(monthly\_income, na.rm = TRUE) ) %>%   
 spread(gender, mean\_inc)

Education Female Male

<dbl> <dbl> <dbl>

1 1 5781. 5564.

2 2 6169. 6267.

3 3 6811. 6313.

4 4 6878. 6804.

5 5 9241. 7463.

* + Es wird Education und Gender gekreuzt
  + spread() wandelt die Liste wieder in eine Tabelle um (Achtung, ist der Zelleninhalt ist nun mean\_inc (ncht n wie bei der Häufigkeitstabelle)

## Korrelationen

* BaseR Variante. Simpel, schnell, aber mit zig Dezimalstellen

hrdata %>%   
 select(age, daily\_rate, distance\_from\_home) %>%   
 cor(use = "pair")

* + Wenn man einen Faktor drin hat, muss man den vorher mit mutate rekodieren. Das kanna auch „on the fly“ in der pipeline sind

hrdata %>%   
 select(age, daily\_rate, distance\_from\_home, attrition) %>%   
 mutate(  
 attrition = as.factor(attrition),  
 attrition = as.numeric(attrition)) %>%   
 cor(use = "pair")

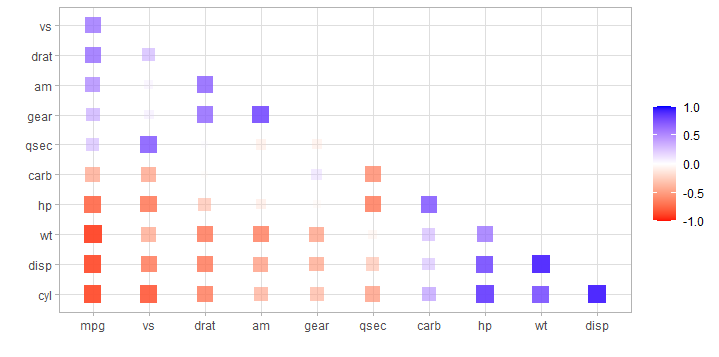
* + Hier war Attrition ein character. Daher muss sie erst in einen factor und dann in numerisch umgewandelt werden.
* **Tidymodels Version:** correlate()

library(corrr)

corr\_mat <- hrdata %>%  
 select(age, daily\_rate, distance\_from\_home) %>%  
 correlate() %>%   
 rearrange() %>% # Sortieren nach Höhe  
 shave() # Eliminieren der upper triangle

* + Essentiell ist correlate. Das Sortieren mi rearrange() würde ich i.d.R. sein lassen, weil die Reihenfolge in einem Artikel immer inhaltlich bedingt wird und shave() ist v.a. dann hilfreich, wenn man die Matrix anschließend in word oder Excel kopiert und die NAs über Suchen/Ersetzen löschen kann.
  + Default Pearson & pairwise. Es gibt aber auch Kendall und Spearman
* Grafik damit

corr\_mat %>%   
 rplot(shape = 15, colours = c("red", "white", "blue")) +  
 theme\_light()



## Missing data: Analyse und treatment

* Anm. Für den folgenden Abschnitt werden im HR-Datensatz (der ja keine fehlenden Werte enhält) per Zufall fehlende Werte erzeugt. Der Code stammt von chatgpt. Das ist für die Funktionsweise der nachfolgende Teile nicht nötig, aber hilfreich.
* Den Block durchlaufen lassen. Das generiert den Datensatz **hrdata\_miss**

set.seed(123) # Setzt einen Seed für Reproduzierbarkeit

# Funktion, um NA-Werte zu einem zufälligen Prozentsatz jeder Variable hinzuzufügen

add\_random\_na <- function(df) {

for (i in 1:ncol(df)) { # Für jede Spalte im Datensatz

# Bestimmen Sie einen zufälligen Prozentsatz von 1% bis 10%

percent\_na <- sample(1:10, 1) / 100

# Bestimmen Sie die Anzahl der NA-Werte, die basierend auf dem   
 # Prozentsatz hinzugefügt werden sollen

num\_na <- ceiling(nrow(df) \* percent\_na)

# Wählen Sie zufällige Zeilen aus, die NA zugewiesen werden sollen  
 na\_rows <- sample(1:nrow(df), num\_na)

# Weisen Sie NA-Werte in den ausgewählten Zeilen für die aktuelle Spalte   
 # zu  
 df[na\_rows, i] <- NA

}  
 return(df)  
}

# Anwenden der Funktion auf Ihren Datensatz  
hrdata\_miss <- add\_random\_na(hrdata)

### Analyse

Hier gibt’s ein online-Buch zu Missing data mittels R:

<https://bookdown.org/mwheymans/bookmi/>

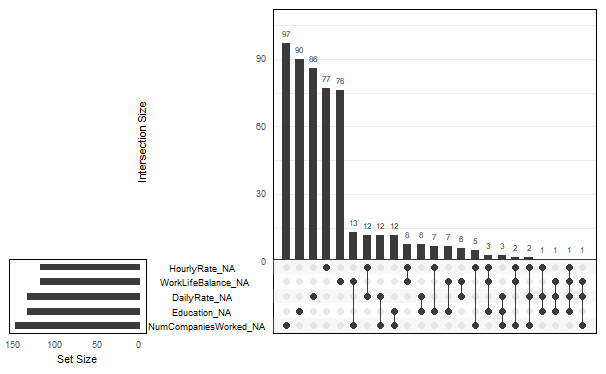
* **Schneller Überblick**

hrdata\_miss %>%   
 skimr::skim()

Relevant sind die Spalten n\_missing und complete\_rate

* **Übersicht über pattern**

hrdata\_miss %>%  
 naniar::gg\_miss\_upset()



* **Zählen der misings in einer bestimmten Variable**

hrdata %>%  
 summarise(sum\_miss = sum(is.na(age)))

Ist in dem Fall halt 0, weil es keine Missings gibt

* **Zählen der Missings ALLER Variablen:**

hrdata %>%  
 purrr::map(~sum(is.na(.))) %>%   
 unlist()

Hierzu wird die map()-Funktion aus dem purrr-paket benutzt (Achtung: 3 "r")

* **Anteil der missings** bekommt man durch die Subtraktion mit den Fällen

hrdata %>%  
 purrr::map\_df(~sum(is.na(.)/length(.))) %>%   
 unlist()%>%   
 round(.,2)

### NAs löschen

* **Radikalste Lösung (don’t!): Alle Fälle löschen, die *irgendwo* fehlen**

hrdata %>%   
 na.omit()

* **Alle Fälle löschen, die überall missing sind**

hrdata %>%   
 filter\_all(any\_vars(!is.na(.)))

### Umwandeln von NaN in NA

hrdata %>%   
 mutate(age = ifelse(is.nan(age), NA, age) )

* …bei mehreren Variablen

hrdata %>%  
 mutate(across(where(is.numeric), ~ifelse(is.nan(.), NA, .)))

### Werte in NA umwandeln

* **Bestimmten Wert bei EINER Variable in NA umwandeln**

hrdata %>%   
 mutate(gender = na\_if(gender, "divers"))

→ Wenn die Sex-Variable einen Eintrag "divers" enthält, dann auf NA setzen (hier unnötig, weil nicht vorhanden)

* **Bei mehreren Variablen**

Hintergrund dürfte meist das Vorhandensein von missing codes (wie -99 etc. sein). Die müssen raus

hrdata %>%  
 mutate(across(c(age, attrition, monthy\_income), na\_if, -99))

* **Bei einem BEREICH von Variablen**

mutate\_at(vars(x1:x5), na\_if, -99)

* **Bei ALLEN Variablen**

hrdata %>%  
 mutate(across(where(is.numeric), na\_if, -99))

(das geht nur bei numerischen Variablen—daher muss die Bedingung rein

* **Bei einer kategorialen Variablen**

hrdata %>%  
 mutate(attrition = na\_if(attrition, "-99"))

### NAs in Werte umwandeln

Manchmal ist es legitim (z.B. bei count-Variablen), ein missing auf 0 zu setzen. Alternative Werte sind je nach Kontext denkbar

* **Spezifische Variablen ersetzen**

hrdata\_miss %>%   
 mutate(  
 hourly\_rate2 = ifelse(is.na(hourly\_rate), 0, hourly\_rate)   
 )

* **NAs aller Variablen durch 0 ersetzen (Don‘t!)**

hrdata\_miss %>%   
 mutate\_if(is.numeric, ~replace(., is.na(.), 0))

* **Missings in Werte umwandeln (z.B. NA → -99)**
  + **Bei einer Variablen**

hrdata\_miss %>%   
 mutate(employee\_count = replace\_na(employee\_count, -99))

* + **Bei mehreren Variablen**

hrdata\_miss %>%   
 mutate\_at(vars(hourly\_rate:monthly\_rate), ~replace\_na(., -99))

Achtung: Dürfen nur numerische Variablen sein

* + **Bei allen (numerischen) Variablen**

hrdata\_miss %>%  
 mutate(across(where(is.numeric), ~replace\_na(., -99)))

# Zeit

* Im Tidyverse kann man mit Datumsangaben oder Zeitvariablen arbeiten
* In Munzert et al. (2014, S. 374) ist ein Beispiel mit Twitter-Daten
* Zentrales Paket ist lubridate ([Cheat sheet](https://github.com/rstudio/cheatsheets/raw/master/lubridate.pdf))
* **Tutorial:** <https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html>
* **Orginal paper:** <https://www.jstatsoft.org/index.php/jss/article/view/v040i03/v40i03.pdf>
* Die hier verwendeten Daten sind die aus dem nycflights-Paket (Flugdaten des NY Flughafens)
* Es gibt 3 Arten
  + **Datum** (<date>)
  + **Zeit** (<time>)
  + **Datetime** (<dttm>): kombiniert Datum und Zeitpunkt (z.B. 2013-04-02 1:05:32).
  + Im base R heißen die POSIXct

## Parsing: Umwandlung in das Format date oder datetime

* Man kann Zeiten aus anderen Variablen herstellen:
  + Aus einem String (siehe 5.1.1)
  + Aus einzelnen Zeit-Komponenten (siehe 5.1.2), wie day, month, year

### Herstellung von Datumsangaben aus einem String (parsing)

* Oft ist eine Datumsvariable als String formatiert. Diese muss durch **ymd(), mdy(), dmy()** in ein date umgewandelt weren. Die gewählte Funktion richtet sich danach, wie im String das Datum beschrieben ist. Klassisches Beispiel ist ein deutsches Datum:

Date Time

<chr> <time>

31. Mai 18 14:00:20

30. Mai 18 08:11:07

24. Mai 18 13:00:13

24. Mai 18 08:00:18

Hier muss bei R gesagt bekommen, was die Elemente (13, Mai, 18) bedeuten. Hier also “day-month-year”. Entsprechend ist die Umwandlung über die dmy()-Funktion

data\_time %>%   
 mutate(Date = **dmy**(Date))

Date Time

<date> <time>

2018-05-31 14:00:20

2018-05-30 08:11:07

2018-05-24 13:00:13

2018-05-24 08:00:18

Achtung: die gewählte Funktion muss zum Inhalt der Klammer passen—sowohl was die Reihenfolge von Jahr, Monat etc. angelangt als auch, was die Nennung von Elementen anbelangt: Enthält die Klammer minuten. muss man auch \_hm wählen (und ncih \_h) sonst kommt der Fehler

All formats failed to parse

Diesen Fehler hatte ich auch mal, weil ich als Rohformat "3/31/99" etc. hatte und aus Versehen mutate(date = dmy(date) genommen hatte—anstatt mdy() Wie man sieht, ist die erste Zahl der Monat!

**Wenn das Format ein anderes ist:**

* **Beispiel "Fri Feb 28 10:11:14 +0000 2014":**

dat$time <- as.POSIXct(dat$created\_at, tz = "UTC",   
 format="%a %b %d %H:%M:%S %z %Y")

Notation im Format-Argument:

* + %a: Wochentag (Abgekürzt in *Buchstaben*, z.B. „Fri“)
  + %d: Tag (In Datumsangabe, z.B. „02“)
  + %b: Monat (Abgekürzt in Buchstaben, z.B. „Feb“)
  + %m: Monat in Zahlangabe
  + %Y: Jahr
  + %z: Zeitzone
  + %H: Stunde
  + %M: Minute
  + %S und sind die Datums- und Zeitvorhersagen

siehe <https://cutt.ly/gr23LeG>

### Zusammensetzung von Datumsangabem aus Komponenten

Manchmal man die einzelnen Komponenten separat gespeichert. hier mal anhand des flights-Datensatzes (Flüge de NYC-Flughafens)

library(nycflights13)

flights %>%   
 select(year, month, day, hour, minute)

year month day hour minute

<int> <int> <int> <dbl> <dbl>

1 2013 1 1 5 15

2 2013 1 1 5 29

3 2013 1 1 5 40

4 2013 1 1 5 45

5 2013 1 1 6 0

6 2013 1 1 5 58

7 2013 1 1 6 0

8 2013 1 1 6 0

* **Zusammenfügen mit make\_date() oder make\_datetime()**

flights %>%  
 select(year, month, day, hour, minute) %>%  
 mutate(departure = make\_datetime(year, month, day, hour, minute) )

year month day hour minute departure

<int> <int> <int> <dbl> <dbl> <dttm>

1 2013 1 1 5 15 2013-01-01 05:15:00

2 2013 1 1 5 29 2013-01-01 05:29:00

3 2013 1 1 5 40 2013-01-01 05:40:00

4 2013 1 1 5 45 2013-01-01 05:45:00

5 2013 1 1 6 0 2013-01-01 06:00:00

6 2013 1 1 5 58 2013-01-01 05:58:00

7 2013 1 1 6 0 2013-01-01 06:00:00

8 2013 1 1 6 0 2013-01-01 06:00:00

9 2013 1 1 6 0 2013-01-01 06:00:00

10 2013 1 1 6 0 2013-01-01 06:00:00

* **Zusammenfügen aus Jahr und Monat zu einem Date**

Ausgangslage war ein tibble in dem nur das Jahr und der Monat vorhanden war (Bier-[Video](https://www.youtube.com/watch?v=1R4X09w7tQ8&t=4s) von Dave).

year month

1 2008 1

2 2008 2

3 2008 3

4 2008 4

5 2008 5

6 2008 6

7 2008 7

Da es keine ym()-Funktion gibt hat er es so gelöst (!)

data %>%

mutate(date = ymd(paste(year, month,1)))

Ergebnis ist eine date-Variable, die halt immer den 01. des Monats angibt

year month date

<int> <int> <date>

1 2008 1 2008-01-01

2 2008 2 2008-02-01

3 2008 3 2008-03-01

4 2008 4 2008-04-01

## Extraktion von Teilelementen

* Aus einer komplexeren datetime-Angabe (z.B. "2013-20-03 5:15:29") kann man sich die Teilelemente extrahieren:

dt <- ymd\_hms("2016-07-08 12:34:56")

* **Extraktion des jeweiligen Elements** aus dem Zeit-Objekt mit den Funktionen

date(dt)

year(x)

month(x)

day(x)

hour(x)

minute(x)

second(x)

yearmonth(x)

**Gundschema geht mittels mutate()**

data %>%

mutate(month = month(date\_month))

(Hier war date\_month ein date (z.B. "2009 Jan"). Ergebnis ist dann eine numerischer Vektor der die Montage von 1-12 durchzählt.

Die Ursprungsvariable kann irgendein Zeitformat sein (datetime, date, etc.). Im o.g. Fall war es „month“

* **Extraktion der Wochennummer.** Die läuft etwas anders. Es gibt zwei Möglichkeiten
  + Mit der isoweek()-Funktion:

weekly\_data <- day\_data %>%

mutate(week = isoweek(time))

Restultiert in einer reinen Zahl der dbl-Klasse

* + Mit der yearweek()-Funktion

weekly\_data <- day\_data %>%

mutate(week = yearweek(time))

Restuliert in einer Variable der week-Klasse, die das Jahr mitnennt, z.B. "2020 W30". Vorteil dieser Variante: Wenn man eine Zeitreihe hat die über einen Jahrwechsel geht, wird das durch die Mitnennung des Jahres adressiert (während bei der isoweek-Variante die Zahl wieder bei 1 anfängt)

* **Eliminierung des Tages aus dem Datum**
  + Hintergrund war der economics-Datensatz, in dem die date-Variable ein vollständiges Datum war (z.B. 1970-07-01), es aber nur monatliche Werte gab (die dann halt immer mit dem 01. anfingen).
  + Lösung: Mittels yearmonth() nur Monat und Jahr extrahieren und die so reduzierte date-Variable beim Umwandeln in das tsibble als Index nehmen (letzteres natürlich nur relevant, wenn man eine Zeitreihe macht). Außerdem kann man die yearweek als Index nehmen der dann mit [1W] angezeigt wird.
* **Extraktion der Wochentags-Nummer**

wday(today())

Grundeinstellung ist hier Sonntag =1, bekommt man durch label=TRUE ausgeschrieben und Reihenfolge angezeigt

Die kann man auch in die Daten schreiben

flights %>%

mutate(departure = make\_datetime(year, month, day, hour, minute)) %>%

mutate(wday = wday(departure, label = TRUE, abbr=FALSE)) %>%

select(departure, wday)

Auf diese Weise kann man auch das Wochenende identifizieren und in eine Variable schreiben (dürfte als saisonale Variable in der PhoneStudy wichtig sein). Beispiel

pedestrian %>%

mutate(

Day = wday(Date, label = TRUE),

Weekend = (Day %in% c("Sa", "So"))

Hier wird im pedestrian Datensatz (im tsibble package) erst die Variable „Day“ gebildet und dann eine weekend-Variable generiert, die TRUE ist wenn der Tag ein Samstag oder Sonntag war. Das kann man sicher auch in eine dummy-Variable übersetzen

**Jahresnummer**

yday(x)

**Extraktion der Quartalsnummer**

quarter(x)

## Runden

Ist nützlich für grouping und summarizing. Zum kann man variable Daten zum Anfang des Monats abgrunden—damit spart man sich die Spezifikation von Intervallen

### floor\_date(): Abrunden zur nächsten unteren Einheit

* (Datenherstellung)

data <- tibble(time = ymd("2020-02-20"))

time

<date>

1 2020-02-20

data %>%  
 mutate(time = floor\_date(time, unit="month"))

time

<date>

1 2020-02-01

🡪 Wird auf 01.02. abgerundet

* floor\_date geht mit allen Komponenten: Year, month, hour, minute, second und dies auch mit bestimmten Werten.

**Beispiel:** Bildung von 2min-Intervallen:

Ausgangslage sind 1min. Auflösungen

1 2020-01-01 00:00:00

2 2020-01-01 00:01:00

3 2020-01-01 00:02:00

4 2020-01-01 00:03:00

5 2020-01-01 00:04:00

6 2020-01-01 00:05:00

7 2020-01-01 00:06:00

8 2020-01-01 00:07:00

9 2020-01-01 00:08:00

data %>%   
 mutate(time = floor\_date(time, "2 mins"))

time

<dttm>

1 2020-01-01 00:00:00

2 2020-01-01 00:00:00

3 2020-01-01 00:02:00

4 2020-01-01 00:02:00

5 2020-01-01 00:04:00

6 2020-01-01 00:04:00

7 2020-01-01 00:06:00

8 2020-01-01 00:06:00

* Wendet man floor\_date auf ein datetime Objekt an und rundet auf eine Datumskomponente (z.B. Monag oder Tag) ab, ist die Zeitkomponente damit hinfällig und mit 00:00:00 angezeigt

### ceiling\_date(): Aufrunden zur nächst oberen Einheit

Dies ist das Komplement zu floordate und rundet gezielt zur nächsten Einheit auf

data\_ts %>%  
 mutate(time = ceiling\_date(time, unit="month"))

time

<date>

1 2020-03-01

### round\_date(): Zur nächstgelegenen Unit auf- oder abrunden

Während floor\_date und ceiling\_date gezielt abrundet, rundet round\_date frei je nach Ausgangswert auf oder ab

Anwendung auf dei Daten von oben (2020-02-20)

data\_ts %>%  
 mutate(time = round\_date(time, unit="month"))

time

<date>

1 2020-03-01

🡪 Der 20.02. ist gegen Ende Feburar, wird daher auf 01.03. aufgerundet

# Text

## Grundprozeduren in der Manipulation von Text

* **Daten:**

library(titanic)

data(titanic\_train)

text <- tibble(titanic\_train)

* **Länge des strings anzeigen** (Anzahl der character)

str\_length(text$Name)

[1] 23 51 22 44 24 16 23 30 49 35 31 24 30 27 36 32 20 28 55 23 20

[22] 21 27 28 29 57 23 30 29 19 24 46 24 21 23 30 16 24 34 27 46 56

[43] 19 40 29 24 17 25 19 45 26 28 40 50 30 17 17 19 28 34 21 19 27

* + Zeigt alle Zeichen inkl. Leerzeichen
  + Wie das innerhalb einer pipeline geht weiß ich nicht
* String in **Kleinbuchstaben** wandeln(str\_to\_lower() )

text %>%   
 mutate(Name\_lower = str\_to\_lower(Name))

* Weitere:
  + In Blockbuchstaben mit str\_to\_upper()
  + Anfangsbuchstaben **groß** schreiben: str\_to\_title()
* **str\_trim():** Whitespace vor dem Anfang oder nach dem Ende entfernen (nicht **zwischendrin**)

temp\_text = tibble(text = c(" abc", "efg "))

temp\_text %>%   
 mutate(text\_without\_space = str\_trim(text))

# A tibble: 2 x 2

text text\_without\_space

<chr> <chr>

1 " abc" abc

2 "efg " efg

Will ich eine Leerstelle **zwischen 2 strings** löschen, geht das mit str\_replace:

mutate(word = str\_replace\_all(word, " ", ""))

* **str\_trunc():** String kürzen

text %>%   
 mutate(Name\_cut = str\_trunc(Name, width = 10, side='right', ellipsis = '')) %>%   
 select(Name, Name\_cut)

# A tibble: 891 x 2

Name Name\_cut

<chr> <chr>

1 Braund, Mr. Owen Harris "Braund, Mr"

2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) "Cumings, M"

3 Heikkinen, Miss. Laina "Heikkinen,"

4 Futrelle, Mrs. Jacques Heath (Lily May Peel) "Futrelle, "

5 Allen, Mr. William Henry "Allen, Mr."

* + **width:** Auf wieviele Zeichen wird gekürzt
  + **side:** Wo wird gekürzt
  + **elipsis:** Wie wird angezeigt, was gekürzt wird (oben: gar nicht; man kann aber auch punkte oder x oder . oder was auch immer eintragen. Das erscheint dann als sympol. Lässt man das elipsis-Argument ganz weg, kommt „...“

## str\_c() und unite: Zusammenfügen von strings

* Macht dasselbe wie paste!
* Ausgangslage: 2 strings

string1 = c("This is part one")

string2 = c("This is part two")

joined\_string = tibble(string1, string2)

* **Zusammenfügen zu einem string, separiert durch einen Bindestrich**

string\_join %>%   
 mutate(string\_vec = str\_c(string1, string1, sep=" - ") )

string1 string2 string\_vec

<chr> <chr> <chr>

1 This is part one This is part two This is part one - This is part one

* Das ganze kann man auch super zum Zusammenfügen von Tabellen machen!

mean\_table = tibble(mean = c(1,5), sd = c(.4, .8))

# A tibble: 2 x 2

mean sd

<dbl> <dbl>

1 1 0.4

2 5 0.8

mean\_table %>%   
 mutate(mean\_sd = str\_c(mean, "(", sd, ")")) %>%   
 select(mean\_sd)

mean\_sd

<chr>

1 1(0.4)

2 5(0.8)

* Wenn man **mehrere strings/Variablen hat** und man will die nicht ausschreien ist unite() besser

unite("top\_frex\_words", word.1:word.5, remove = TRUE, sep=", ")

Hier ist top\_frex\_words die neue Variable, in der die 5 Worte von 1-5 zusammengefügt werden. Mit str\_c() muss man sie glaub ich als word.1, word.2…. ausschreiben

## str\_split() & separate(): Zerteilen eines strings / Tokenizing

* Der command trennt einen string automatisch in alle Bestandteile. Ergebnisse ist ein **genestetes Tibble**, dass man wie bei den broom-Beispielen unnesten kann

text %>%   
 mutate(Names = str\_split(Name, pattern=" ")) %>%   
 select(PassengerId, Names)

A tibble: 891 x 2

PassengerId Names

<int> <list>

1 1 <chr [4]>

2 2 <chr [7]>

3 3 <chr [3]>

4 4 <chr [7]>

5 5 <chr [4]>

6 6 <chr [3]>

Unnested man es jetzt, stehen alle Begriffe genested in IDs untereinander

# A tibble: 3,626 x 2

PassengerId Names

<int> <chr>

1 Braund,

1 Mr.

1 Owen

1 Harris

2 Cumings,

2 Mrs.

2 John

* **Tokenization**
  + **Einzelne Worte in token**

text %>%  
 unnest\_tokens(word, text)

(word ist die Zerteilungseinheit)

* + **Bei ngrams** (n-grams, bigrams):

text %>%   
 filter(!is.na(text) & text !="") %>%   
 unnest\_tokens(word, text, token = "ngrams", n = 2)

* **separate(): Teilen eines strings in verschiedene Variablen**
  + Mittels separate() (siehe Abschnitt) aus tidyR kann man einen string in verschiedene VARIABLEN entlang eines separators teilen:

corpus = c("Heute mach ich dies; morgen mach ich das")

textcorpus = tibble(corpus)

textcorpus %>%   
 separate(corpus, c("Heute", "Morgen"), sep=";")

Heute Morgen

<chr> <chr>

1 Heute mach ich dies " morgen mach ich das"

Dabei wird die Ursprungsvariable eliminiert

## str\_detect: Pattern suchen und zählen

### str\_detect(): Suche nach einem Zeichen, Wort oder pattern

* Sehr geeignet mit filter
* **Beispiel 1: Suche nach einem bestimmten Word**

text %>%   
 filter(str\_detect(Name, pattern="William")) %>%   
 select(Name)

Name

<chr>

1 Allen, Mr. William Henry

2 Saundercock, Mr. William Henry

3 Williams, Mr. Charles Eugene

4 Sloper, Mr. William Thompson

5 Spencer, Mrs. William Augustus (Marie Eugenie)

6 Turpin, Mrs. William John Robert (Dorothy Ann Wonnacott)

* + Das geht auch mit Worteilen (pattern = ".ill")
  + Achtung: Punkte sind Platzhalter. Sucht man nach „Mr.“ findet er auch Mrs.
  + str\_subset() scheint m.E. dasselbe zu machen

tibble(Willi = str\_subset(text$Name, "William"))

* **Beispiel 2: Flexibles Filtern nach pattern / regex**

season1 %>%   
 filter(str\_detect(title, "[T|t]he"))

Hier enthält das tibble season1 drei fiktive Episoden in der Variable „title“. Es werden nun alle rausgefiltert, die „the“ oder „The“ enhalten

* **Beispiel 3: Kombinationen von regexes**

Hier sollten Substantive gefunden werden. Strategie: „finde alle the mit „a“ oder „the“ anfangen (^) und danach ein Wort haben

sentences <- as\_tibble(sentences)

noun <- "(a|the) ([^ ]+)"

sentences %>%   
 filter(str\_detect(value, noun))

Analog kann man die Worte mit str\_extract() extrahieren (siehe nächster Abschnitt mit dem selben Beispiel

* Will man die row number der fittenden Elemente haben, kann man das mit str\_which (aka „which string“) rausbekommen. Dafür müssen die Daten anscheinend als Vektor rausgezogen werden

words %>%   
 pull(words) %>%   
 str\_which("[abc]")

Ausgabe ist ein numerischer Vektor.

* **str\_count** zählt die matches pro string

words\_sub %>%   
 pull(words) %>%   
 str\_count("a")

[1] 2 0 0 1 1 0 1 0 1 0 0 0 0 0 0 3 1 0 0 0 0 0 0 0 1 0 0 etc.

Z.B. sind im ersten Wort („manage“) 2 a’s

## str\_extract() und str\_sub(): Extrahieren von Zeichen

### Extraktion nach der Position des Zeichens

text %>%  
 mutate(newstring = str\_sub(Name, start = 1L, end = 10L)

Start und End bezeichnen die Stellung. Hiwer würden die ersten 10 Zeichen (inkl. Leerzeichen) extrahiert

### Extraktion nach Inhalt

* Z.B. bestimmer Begriffe oder regexes
* str\_extract findet die gesuchten Teil nicht nur (wie str\_detec), es schneidt sie aus dem string heraus.
* Eine alternative ist extract() aus tidyR—v.a. bei festen Zeichenketten (wie „Meier (2002)“), wo die „(2002)“ extrahiert und in eine eigene Spalte kopiert wird (siehe Beispiel #5)
* **Beispiel 1: Suche nach bestimmten Zeichen:**

Geht einfach ohne die character-class mit dem konkreten Bezug

text %>%

mutate(new\_col = str\_extract(Ticket, pattern="PC")) %>%

filter(!is.na(new\_col)) %>%

select(Ticket)

# A tibble: 60 x 2

Ticket new\_col

<chr> <chr>

1 PC 17599 PC

2 PC 17601 PC

3 PC 17569 PC

4 PC 17604 PC

5 PC 17572 PC

6 PC 17605 PC

* + Kombinationen gehen durch z.B. pattern=c("PC|A/5|PP")
  + ! Der Unterschied zwischen str\_extract und str\_detect:
    - str\_extract extrahiert das Ziel-Pattern (s.o. „PC“) innerhalb mutate
    - str\_detect gibt eine logische Variable mit TRUE /FALSE aus. Deshalb geht es direkt im filter-Befehl (nicht mutate).
* **Beispiel 2: Strings mit Farben extrahieren**

Hier mal am Datensatz sentences:

data(sentences)

sentences = as\_tibble(sentences)

Step 1: Suchvektor generieren (könnte auch nen Dictionary sein)

colours <- c("red|orange|yellow|green|blue|purple")

Step 2: Neue Variable generieren, die die extrahierte Farbe enhält und diese dann filtern

sentences %>%

mutate(new\_col = str\_extract(value, colours)) %>%

filter(!is.na(new\_col))

value new\_col

<chr> <chr>

1 Glue the sheet to the dark blue background. blue

2 Two blue fish swam in the tank. blue

3 The colt reared and threw the tall rider. red

4 The wide road shimmered in the hot sun. red

5 See the cat glaring at the scared mouse. red

* **Beispiel 3: Alle Strings filtern, die in eine regex enthalten**

Die Ticket-Variable im Titanik datensatz enhält oft seltsame Buchstaben. Die folgende pipeline identifizeirt alles Einträge dei die definierten pattern haben (=mind. 1 Buchstabe) und zieht sie raus:

text %>%

mutate(new\_col = str\_extract(Ticket, pattern="[A-Z]")) %>%

filter(!is.na(new\_col)) %>%

select(Ticket)

# A tibble: 230 x 1

Ticket

<chr>

1 A/5 21171

2 PC 17599

3 STON/O2. 3101282

4 PP 9549

5 A/5. 2151

6 PC 17601

7 PC 17569

8 C.A. 24579

* **Beispiel 4: Grouped match mit einer regex**

(Wiederholung des Beispiels aus dem str\_detect-Abschnitt. Mittels regex sollen alle Substantive „(a | the) + Wort“ extrahiert werden

noun <- "(a|the) ([^ ]+)"

sentences %>%

mutate(subst = str\_extract(value, noun))

# A tibble: 720 x 2

value **subst**

<chr> <**chr**>

1 The birch canoe slid on the smooth planks. **the smooth**

2 Glue the sheet to the dark blue background. **the sheet**

3 It's easy to tell the depth of a well. **the depth**

4 These days a chicken leg is a rare dish.  **a chicken**

5 Rice is often served in round bowls. **NA**

6 The juice of lemons makes fine punch. **NA**

7 The box was thrown beside the parked truck. **the parked**

* **Beispiel 5: Alternative** extract()

author = c("Meier (2002)", "Steinmetz et al. (2017)", "Schmidt & Schulze (2015)")

refs = tibble(author)

refs %>%

extract(author, "year", "\\((\\d\\d\\d\\d)\\)$", remove=FALSE, convert=TRUE)

author year

<chr> <int>

1 Meier (2002) 2002

2 Steinmetz et al. (2017) 2017

3 Schmidt und Schulze (2015) 2015

* + remove=FALSE: Behält den Ausdruck in der Ursprungsvariable
  + convert=TRUE: Konvertiert das Jahr in einen integer

Das Analogon mit str\_extract() wäre

refs %>%

mutate(year = str\_extract(author, "[\\d][\\d][\\d][\\d]"))

Mit der regex in extract klappt es nicht. Außerem liefert str\_extract einen character, der manuell imkodiert werden muss

* **Parsen von Zahlen.**

mit parse\_numbers() kann man einfach Zahlen extrahieren, was in vielen Fällen bereits ausreicht:

data %>%

mutate(X = parse\_number(X)

* + Das ist z.B. anwendbar, wenn der Eintrag sowas wie #1, #2 etc. ist.
  + Sind im string 2 Zahlen (z.B. „5 people in 10“, wird die erste genommen (🡪 5)

## str\_remove(): Löschen von Zeichen

* Das Prinzip ist das selbe wie bei str\_extract. Entweder man legt einen Zielvektor vorher fest und löscht dann alle string-Elemnete die den Vektor matchen oder man macht das in der Funktion: Hier werden in allen Titanik-Tickets die Kombinationen PC, A/5 und PP gelöscht
* Weil dann whitespace entsteht, lösch ich den gleich mit:

text %>%

mutate(new\_col = str\_remove(Ticket, pattern=c("PC|A/5|PP") ),

new\_col = str\_trim(new\_col)) %>%

select(new\_col)

Ticket new\_col

<chr> <chr>

1 A/5 21171 21171

2 PC 17599 17599

3 STON/O2. 3101282 STON/O2. 3101282

4 113803 113803

## str\_replace(): Replace / Ersetze

* Z.B. Ersetzen des Namens William in „Willi“

text %>%

mutate(Name = str\_replace(Name, pattern="William", replacement = "Willi")) %>%

filter(str\_detect(Name, "Willi")) %>%

select(Name)

Name

<chr>

1 Allen, Mr. Willi Henry

2 Saundercock, Mr. Willi Henry

3 Willis, Mr. Charles Eugene

4 Sloper, Mr. Willi Thompson

5 Spencer, Mrs. Willi Augustus (Marie Eugenie)

6 Turpin, Mrs. Willi John Robert (Dorothy Ann Wonnacott)

* Auch hier kann das pattern OR-Kombinationen enthalten oder regexes

## Regular expressions

### Ressourcen

* Videos: <https://www.youtube.com/watch?v=rhzKDrUiJVk>
* Webseiten:
  + **[Fehler! Linkreferenz ungültig.](https://regexr.com/59hn4)** Zum Ausprobieren (Vorsicht, oft nicht direkt übertragbar in stringr, z.B. \d (allgemeine regex) vs. \\d (stringr-Form). Der Grund ist, das \ auch ein Zeichen ist, dass „excaped“ werden muss—und dies passiert eben mit einem weiteren \ (siehe Abschnitt escape unten).
  + <https://regexone.com/> Tutorial mit Übungen
  + <https://www.jumpingrivers.com/blog/regular-expressions-every-r-programmer-should-know/> **Rbloggers tutorial NOCH NICHT GELESEN**
* Regular expressions sind ganz einfach „Pattern“, die die Grundlage für die Suche (z.B. mit filter(str\_detec()) oder Manipulation mittels mutate(x = str\_extract()) etc. sind. All diesen liegt ein pattern zugrunde, mit dem der Eintrag gematcht wird. Dieses Pattern kann variieren von fix zu flexible in der Rangreihe
  + (pattern="William")
  + (pattern = "William|John|Joe")
  + (pattern = "[T|t]his")
  + (pattern = "(a|the) ([^ ]+)")

### Wildcards

#### Der Punkt (.)

* Ein Punkt, gibt an, dass jedes Zeichen (inkl. Interpunktion) gematcht wird (Achtung: Sucht man einen wortwörtlichen Punkt, muss man das escape-Symbol (\\) benutzten (siehe unten mehr)

Die folgenden Beispiele gelten mal für folgenden string

X = c("till","Bill","still","silly","thrill","until","thriller")

X = tibble(X)

X %>% filter(str\_detect(X, pattern=".ill"))

X

<chr>

1 till

2 Bill

3 still

4 silly

5 thrill

6 thriller

* + Bis auf until findet er alles
  + Der Punkt ist ein Platzhalter für jeden Buchstaben vor ill
  + Jeder weitere Punkt schließt einen weiteren Buchstaben ein (davor oder dahinter
  + Beispiel:

|  |  |
| --- | --- |
| eo | Some stupid p**eo**ple will think |
| .eo | Some stupid **peo**ple will think |
| ...eo | Some stupi**d peo**ple will think |
| eo.. | Some stupid p**eopl**e will think |
| ..eo.. | Some stupid **peop**le will think |

#### Anchors

* **^: Beginnend mit**

Kommt ^ **außerhalb** des patterns vor, bedeutet es „beginnend mit“

Beispiel "^[He|he]" 🡪 Findet alle strings die entweder mit He oder he beginnen

Startet es innerhalb des patterns, bedeutet es dagegen **NOT**

Z.B. "[^t]ill" 🡪 Findet „kill“ und „bill“, aber nicht „still“ und „till“

* **$: Endend mit**

$ Ist das Gegenteil, das beschänkt die Suche auf das Ende

"cool$" -> Findet „Das ist ja cool“

### Character class

Mittels einer character-class kann man auf einen Schlag einen ganzen Bereich definieren, mit dem die Suche fittet—z.B. alle Buchstaben oder alle Zahlen.

Achtung: Hier weichen die regexes in stringr von der der o.g. Videos und der regex-Website ab!

Schreibweise: str\_detect(X, pattern="[:alpha:]")

|  |  |  |
| --- | --- | --- |
| **Alternative** | **Target** | **Beispiel** |
| [:alpha:] | Buchstaben | **abc** **ABC** 123 .!?{( |
| [:lower:] | Kleinbuchstaben. Den Suchbereiche kann man spezifizieren, z.B. mit [e-h] | **abc** ABC 123 .!?{( |
| [:upper:] | Großbuchstaben | abc **ABC** 123 .!?{( |
| \\d | Zahlen | abc ABC **123** .!?{( |
| [\\D](file:///\\D) | Nicht-Zahlen | **abc ABC** 123 **.!?{(** |
| [:alnum:] | Buchstaben und Zahlen | **abc ABC 123** .!?{( |
| [:punct:] | Zeichen | abc ABC 123 **.!?{(** |
| [:space:] | Leerzeichen (whitespace) | abc ABC 123 .!?{( |
| \\S | Match non-whitespace. | **abc ABC 123 .!?{(** |

### Zählen des Vorkommens / repetitions

* Setzt man +, \*, ? hinter ein Zeichen, Wort oder die character class folgt daraus eine bestimmte Wiederholung (oder deren Ausschluss).
* !! Diese Operationen erscheinen bei einzelnen Buchstaben/Worten wenig sinnvoll (v.a. deren Unterschiede). Sie sind aber bei Kombinationen extrem hilfreich.
* Anm. Im folgenden sind die im Match-Beispiel rotgefärbten Buchstaben nur deshalb rot, weil damit auf eine Besonderheit hingewiesen werden soll. Der Match bezieht sich auf alle fett markierten

|  |  |  |  |
| --- | --- | --- | --- |
| **Expr** | **Erklärung** | **Bsp.** | **Match** |
| x+ | **1 or more:** x muss mindestens einmal vorkommen | e+ | Th**e** cat ran down th**e** str**ee**t. It was s**e**arching for a mous**e** to **e**at |
| x? | **0 or 1:** x muss nicht vorkommen—wenn dann aber nur 1x. Dies und das nächste macht x optional. Es kann vorkommen, muss aber nicht Bei **Kombinationen** interessant, z.B. e+a?:Es muss ein e drin sein—optional gefolgt von a  (Anm. e? findet auch str**ee**t. Es geht um die Anz. in komplexeren Elementen) | e+a? | Th**e** cat ran down th**e** str**ee**t. It was s**ea**rching for a mous**e** to **ea**t  (mind. 1x e, und ea optional) |
| x\* | **0 or more:** x *kann* im string vorkommen (egal wie oft), muss aber nicht. Bei Kombinationen relevant | e\*a | The c**a**t r**a**n down the street. It w**a**s s**ea**rching for **a** mouse to **ea**t  (🡪 findet 1x a und optional ea) |
|  |  | re\* | The cat **r**an down the st**ree**t. It was sea**r**ching fo**r** a mouse to eat  (findet ein r und optonal re; dann aber auch mehrfach) |
|  |  | grape\*+ | **"grape"**, **"grapefruit"**  Bei grape.+ würde grape nicht gematcht |
|  |  | total .\* | **“total grain”, “total non-grain”**  Alles nach total! (würde auch mit .+ gehen |
| x{n} | Finde Elemente, die n-mal das Zeichen x haben | e{2} | The cat ran down the str**ee**t. It was searching for a mouse to eat |
|  | ! Finde ein e 🡪 „street“ weil es einmal e hat. | e{1} | Th**e** cat ran down th**e** str**ee**t. It was s**e**arching for a mous**e** to **e**at |
|  | Kombis mit einer character class | [\\w]{4} | The cat ran **down** the **stre**et. It was **sear**ching for a **mous**e to eat |
| x{n,} | Finde Elemente mit n oder mehr | [\\w]{4,} | The cat ran **down** the **street**. It was **searching** for a **mouse** to eat |
| x{n,m} | Finde Elemente die zwischen n und m Zeichen haben | [\\w]{4,5} | The cat ran **down** the **stree**t. It was **searc**hing for a **mouse** to eat |

**! Multiple matches:** Wichtig: Der parsing Algothmus stoppt, sobald das pattern nicht mehr zutrifft. Das führt dazu, dass multiple matches in eiem string (Z.B. “ich mach 42, aber 12 ist auch ok) durch durch-geparsed werden. Das wird durch die stringr „\_all“-Varianten adressiert!

#### Escape-Funktion

Escaping bedeutet, sich direkt auf einen Metacharacter zu fokussieren, anstatt ihn nur als wildcard zu benutzen. Das geht über doppelte backslashes.

Beispiel:

\\. findet Punkte

\\( findet eine offene Klammer

### Oder

„Oder“-Suchen werden mit square brackets gemacht.

* **Datengrundlage:** Ein sample aus den words-Datensatz in stringr

words\_sub = words %>%

sample\_n(50)

[1] "thing" "match" "get" "kid" "sister" "project"

[7] "toward" "special" "practise" "around" "bloke" "hard"

[13] "account" "achieve" "field" "sell" "detail" "tea"

[19] "suit" "enjoy" etc.

Die folgenden Beispiele dann entsprechend einsetzen

words\_sub %>%

filter(str\_detect(words, "[abc]"))

#### Beispiele

* [abc] findet a oder b oder c. Darauf basiert auch das "[Th]is"
* [^abc] findet alle die nicht a oder b oder c haben
* "^[abc].+" findet alle Worte die entweder mit a oder b oder c *anfangen*. Merke: ^= Beginnt mit a oder b; .+ = alle Zeichen danach (bis zum nächsten Leerzeichen, oder Punktion)
* (?=c): Finde alle, die Buchstaben vor c haben (findet aber auch welche, die damit anfangen)

### Capture groups

* Wichtig, wenn der string mehrere Elemente enthält
* Werden durch Klammen angezeigt ("( )")
* Das was in der Klammer ist, wird gematcht und widergegeben (z.B. Worte)
* **Beispiel 1:** Alle Farben kommen im selben string vor. Sie sollen alle geparsed werden

colors = "red, blue, green"

str\_match(colors, "(\\w+)\\W\*(\\w+)\\W\*(\\w+)\\W\*")

„Ziehe alle alphanumerischen Zeichen (🡪 erste CG), dann folgt whitespace (🡪 ignorieren), dann dasseble noch 3 mal“. Problem: Man muss soviele Wiederolungen manuell machen wie token im string sind.

* **Beispiel 2:** Capture apples and bananas.
  + Vektor enthält Früchte. Zielausgabe sollen apples und bananas sein:

str\_match(fruits, „(ap|ban).\*“))

Achtung: es dürfen keine Leerzeichen um das | vorhanden sein

Anwendung auf en tibble:

colors2 %>%

filter(str\_detect(value, "(re|gr).\*" ))

value

<chr>

1 red

2 green

* **Telefonnummer-Beispiel** (aus Colin Quirks Video)

phone\_numbers = tibble(phone = c("(541) 431 3104","(947) 391-3940","Home: 352-495-2391","(931)-739-3502","8599526130"))

phone

<chr>

1 (541) 431 3104

2 (947) 391-3940

3 Home: 352-495-2391

4 (931)-739-3502

5 8599526130

phone\_numbers %>%

mutate(number = str\_extract(phone, pattern="(\\d.\*)")) %>%

mutate(number = str\_remove(number, pattern="\\)")) %>%

mutate(number = str\_remove\_all(number, pattern="\\-")) %>%

mutate(number = str\_remove\_all(number, pattern="\\W"))

phone number

<chr> <chr>

1 (541) 431 3104 5414313104

2 (947) 391-3940 9473913940

3 Home: 352-495-2391 3524952391

4 (931)-739-3502 9317393502

5 8599526130 8599526130

Ich weiß nicht, ob das effizienter geht. Das Extrahieren der Zahlen und das Löschen der Klammern/Bindestriche muss glaub ich getrennt passieren, aber ich bekomme es nicht hin, Klammern und Bindestriche in einem pattern zu löschen

## Beispiele für regexes

* **Wieviel Worte fangen mit t an?**

( „words“ ist ein in stringr enthaltender Character-Vektor mit rund 1000 Worten)

sum(str\_detect(words, "^t"))

65

* **Wieviel Prozent der Worte enden mit einem Vokal**

mean(str\_detect(words, "[aeiou]$"))

[1] 0.2765306

* **Anzahl der Leute die KEINEN Vokal enthalten**

sum( !str\_detect(words, "[aeiou]"))

* **Zählen von Elementen in einem String-Vektor** (#count)

df %>%

mutate(N\_a = str\_count(words, "e$"))

A tibble: 980 x 2

words N\_a

<chr> <int>

1 a 0

2 able 1

3 about 0

4 absolute 1

5 accept 0

6 account 0

7 achieve 1

* Ich weiß den Unterschied zu str\_lenght() nicht
* Count zählt per default die Zeichen--wenn man die Anzahl der Sätze will geht das mit boundary()
* **Komplexes Beispiel 1**

title = c("The beginning",

"First arrival (Part 1)",

"Kill the dot.")

season1 = tibble(title)

Ziel: Lösche strings mit einem Punkt oder einem Klammer (sammt inhalt)

season1 %>%

mutate(episodeName = str\_remove\_all(title, "\\.| \\(.\*") )

title episodeName

1 The beginning The beginning

2 First arrival (Part 1) First arrival

3 Kill the dot. Kill the dot

* + \\. löscht den Punkt
  + (.\* sucht die Klammer und löscht alles ab dort
  + | ist „oder“
* **Komplexes Beispiel 2**

Ein Bespiel aus Julia Silge’s LASSO Video: erst wird ein Objekt gebildet, auf das in der Funktion remove\_all bezoegen wird. In dem Video werden aus Serientiteln alle Punkte, Nummern, „the“, „and“ und die Worte A und B entfernt

remove\_regex = "[:punct:]|[:digit:] |WortA |WortB |the |and]"

(Achtung: Die Lerzeichen sind wichtig)

mutate(variableNew = str\_remove\_all(variableOld, remove\_regex)

Alternative: Den character class operator mit dem Meta-Charakter verwenden

C = c("abc", "a.c", "a\*c", "a c")

str\_detect(C, "[.]")

Findet „a.c“. Tippt man aber "[a.]", findet er alle, weil nun der Punkt als wildcard für alles nach a interpretiert wird. Wird das a aber vor der Klasse geschrieben "a[.]" findet er „a.“

Schließlich kann auch whitespace damit gefunden werden: "a[ ]“ findet „a c“

## Textfeatures (GIBT’S WOHL NICH MEHR)

* Mit dem textfeatures Package lassen sich zentrale Merkmale eines Textes (inkl. mehrerer Sentiment scores extrahieren
* Fallstudien
  1. <https://rpubs.com/jflowers/563229>
  2. [Video](https://www.youtube.com/watch?v=wd4MZHx9F9Y&t=312s&ab_channel=JuliaSilge) von Julia über The next airbender
* Mit dem word\_dim Arugment kann man word2vec Dimensionen extrahieren. Entweder eine Zahl oder NULL wählen (dann wird eine „reasonable“ Anzahl gewählt)
* Beispiel

textfeatures <- textdata %>%

select(text) %>%

textfeatures(., sentiment=TRUE,

word\_dims = 300,

normalize=FALSE)

|  |  |
| --- | --- |
| n\_urls | Die Anzahl der URLs in einem Text. |
| n\_uq\_urls | Die Anzahl der unique URLs in einem Text. |
| n\_hashtags | Die Anzahl der Hashtags in einem Text. |
| n\_uq\_hashtags | Die Anzahl der unique Hashtags in einem Text. |
| n\_mentions | Die Anzahl der Erwähnungen in einem Text. |
| n\_uq\_mentions | Die Anzahl der eindeutigen Erwähnungen in einem Text. |
| n\_chars | Die Anzahl der Zeichen in einem Text. |
| n\_uq\_chars | Die Anzahl der eindeutigen Zeichen in einem Text. |
| n\_commas | Die Anzahl der Kommas in einem Text. |
| n\_digits | Die Anzahl der Ziffern in einem Text. |
| n\_exclaims | Die Anzahl der Ausrufezeichen in einem Text. |
| n\_extraspaces | Die Anzahl der zusätzlichen Leerzeichen in einem Text. |
| n\_lowers | Die Anzahl der Kleinbuchstaben in einem Text. |
| n\_lowersp | Der Prozentsatz der Kleinbuchstaben in einem Text. |
| n\_periods | Die Anzahl der Punkte in einem Text. |
| n\_words | Die Anzahl der Wörter in einem Text. |
| n\_uq\_words | Die Anzahl der eindeutigen Wörter in einem Text. |
| n\_caps | Die Anzahl der Großbuchstaben in einem Text. |
| n\_nonasciis | Die Anzahl der nicht-ASCII-Zeichen in einem Text. |
| n\_puncts | Die Anzahl der Satzzeichen in einem Text. |
| n\_capsp | Der Prozentsatz der Großbuchstaben in einem Text. |
| n\_charsperword | Die durchschnittliche Anzahl von Zeichen pro Wort in einem Text. |
| sent\_afinn | Das Sentiment eines Textes, das mithilfe des AFINN-Lexikons berechnet wurde. Geht von -5 bis +5 [noch mal checken: Ich glaube, die werden aufsummiert für den gesamten Text. Müsste man also durch n\_char teilen] |
| sent\_bing | Das Sentiment eines Textes, das mithilfe des Bing-Lexikons berechnet wurde. Geht von -1 bis +1 (anscheinend ebefalls aufsummiert) |
| sent\_syuzhet | Das Sentiment eines Textes, das mithilfe des Syuzhet-Lexikons berechnet wurde. Geht von -1 bis +1 |
| sent\_vader | Das Sentiment eines Textes, das mithilfe des VADER-Lexikons berechnet wurde. |
| n\_polite | Die Anzahl der höflichen Wörter in einem Text. |
| n\_first\_person | Die Anzahl der Personalpronomen im ersten Person Singular in einem Text. |
| n\_first\_personp | Der Prozentsatz der Personalpronomen im ersten Person Singular in einem Text. |
| n\_second\_person | Die Anzahl der Personalpronomen im zweiten Person Singular in einem Text. |
| n\_second\_personp | Der Prozentsatz der Personalpronomen im zweiten Person Singular in einem Text. |
| n\_third\_person | Die Anzahl der Personalpronomen im dritten Person Singular in einem Text. |
| n\_tobe | Die Anzahl der Verben "sein" in einem Text. |
| n\_prepositions | Die Anzahl der Präpositionen in einem Text |

## Part-of-speech (POS) tagging

<https://ladal.edu.au/tagging.html>

(der Code ging so nicht—konnte ihn anpassen). Meiner ist im Skript POS.R):

library(dplyr)

library(stringr)

library(udpipe)

library(flextable)

library(tidyverse)

# load text

text <- readLines("https://slcladal.github.io/data/testcorpus/linguistics06.txt", skipNul = T)

# clean data

text <- text %>%

str\_squish()

# download language model

m\_eng <- udpipe::udpipe\_download\_model(language = "english-ewt")

m\_eng <- udpipe\_load\_model(file = "C:\\Users\\Holger\\Dropbox\\Science Themenfelder\\NLP\\english-ewt-ud-2.5-191206.udpipe")

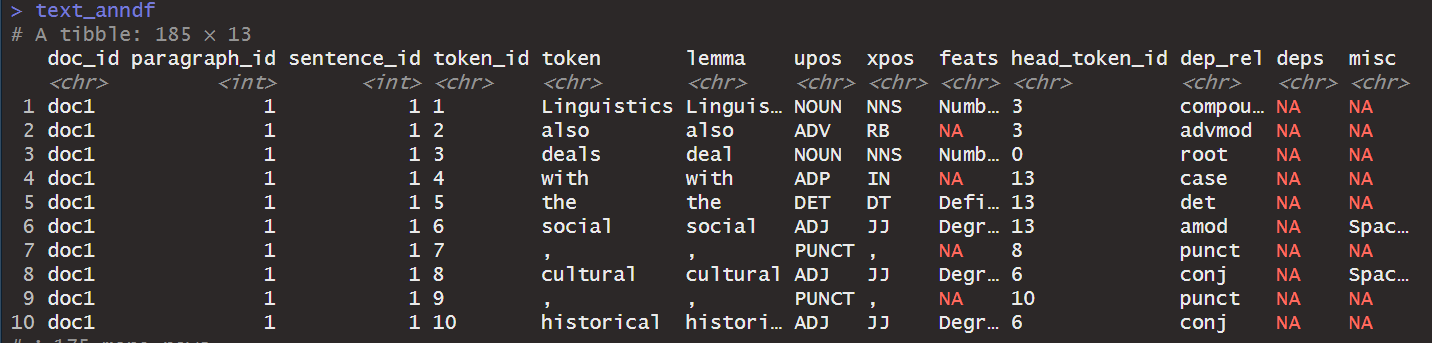
text\_anndf <- udpipe::udpipe\_annotate(m\_eng, x = text) %>%

as\_tibble() %>%

dplyr::select(-sentence)

text\_anndf

die POS sind in der Variable upos



# Grafiken

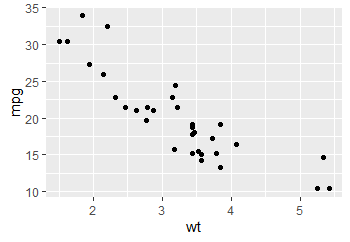
* Grafiken setzen sich in ggplo2 aus **Layern** zusammen. Ein plot besteht dabei zunächst aus 2 Basislayern und optional weiteren. Das macht einfache Grafiken schnell möglich und bietet Möglichkeiten für das Finetuning.
* Die zentrale Funktion ist ggplot(). Sie enthält das mapping (→ Zuweisung der Variablen) das in der Sub-Funktion aes (für „aesthetics“) bestimmt wird:

dataset %>%  
 ggplot(aes(x, y)

* Anschließend folgt das betreffende geom (das heißt das geometrische Element). Die wichtigsten sind
  + **geom\_point:** Punkte (z.B. in einem Scatterplot)
  + **geom\_line:** Linien
  + **geom\_bar** oder **geom\_col:** Balkendiagramm
  + **geom\_boxplot:** Boxplot

dataset %>%   
 ggplot(aes(x, y)) +  
 geom\_point()

* An diese beiden Grund-Layer können sich dann bei Bedarf weitere Layer anschließen wie z.B. Aufsplittung des plots in zwei oder mehrere nach einer Gruppierungsvariable oder Drehen des plots um 90°.
* **Achtung:** Diese werden mit einem **simplen Pluszeichen** (nicht pipe-operator) an die Basis-Layer angehängt.
* Die Basisvariante von ggplot-Grafiken kommt mit einem grauen Hintergrund. Dies nennt man „Theme“:



Man kann das theme durch den layer theme\_.... ändern. Attraktive Alternativen sind

* + theme\_bw()
  + theme\_light()
  + theme\_classic()
  + theme\_minimal()

Meine Favorierten sind theme\_light() und theme\_minimal()

Anstatt den layer manuell an einen plot anzuhängen kann man das theme am Beginn des Skritps durch

theme\_set(theme\_minimal() )

bestimmen. Auf die Klammern achten!

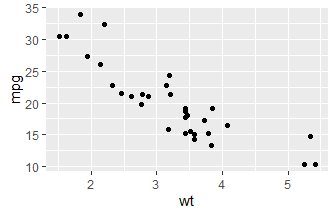
## Scatterplot

### Daten für das kommende Kapitel

* Die hrdata waren simuliert und enthielten keinerleich sinnvolle Muster, für die eine grafische Veranschaulichung attraktiv wäre.
* Daher nutze ich im folgenden den **mtcars-Datensatz**, der automatisch mitgeladen wird. Er enthält 32 Autos und deren Eigenschaften

### Grundform

mtcars %>%   
 ggplot(aes(wt, mpg))+ # Mapping der Variablen  
 geom\_point() # Auswahl des geoms

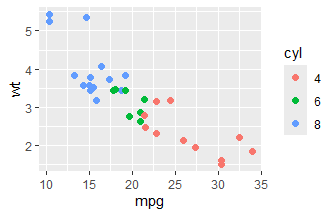


### Informationen aus Drittvariablen addieren

Genauso wie die Farbe können die anderen Attribute bestimmt werden. Wichtig: Wie oben gesagt, können die auf Variablen im Datensatz bezogen sein (🡪 aes() ) oder allgemein gesetzt werden. Wenn letzteres, müssen sie in die geom-function (local mapping, siehe unten).

#### Farbe nach einer kategorialen Drittvariablen

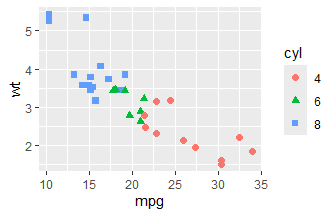
mtcars %>%  
 mutate(cyl = as.factor(cyl)) %>% #nur nötig weil cyl numerisch kodiert ist  
 ggplot(aes(wt, mpg))+  
 geom\_point(aes(colour = cyl))



#### Form nach einer kategorialen Drittvariablen

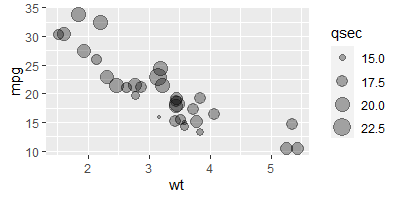
* Analog zu oben geht dies mit dem shape-Argument
* Hier illustrier ich mal Farbe und Form

mtcars %>%  
 mutate(cyl = as.factor(cyl)) %>%   
 ggplot(aes(mpg, wt, color = cyl, shape = cyl)) +  
 geom\_point(size = 2)



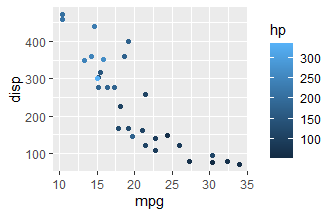
#### Größe der bubbles nach einer kontinuierlichen Drittvariablen

mtcars %>%  
 ggplot(aes(wt, mpg, size = qsec)) +  
 geom\_point(alpha = .3)



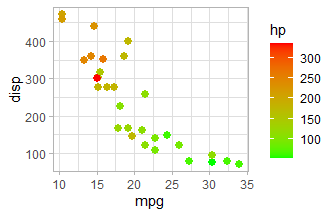
### Farbe nach einer kontinuierlichen Drittvariablen

mtcars %>%   
 ggplot(aes(x=mpg, y=disp, colour=hp) )+  
 geom\_point()



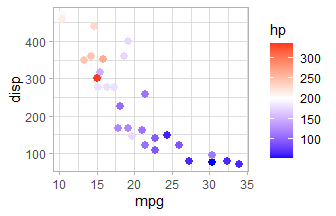
Die **Abstufungen** und Farben kann man frei wählen:

**1. Version: Kontinuierlicher Übergang von einer Farbe zur nächsten**



mtcars %>%   
 ggplot(aes(x=mpg, y=disp) )+  
 geom\_point(aes(colour=hp), size=2.5)+  
 scale\_color\_gradient(low="green",  
 high = "red")+  
 theme\_light()

**2. Version: Wal eines neutralen Mittelpunktes**



(Warum das nicht wirlich

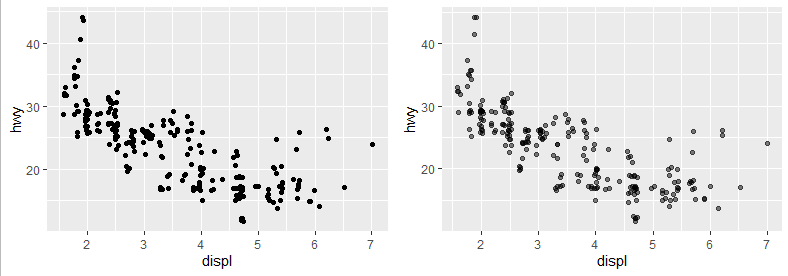
mtcars %>%   
 ggplot(aes(x=mpg, y=disp) )+  
 geom\_point(aes(colour=hp), size=2.5)+  
 scale\_color\_gradient2(low="blue",  
 high = "red",  
 midpoint=200)+  
 theme\_light()

### Overplotting

Bei großen Datensätzen überlappen sich die Punkte, so dass die Masse nicht sichtbar ist. Optionen:

* **Jitter:** Wenn es viele Punkte gibt, sieht man die Masse nicht. Das kann man durch "jitter" ändern. Klappt natürlich nur, wenn es genug sichtbaren Raum gibt (im Massenzentrum ist eh alles schwarz). Aber: Kann man mit alpha (s.u.) kombinieren, um die Punkt noch ewas transparenter zu machen

ggplot(mpg) +  
 geom\_jitter(aes(x = displ, y = hwy))



* **jittering-Ausmaß**: width=.01 und height = .01—d.h. Höhe und/oder Breite!
* **Alternative:** Als Argument in geom\_point():

mpg %>%   
 ggplot(aes(x = displ, y = hwy)) +  
 geom\_point(position = "jitter")

* **Hollow circles (Kreise)**

geom\_point(shape = 1)

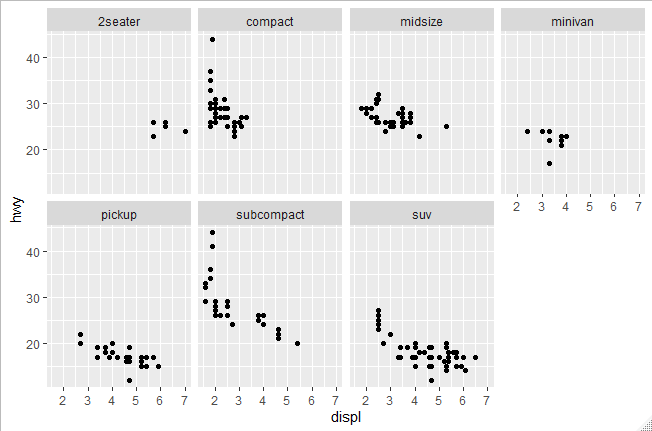
* **Reduktion auf Pixel**

geom\_point(shape = ".")

* **Transparenz ändern** (Alpha-Blending): Wird als **Bruch** („1/10“) oder **Dezimalzahl** gesteuert. Die Zahl im Nenner ist die Anzahl der Punkte, die man überlappen müsste, damit der Punkt schwarz wird.   
  geom\_point(alpha = 1/10)

### Facet wrap: Aufdröseln von Gruppen

ggplot(mpg) +  
 geom\_point(aes(x = displ, y = hwy)) +  
 facet\_wrap(~ class, nrow = 2) #Man beachte die Steuerung durch nrow oder ncol!



* Manchmal unterscheiden sich die Gruppen in den range in Y enorm. Um die pattern zu sehen kann man das Argument scales= "free\_y" in die facet-wrap-Funktion einfügen.
* Will man es noch interaktiver kann man mit facet\_grid(drv ~ cyl) eine 2x2-Kreuzung haben:

ggplot(mpg) +  
 geom\_point(aes(x = displ, y = hwy)) +  
 facet\_grid(drv ~ cyl)

**Faktoren ordnen / Reihenfolge ändern**

* Häufig hat man Faktorenlevel wie PC1, PC2 oder Topic 1 und 2. Mit fct\_inorder() bekommt man sie in die richtige Reihenfolge

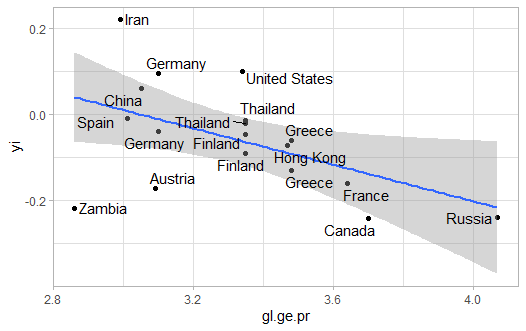
mutate(component = forcats::fct\_inorder(component))

### Punkte labeln

Geht mit geom\_text\_repel aus dem ggrepel-package

Hier ein Beispiel aus der gender-Meta. Zentrale Punkte sind unterstrichen

data %>%   
 ggplot(aes(x = gl.ge.pr, y=yi, label=country))+  
 geom\_point()+  
 geom\_smooth(method="lm")+  
 ggrepel::geom\_text\_repel()+  
 theme\_light()



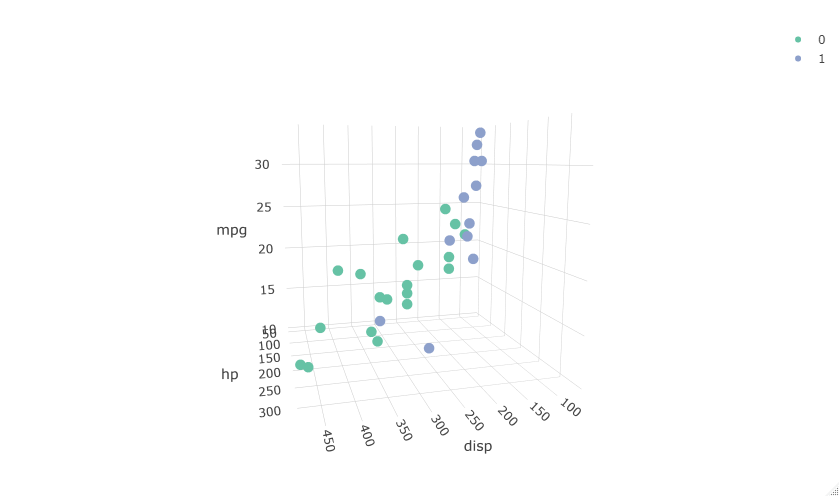
* alternative ist geom\_text(check\_overlap = TRUE, hjust = "inward") was die labels direkt an die Punkte flanscht, was die Zuordnung einfacher macht

### Interaktive 3D plots

* Leider gibt’s keine ggplot-Variante. Die attraktivste Alternative ist plot\_ly aus dem plotly-package

mtcars %>%   
 mutate(am = as\_factor(am)) %>%   
 plot\_ly(  
 x = ~ disp,  
 y = ~ hp,  
 z = ~ mpg,   
 type = "scatter3d",  
 mode = "markers",  
 color = ~ as.factor(am),  
 marker = list(size = 6)) #Größe der Punkte

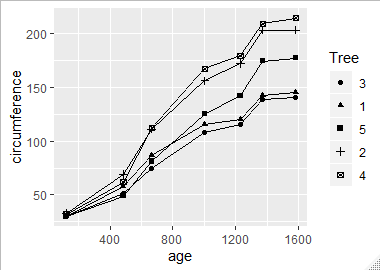
**#Achtung: Die Z-Achse ist die Y-Variable!**



* Achtung: Man kann eine Abszisse ja von beiden Seiten anschauen, daher aufpassen, dass man die "richtige" waehlt (die mit richtiger Richtung er Skalierung)

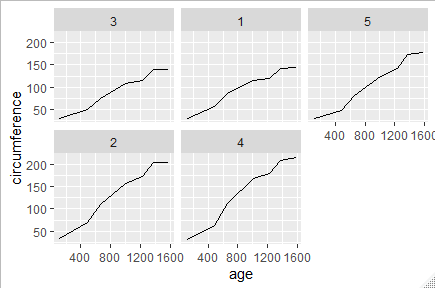
## Linien und Kurven

Orange %>%  
 ggplot(aes(age, circumference, shape=Tree)) +  
 geom\_point() +   
 geom\_line()



Tree ist hier die Gruppierungsvariable, die druch die Punkte-Symbole angezeigt wird. Insgesamt **Möglichkeiten für** **Gruppierungen**:

* shape = <group>
* color = <group>
* linetype = <group>
* facet\_wrap(~ Tree) (🡪 Teilt den plot in mehrere Teilplots. Gut für multi-level-Längsschnittmodelle, wenn das N zu groß ist:



* facet\_grid(row\_variable ~ column\_variable): Kreuzt die Aufteilung von facet\_wrap() noch mal

### Spaghetti plots / growth

* Die sind eine Alternative für das facet-wrap. Hier werden alle Units abgebildet
* Die Daten sind hier in Multi-level-Struktur / long format. Wenn sie im wide format vorliegen, müssen sie mit gather() (siehe im entspr. Abschnitt) umgeschichtet werden. Das geht alles in der selben pipeline
* Beispiel hier: Jede Person („Subject“) hat 6 Größen-Werte im Längsschnitt. Prädiktor „age“ ist zentriert.

# Daten für die Beispiele

data(Oxboys, package = "nlme")  
Oxboys <- as\_tibble(Oxboys)  
Oxboys %>%   
print(n=30)

# A tibble: 234 x 4

Subject age height Occasion

<ord> <dbl> <dbl> <ord>

1 1 -1 140. 1

2 1 -0.748 143. 2

3 1 -0.463 145. 3

4 1 -0.164 147. 4

5 1 -0.0027 148. 5

6 1 0.247 150. 6

7 1 0.556 152. 7

8 1 0.778 153. 8

9 1 0.994 156. 9

10 2 -1 137. 1

11 2 -0.748 139. 2

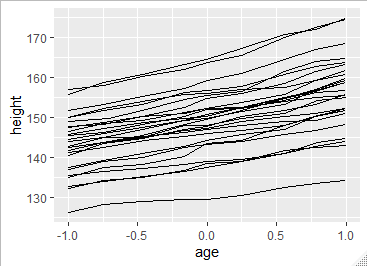
12 2 -0.463 140. 3

13 2 -0.164 143. 4

14 2 -0.0027 143. 5

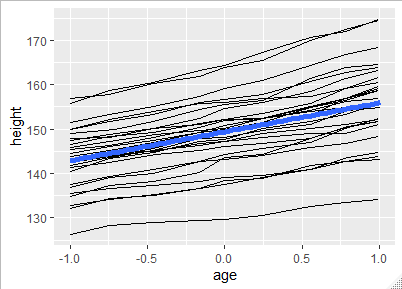
15 2 0.247 144 6

Oxboys %>%  
 ggplot(aes(age, height)) +   
 geom\_line(aes(group = Subject))

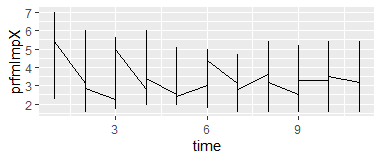


Hinzufügen eines mittleren trends (method=gam als default). Dann sieht man ja ob er linear ist

Oxboys %>%  
 ggplot(aes(age, height)) + geom\_line(aes(group = Subject)) +   
 geom\_smooth(se = FALSE)



**Hinweis:** Wenn der plot so aus sieht, fehlt in geom\_line die Subject ID



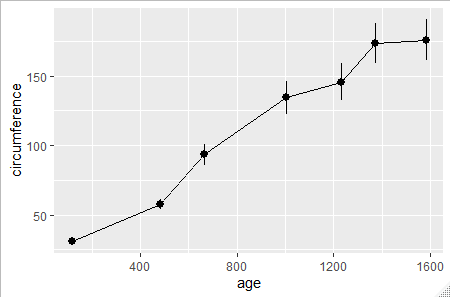
## Darstellen von Statistiken

* Mittelwertsverlauf über das Alter (inkl. SE des Mittelwerts

ggplot(Orange, aes(age, circumference)) +

stat\_summary(funy=mean, geom="line") +

stat\_summary(fun.data=mean\_se, geom="pointrange")

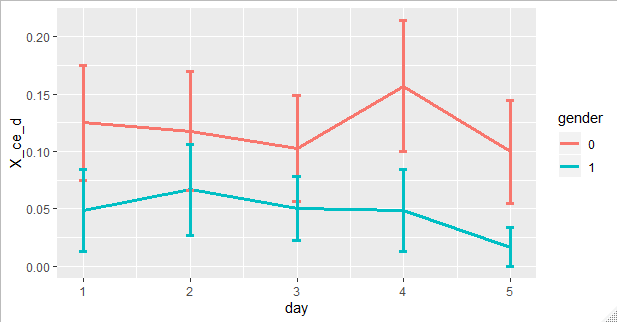


* Die SE kann man auch mit „errorbar“ machen, hier in Kombi mit einer Gruppierungsvariable aus einer Tagebuchstudie:

ggplot(global\_dt, aes(day, X\_ce\_d, color=gender)) +

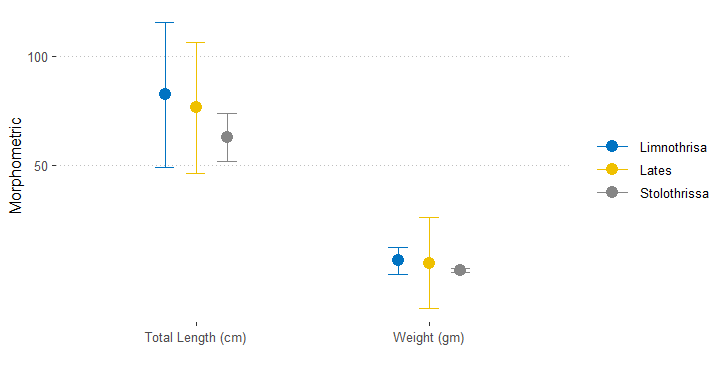
stat\_summary(fun.y=mean, geom="line", size=1.3) +

stat\_summary(fun.data = mean\_se, geom="errorbar", size=1.1, width=.08)



Anpassungen:

* Size = # ändert die Dicke der Linien
* width = # verkleiner die Breite der errorbars
* Weiteres schönes Beispiel



Daten sind unter „z Misc Science and Methods\\dagaa.csv“

* + Hier wird der Mittelwert der Länge und des Gewichts 3er Fischarten geplottet plus die SD.
  + Der clue ist, dass zwei vairablen upper und lower angelegt werden, bei der vom Mittelwert 1 SD addiert oder subtrahier wird. Es geht also nicht direkt mit der SD.
  + Die geilen Farben kommen aus dem package ggsci (siehe die line unten)
  + Die line mit ggpubr:: ist auch sehr nett!

dagaa.clean %>%

pivot\_longer(cols = 2:3, names\_to = "variable", values\_to = "value") %>%

group\_by(species, variable) %>%

summarise(data.mean = mean(value),

data.sd = sd(value),

upper = data.mean+data.sd,

lower = data.mean-data.sd)%>%

ggplot(aes(x = variable, y = data.mean, col = species)) +

geom\_point(position = position\_dodge(.4), size = 4)+

geom\_errorbar(aes(ymin = lower, ymax = upper),

position = position\_dodge(.4), width = .25)+

ggsci::scale\_color\_jco(label = c("Limnothrisa","Lates", "Stolothrissa"))+

ggpubr::theme\_pubclean()+

theme(legend.title = element\_blank(), legend.key = element\_blank(),

legend.key.width = unit(2, "lines"), legend.position = "right")+

coord\_cartesian(expand = TRUE) +

scale\_y\_continuous(name = "Morphometric", breaks = seq(50,350,50))+

scale\_x\_discrete(name = "", labels = c("Total Length (cm)", "Weight (gm)"))

### Einfügen von Regressionsgeraden in scatterplots

Linien und Kurven werden nach Regressionen eingefügt

* method="lm" ist eine Regressionsgerade. Bei method = "rlm" eine robuste Gerade, die weniger durch outlier anfällig ist

Man kann auch mehrere geoms übereinanderlegen--hier den scatterplot und eine Linie/Kurve. Das geht einfach durch Nennung beider geoms:

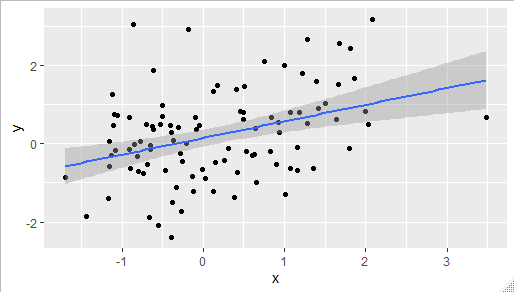
* **Regressionslinie**

data.tb %>%

ggplot(aes(x, y)) +

geom\_point() +

geom\_smooth(method="lm")

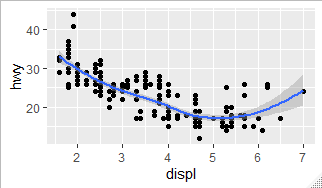


* **Kurve**

ggplot(mpg, aes(x = displ, y = hwy)) +

geom\_point() +

geom\_smooth()

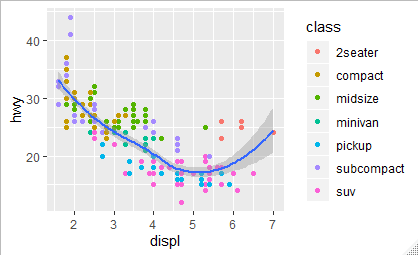


* Bei N<1000 wird die Kurve durch loess geschätzt (local regression), d.h. "method=loess" (default); bei N>1000 ist es ein generalized additive model method="gam". In dem Fall muss man aber eine Formel hinter method = …. Einfügen, die entweder formula = y ~ s(x) or y ~ s(x, bs = "cs")
* Die wiggliness wird durch das Argument „span = .2“ geregelt; Bereich ist 0-1. Default ist 1
* Will man nur die Punkte nach Gruppen aufteilen, aber nicht die Kurve, ist das ein local mapping der Punkte

ggplot(mpg, aes(x = displ, y = hwy)) +

geom\_point(aes(color = class)) +

geom\_smooth()

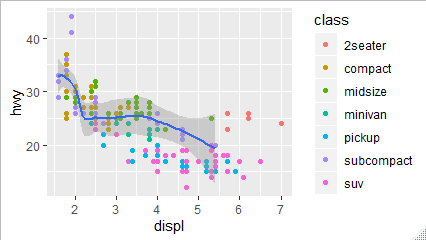


* Man kann auch geoms--abweichend von anderen geoms für andere Subsets erzeugen:

ggplot(mpg, aes(x = displ, y = hwy)) +

geom\_point(aes(color = class)) +

geom\_smooth(data = filter(mpg, class == "subcompact"))



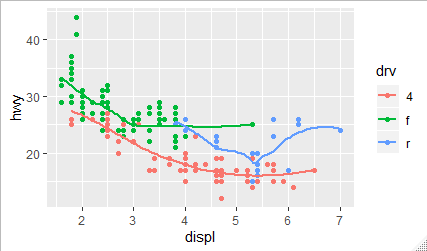
Hier wird die smoothing line nur für die Subgruppe der "subcompact" cars angezeigt. Macht hier nicht wirklich Sinn; es geht aber ums Prinzip

* **Punkte und Kurven für Subgruppen** (🡪 Moderatoreffekte)

ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +

geom\_point() +

geom\_smooth(se = FALSE)



## Barplots

* Es gibt 2 Arten von Barplots
  + **geom\_bar:** Anzeige der Häufigkeiten
  + **geom\_col:** Geht zwar auch mit Häufäigketien (wenn count() vorgeschaltet ist, eignetisch aber besser für die Anzeige von Werten (z.B. Stats)

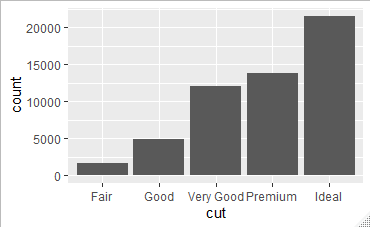
### geom\_bar

Datensatz für die Beispiele:

diamonds <- ggplot2::diamonds

ggplot(diamonds, aes(x = cut)) +

geom\_bar()

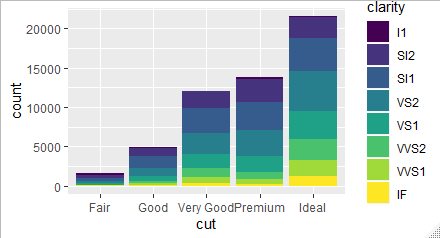


Hier ist cut eine kategoriale Variable.

* **Säulen schachteln nach einer anderen Variablen:**

ggplot(diamonds, aes(x = cut, fill = clarity)) +

geom\_bar()



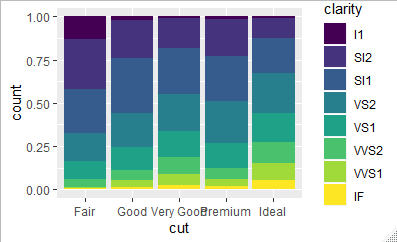
Diese Form kombiniert absolute Häufigkeit mit relativer \*pro Säule\*. Eine Vergleichbarkeit der Säulen bekommt man mit "position=fill", die für jede Säule 100% veranschlagt (hier muss aes, bzw. zumindest position in den geom-part).

diamonds %>%

ggplot() +

geom\_bar(aes(x = cut, fill = clarity),position = "fill")

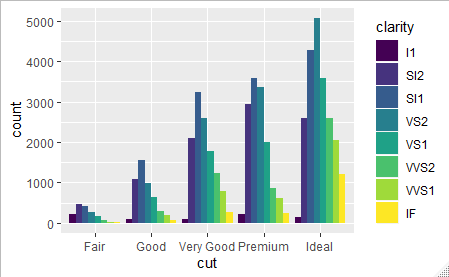
(Achtung: Local mapping—kein globales!)



* **Nebeneinander**

ggplot(diamonds, aes(x = cut, fill = clarity)) +

geom\_bar(position="dodge")



Default für die Y-Achse ist eine Häufigkeitsausgabe (count)

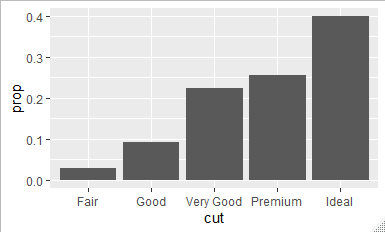
Man kann aber auch andere Dinge ausgeben lassen--auch Funktionen (s. S. 23ff)).

Interessant dürften v.a. **Proportionen** sein

ggplot(diamonds) +

geom\_bar(aes(x = cut, y = ..prop.., group = 1)

)



(Was hier "group = 1" bedeutet, weiß ich nicht--andere Zahlen ergeben dasselbe)

* **Anzeige von Statistiken im plot**

ggplot(top\_N\_SOC(10),

aes(x = reorder(SOC\_NAME, num\_apps), y = num\_apps)) +

geom\_bar(stat = "identity", alpha = 0.9, fill = "blue", width = 0.7) +

coord\_flip() +

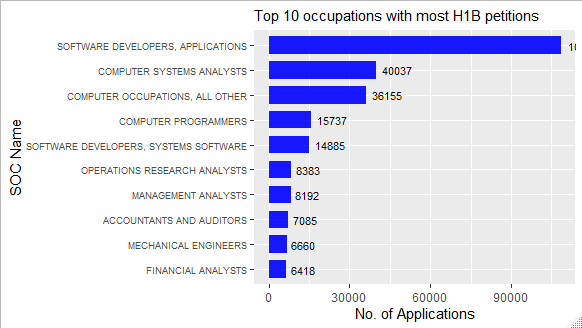
geom\_text(aes(label = num\_apps), hjust = -.2, size = 3) +

ggtitle("Top 10 occupations with most H1B petitions") +

theme(plot.title = element\_text(size = rel(1)),

axis.text.y = element\_text(size = rel(0.8))) +

labs(x = "SOC Name", y = "No. of Applications")



Hier werden ne Masse von Optionen angewandt

* Die Daten sind eine Tabelle, die die Anzahl von Bewerbungen für Berufe enthalten. Durch „(10)“ werden daraus nur die obersten 10 betrachtet
* geom\_bar(stat = "identity", alpha = 0.9, fill = "blue", width = 0.7)
  + stat=“identity“: Die default-Variante zählt einfach die Anzahl der Fälle pro Kategorie, „identity“ führt dazu, dass die y-Werte direkt übernommen werden (wie oben die Anzahl der Bewerbungen)
  + alpha: Wert zwischen 0-1; steuert die Transparenz
  + width= Breite der Balken
* geom\_text(aes(label = num\_apps), hjust = -.2, size = 3)
  + Betrifft die Beschriftungen (Zahl der Bewerbungen)
  + Label=Was für Werte sollten angezeigt werden (🡪 num\_apps; Zahl der Bewerbungen)
  + hjust=Steuert, wie nah die Werte an den Balken sind. Weiß im Moment nicht, ob die absolut (im Raum) oder relativ zu den Balken sind. Einfach rumprobieren
  + size=Größe der Werte
* theme(plot.title = element\_text(size = rel(1)),   
   axis.text.y = element\_text(size = rel(0.8))) +
  + Erste Zeile: Steuert die Größe des Titels („Top 10…“): rel(1) ist wohl Default
  + Zweite Zeile. Steuert die Beschriftung der Y-Achse (Berufsbezeichnung)

### geom\_col

* Das Beispiel zeigt eine Faktorladungsmatrix (pca)

pca %>%

mutate(component = fct\_inorder(component)) %>% #Ordet die PCs !

group\_by(component) %>%

top\_n(6, abs(value)) %>%

ungroup() %>%

mutate(terms = tidytext::reorder\_within(terms, abs(value), component)) %>%

ggplot(aes(abs(value), terms, fill = value > 0)) +

geom\_col() +

facet\_wrap(~component, scales = "free\_y") +

scale\_y\_reordered() +

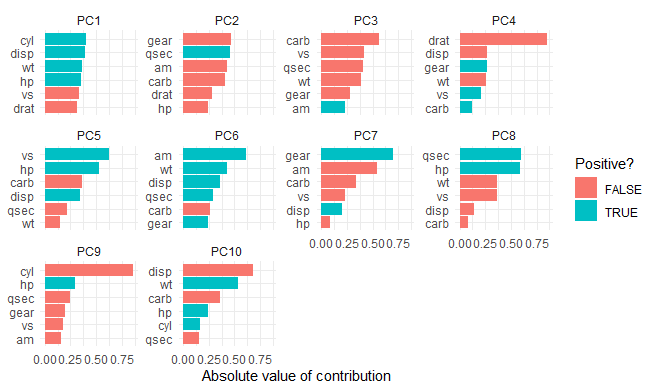
labs(

x = "Absolute value of contribution",

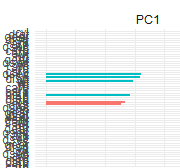
y = NULL, fill = "Positive?"

)+

theme\_minimal()



* Interessant:
  + Anstatt die ursprünglichen Richtungen (positiv /negativ) zu plotten, wird hier der absolute Wert genommen und die Richtung durch Färbung gezeigt
  + fct\_inorder (forcats) ordnet die Faktor-levels im facet-wrap
  + reorder\_within (tidytext) ordnet *innerhalb jedes wraps* die Kategorien nach ihrer Faktorladung
  + theme\_minimal() ist ein nettes theme für den plot
  + Die Zeilen rund um group() dienen nur dazu, für jede PC die top 6 Variablen auszuwählen. Kann man auch weglassen
  + Das free-y-Argument ist wichtig, sonst werden in jedem wrap eine Vielzahl von Kategorien angezeigt—nämlich soviele wie es Kategorie-Ladungs-Kombinationen gibt:



### stat\_summary: Anzeige von Statistiken anstelle Häufigkeiten

Ein schönes Beispiel ist die folgende Funktion, die min, max und Median einer Variablen über die Kategorien zeigt:

ggplot(data = diamonds) +

stat\_summary(aes(x = cut, y = depth),

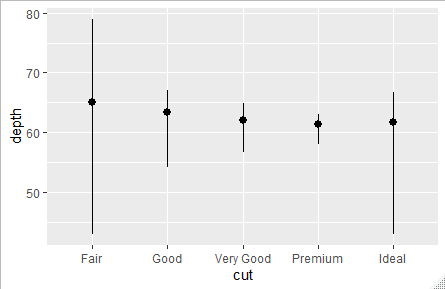
fun.ymin = min,

fun.ymax = max,

fun.y = median

)

Das dürte m.E. identisch sein zu einem gruppierten boxplot...Könnte aber noch andere Anwendungen geben.

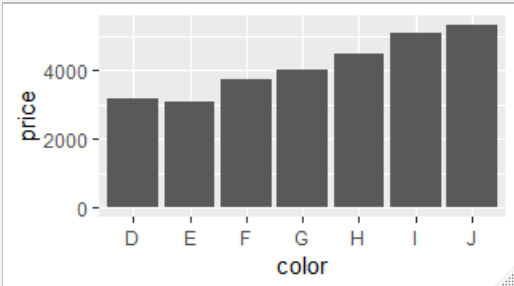


Das kann man auch im bar plot machen:

ggplot(diamonds, aes(color, price)) +

geom\_bar(stat = "summary\_bin", fun.y = mean)

Der Mittelwert des Preises für die verschiedenen Diamanten-Farben („color“) wird angezeigt



## Histogramme und Polygone

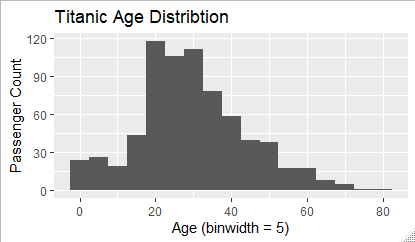
ggplot(titanic, aes(x = Age)) +

geom\_histogram(binwidth = 5) +

labs(y = "Passenger Count",

x = "Age (binwidth = 5)",

title = "Titanic Age Distribtion")



* Die Säulenbreite wird mit dem Argument binwidth = 1 gesteuert
* Alternativ kann man mit bins = 10 die Anzahl der säulen determinieren
* Mit breaks kann man die exakten cuts bestimmen
* Es ist wichtig damit zu experimentieren. Der default-Wert ist simpel und teilt alle Werte in 30 Säulen

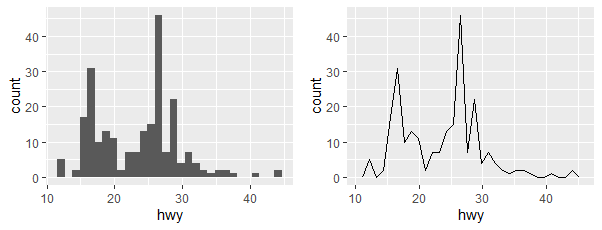
Eine Alternative zu Histgrammen sind frequency-Polygons

p1 = ggplot(mpg, aes(hwy)) + geom\_histogram()

p2 = ggplot(mpg, aes(hwy)) + geom\_freqpoly()

library(gridExtra)

grid.arrange(p1,p2,nrow=1)



### Aufsplittung nach Gruppen

Hier bei gibt es zwei Wege: Das Histogramm der einen Gruppe wird auf das der andren oben drauf gesattelt (Rohversion) oder sie liegen hintereinander (position = „identity“)

library(gridExtra)

* **Version 1: Aufeinander (gestackt)**

stacked <- mtcars %>%

mutate(am=as.factor(am)) %>%

ggplot(aes(x = mpg, fill = am)) +

geom\_histogram(binwidth = 5) +

ylim(0, 10)

* **Version 2: Übereinander (mit Transparenter Darstellung)**

overlapping = mtcars %>%

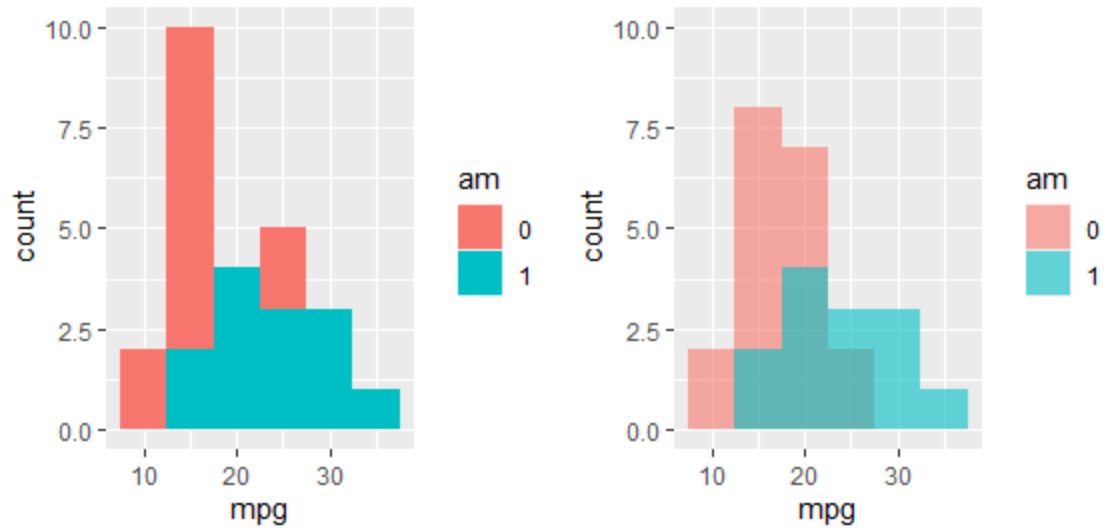
mutate(am=as.factor(am)) %>%

ggplot(aes(x = mpg, fill = am)) +

geom\_histogram(binwidth = 5, position="identity", alpha=.6) +

ylim(0, 10)

grid.arrange(stacked,overlapping, ncol=2)



Links der gestackte—rechts der überlappende

### Aufsplittung nach mehreren Gruppen

ggplot(titanic, aes(x = Age, fill = Survived)) +

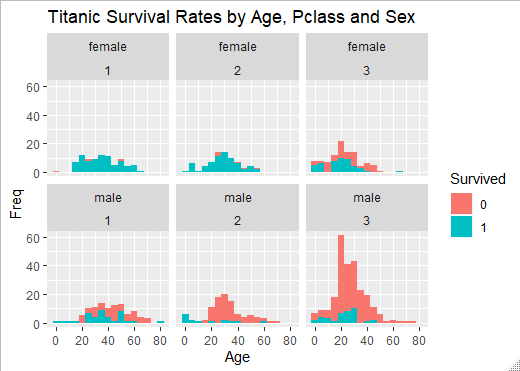
facet\_wrap(Sex ~ Pclass) +

geom\_histogram(binwidth = 5) +

labs(y = "Freq",

x = "Age",

title = "Titanic Survival Rates by Age, Pclass and Sex")



(Achtung: Die Balken sind übereinandergeschachelt, nicht hinter-/voreinander. So haben in der 1. und Klasse nahezu alle Frauen überlebt und nur ganz wenige (die roten) nicht. Vgl. dazu den passenden density-plot unten, der andere Informationen trägt).

## Density plots

* DP sind eine Art „ge-smooth’tes“ Histogramm.
* **Achtung:** Auf der Y-Achse sehen relative Häufigkeiten, nicht absolute. Die Größe jeder Fläche ist für jede Gruppe auf 1 standardisiert. Man kann also bei Gruppenvergleichen nur die Unterscheide in den Verschiebungen auf der X-Achse beurteilen, nicht wie viele Personen den jeweiligen X-Achsen-Wert haben.
* Wickham sagt, dass er die nicht mag, weil sie (wahrscheinlich deshalb) schwer zu interpreti

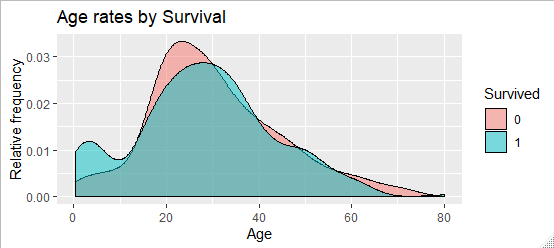
ggplot(titanic, aes(x = Age, fill = Survived)) +

geom\_density(alpha = 0.5) +

labs(y = "Relative frequency",

x = "Age",

title = "Age rates by Survival")



* Alpha ist die Transparenz; je höher um so niedriger
* Zusätzlich zum fill-comman kann man color=<Gruppe> hinzufügen, dann werden die Raänder auch farbig. Size=… steuert die Dicke der Linien

### Aufsplittung nach mehreren Gruppen

ggplot(titanic, aes(x = Age, fill = Survived)) +

facet\_wrap( ~ Sex) +

geom\_density(alpha = 0.5) +

labs(y = "Relative frequency",

x = "Age",

title = "Age rages by survival and Sex")



ggplot(titanic, aes(x = Age, fill = Survived)) +

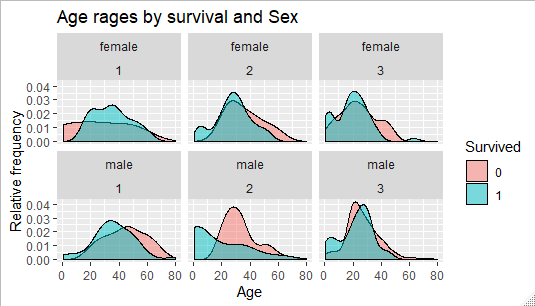
facet\_wrap(Sex ~ Pclass) +

geom\_density(alpha = 0.5) +

labs(y = "Relative frequency",

x = "Age",

title = "Age rages by survival and Sex")



## Ridges plots

* Hierfür müssen die Daten ins longformat verwandelt werden

library(ggridges)

reg\_coefs1 %>%

ggplot(aes(x = trend, y = continent, group = continent)) +

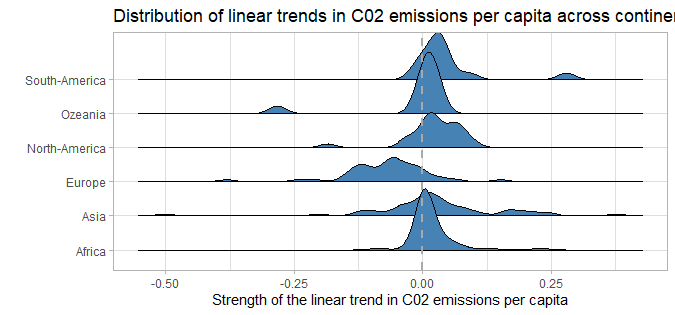
geom\_density\_ridges2(fill="steelblue")+

geom\_vline(xintercept = 0, linetype = "dashed", color="darkgrey", size=1)+

labs(x = "Strength of the linear trend in C02 emissions per capita",

y="",

title = "Distribution of linear trends in C02 emissions per capita across continents")



## Boxplots

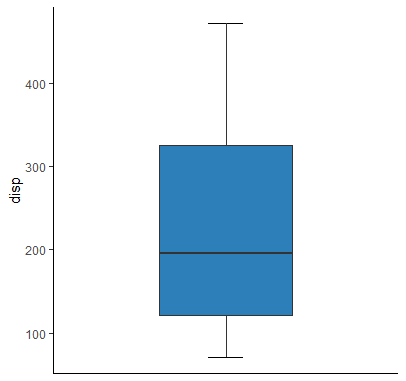
### Einfacher boxplot

mtcars %>%

ggplot(aes(y=disp))+

geom\_boxplot(fill="#2c7fb8") +

scale\_x\_discrete()



* **Anmerkung:** scale\_x\_discrete() macht den boxplot schmaler (weil es die Skalierung der X-Achse, die ja keine Bedeutung hat, entfernt)
* **Optionen:** 
  + **Punkte hinzufügen:** geom\_jitter(width = .2) als weiteres geom
  + **Querlinen an den Antennen:** stat\_boxplot(geom="errorbar", width=.2)+

### Für mehrere Variablen

* Hier musst der Datensatz in longformat umgewandelt werden

data %>%  
 select(contains("english\_")) %>%  
 pivot\_longer(1:4, names\_to = "variable", values\_to = "value") %>%  
 mutate(variable = str\_remove\_all(variable, "english\_")) %>%  
 mutate(variable = case\_when(  
 variable == "lecture\_ba" ~ "Vorlesung Bachelor",  
 variable == "tutorial\_ba" ~ "Übung Bachelor",  
 variable == "lecture\_ma" ~ "Vorlesung Master",  
 variable == "tutorial\_ma" ~ "Übung Master",  
 TRUE ~ variable)) %>%   
 group\_by(variable) %>%   
 filter(value != "NA") %>%   
 ggplot(aes(variable, value))+  
 geom\_boxplot(  
 fill = "steelblue",   
 width = .6,  
 linewidth = .7)+  
 geom\_jitter(  
 width = 0.05,   
 alpha = .2)+  
 coord\_flip()+  
 labs(x="",  
 y="",  
 title = "In welcher Darbietungsform würden Sie Lehrveranstaltungen   
 bevorzugen?") +  
 ylim(labels=c('100% deutsch', '75% deutsch', '50%', '75% englisch', '100%   
 englisch'))

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

# Grafikparameter und Generelles

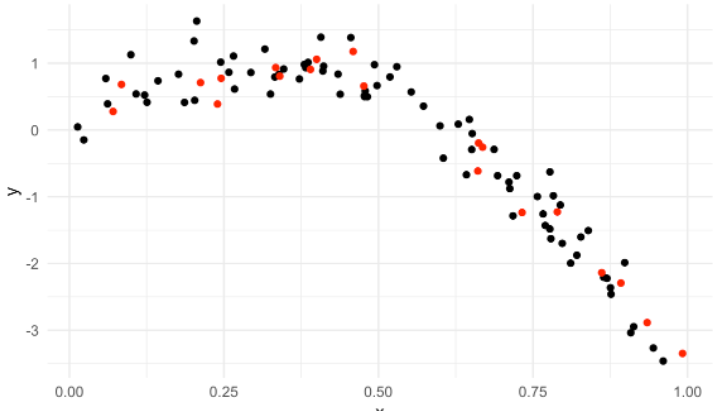
## Zwei Grafiken überlagern

* Szenario: Man hat 2 Datensätze und möchte eine Grafik, in dem beide plots sind
* Geht in dem man einfach in dem man erst eine Standardpipeline anlegt und dann, dass selbe geom wiederholt, aber als Argument "data = name" addiert, z.B.
* (#addieren)

gplot(train\_df, aes(x = x, y = y)) +

geom\_point() +

geom\_point(data = test\_df, color = "red")



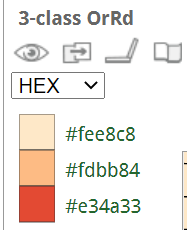
## Farben

* Liste mit coolen Farben: <https://www.datanovia.com/en/blog/awesome-list-of-657-r-color-names/>

Hiermit kann an jede Grafik einfach ein Frabenname angehängt weren (hier mal bei 2 Gruppen)

scale\_color\_manual(values = c("dodgerblue", "firebrick3"))

* Andere Möglichkeit: Colorbrewer (<https://colorbrewer2.org>) und dann die codes in der Funktion antsprechen:   
  scale\_color\_manual(values = c("#74a9cf", "#d7301f"))



Da kann man natürlich Farben aus völlig unterschiedlichen Paletten zusammenstellen

## Manuelle Färbung

* Es gibt zwei Hauptaspeekte bei der Wahl der Farbskala in R:
* **fill vs. color** 
  + fill wird normalerweise für die farbliche Füllung von Elementen wie Balken in einem Balkendiagramm verwendet.
  + color wird für die Farbe von Linien und Umrandungen, wie die Umrisse von Punkten oder Balken, genutzt.
* **Diskrete vs. Kontinuierliche Skalen**
  + Diskrete Skalen (kategorial): Für kategoriale Daten wie Faktoren oder diskrete Variablen.
  + Kontinuierliche Skalen: Für numerische Daten mit einem kontinuierlichen Bereich.

### Diskrete Farbskalen

* **Automatische Zuweisung:** Mit scale\_color\_discrete() (für Umrandungen) oder scale\_fill\_discrete() (für Füllungen) werden den Ebenen eines Faktors automatisch unterschiedliche Farben zugewiesen. Diese Funktionen benötigen keine zusätzlichen Argumente und wählen Farben aus einer vordefinierten Palette.
* **Manuelle Zuweisung:** Um den Faktorlevels manuelle Farben zuzuweisen, verwendet man scale\_color\_manual(values = c("red", "steelblue")) für Umrandungen oder scale\_fill\_manual(values = c("yellow", "green")) für Füllungen. Hier geben Sie eine Liste von Farben an, die den Levels des Faktors entsprechen.

### Kontinuierliche Farbskalen

* Gradientenfarben: Für kontinuierliche Variablen verwendet man Gradientenskalen wie scale\_color\_gradient(low = "blue", high = "red") oder scale\_fill\_gradient(low = "blue", high = "red"). Diese Skalen bilden einen Übergang zwischen zwei Farben, entsprechend dem niedrigsten und höchsten Wert der Variablen.
* Erweiterte Optionen: Es gibt auch komplexere Gradientenfunktionen wie scale\_color\_gradient2() und scale\_color\_gradientn(), die es ermöglichen, mehrere Farben und Zwischenpunkte zu definieren.
* Benennung und Anpassung
* Benennung der Legende: Um die Legende umzubenennen und anzupassen, kann man das Argument name in den Farbskala-Funktionen verwenden, z.B. scale\_color\_discrete(name = "Meine Legende").
* Anpassen der Breaks und Labels: Mit den Argumenten breaks und labels können Sie steuern, welche Werte in der Legende angezeigt werden und wie sie beschriftet sind.

## Achsen

* **Achse ausblenden**

theme(axis.text.x = element\_blank())

* **Limits festlegen**

…lim als weitere Zeile

xlim("f", "r") #kategoriale X-Achse

ylim(20, 30)

*Beispiel*

[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(mpg, wt)) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)() +

xlim(15, 20)

*Alternative:* In dem man einfach einen filter-Befehl vorsetzt kann man bestimmte Einschränkungen der Achsen einnehmen, z.B. sollen hier nur die x-Werte >25 angezeigt werden

delays %>%

filter(x > 25) %>%

ggplot(mapping = aes(x = x, y = delay)) +

geom\_point()

Das geht mit dem Bereichsoperator genauso (%in% c(25:100) )

**Einfacher geht es mit expand\_limits()**

ggplot(mtcars, aes(mpg, wt)) +

geom\_point()+

expand\_limits(y = -5)

* **Ganzzahlige Kategorien**

Manchmal sind die Achsenbeschriftungen Dezimalzahlen. Das kann man ändern mit

scale\_x\_continuous(breaks= 1:10)

Der Befehle fügt mit der Brechstange die Wert 1-10 ein. Wenn die x-Dimension von 1-20 geht, wird ab 11 nix angezeigt

* **Logarithmieren**

scale\_x\_log10()

bzw.

scale\_y\_log10()

* **Addieren einer horizontalen linie**

geom\_hline(yintercept=20, linetype="dashed", color = "red")

* **Addieren einer vertikalen Linie**

geom\_vline(xintercept = 3, linetype="dotted", color = "blue", size=1.5)

* **Bei einem Gruppenvergleich mittels facet.wrap() die Skalen frei variieren lassen**

Manchmal unterscheiden sich die Gruppen in den range in Y enorm. Um die pattern zu sehen kann man das Argument scales= "free.y" in die facet-wrap-Funktion einfügen, z.B.

facet\_wrap(~ country, scales = "free.y"

* **Bei Gruppenvergleichen alle beim selben Y-Wert verankern** (z.B. vor einem facet\_wrap)

Wenn Gruppen unterschiedliche level haben, passt facet\_wrap das wohl an. Schaltet man expand\_limits(y=0) *davor* (also als eigene line im ggplot-pipeline, verankert es alle Teile bei dem gewünschen Wert

* **X-Achsen-Beschriftung um 90° rotieren**

...

geom\_line() +

theme(axis.text.x = element\_text(angle = 90, hjust = 1))

## Text / labels

### Schriftart ändern

#Daten

x = rnorm(100)

y=.4\*x + rnorm(100)

library(tidyverse)

data = tibble(x,y)

# Font definieren (Reicht einmal am Anfang des Skripts)

windowsFonts(A = windowsFont("Times New Roman"))

#Plot (theme\_classic o.Ä. muss vorher kommen)

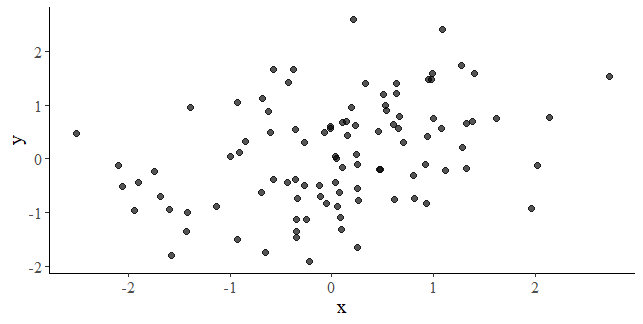
data %>%

ggplot(aes(x,y))+

geom\_point(size=2, alpha =2/3)+

theme\_classic()+

theme(text = element\_text(family = "Times", size = 15)) +



### Unter- und Überschriften

Weiteres Argument in einer eigenen Zeile

geom\_bar() +

labs(y = "Passenger Count",

title = "Titanic Survival Rates",

subtitle ="WTF")

### Gemeinsame Überschrift bei Multiplots mit gridExtra::grid.arrange

gridExtra::grid.arrange(p1,p2, nrow=1, top = "Number of firms")

### Weiterer Text

* + <https://ggplot2-book.org/annotations.html>
  + Durch geom\_text und die x und y-Koordinaten kann man Text addieren
  + Family steuert die Schriftart ("sans", "serif", "mono")
  + Fontface steuert die Formatierung (plain, bold, italic)
  + Daneben kann man nocht die fontsize („size“) kontrollieren—durch eine getrennte „theme“-Zeile (einfach unten dran hängen):

theme(axis.text=element\_text(size=15),   
 axis.title=element\_text(size=13,face="bold"))

### Text rotieren

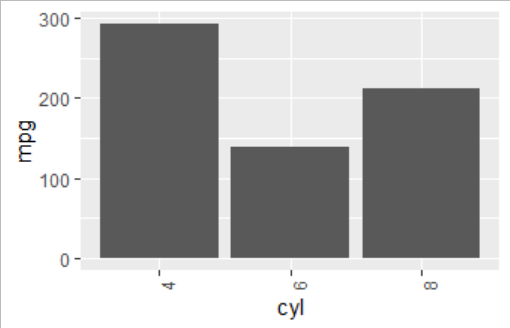
Wenn man Labels an einer X-Achse z.B. schräg oder oder vertial haben möchte:

mtcars %>%

ggplot(aes(x = factor(cyl), y=mpg))+

geom\_col()+

theme(axis.text.x= element\_text(angle = 90, hjust=1))



(Man bemerke das factor(cyl): Cyl ist double—das Umformatieren zm factor geht innerhalb ggplot!)

## Legenden

### Namen der Legende ändern

* Einfach als Teil von labs()

labs(fill = "Bodymass in g") oder

labs(color = "Bodymass in g")

je nach dem, was benutzt wurde um die Legende zu bekommen.

Ein Bild, das Text, Screenshot, Karte, Diagramm enthält.

Automatisch generierte Beschreibung

### Legende entfernen

theme(legend.position = "none")

### Text der Kategorien in der der Legende ändern (Legendenname und Kategorie-Namen

* Welche Funktion man nimmt hängt ab davon, ob die Legende sich auf eine **Färbung** der Umrandung (**\_color\_**) oder farbliche oder größenähnliche **Füllung (\_fill\_)** bezieht
* Es kann so wohl der Name der Legende als auch die Beschriftung der Kategorien verändert werden
* **Umrandung**

scale\_color\_discrete(name = "Type of Retweet",   
 labels = c("Number of conspiracy retweets",   
 "Overall number of retweets"))

* **Füllung**

scale\_fill\_discrete()

* **Änderung von Text und Farbe**

scale\_color\_manual(values = c("#2c7fb8", "#d7301f"),  
 name = "Traffic volume",   
 labels = c("Low", "High"))

* **Änderung des Textes bei Verwendung des size-Arguments**

guides(size = guide\_legend(title = "Sample size"))

* **Titel der Legende ändern**

labs(fill = “new title”)

### Position der Legende ändern

* Einfach nach oben:

theme(legend.position = "top" )

* An eine spezifische Stelle im Plot:

theme(legend.position = c(.3, .85))

* + Linker Wert: Horizontale lage
  + Rechter Wert: Vertikale Lage

## Themes

<https://www.datanovia.com/en/blog/ggplot-themes-gallery/>

* **Mehrere Grafiken nebeneinander oder untereinander:**

p1 = ggplot(mpg, aes(hwy)) + geom\_histogram()

p2 = ggplot(mpg, aes(hwy)) + geom\_freqpoly()

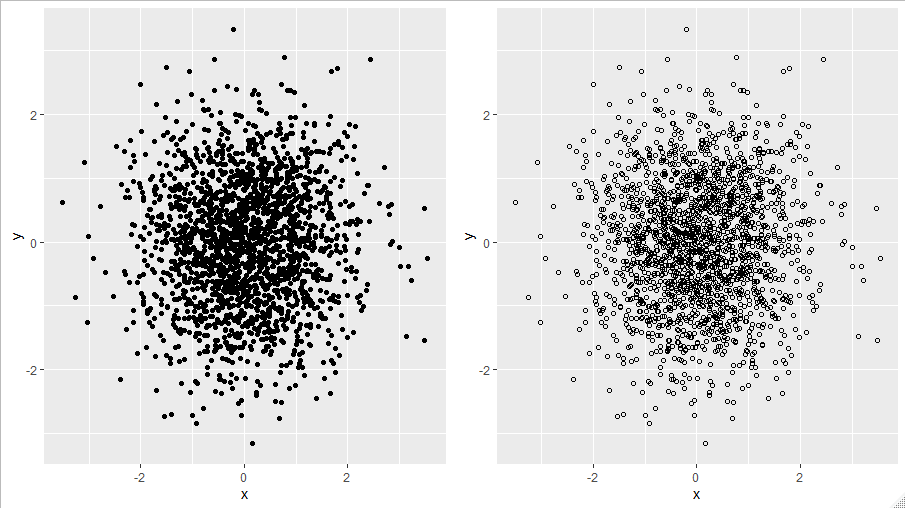
library(gridExtra)

grid.arrange(p1,p2,nrow=1)

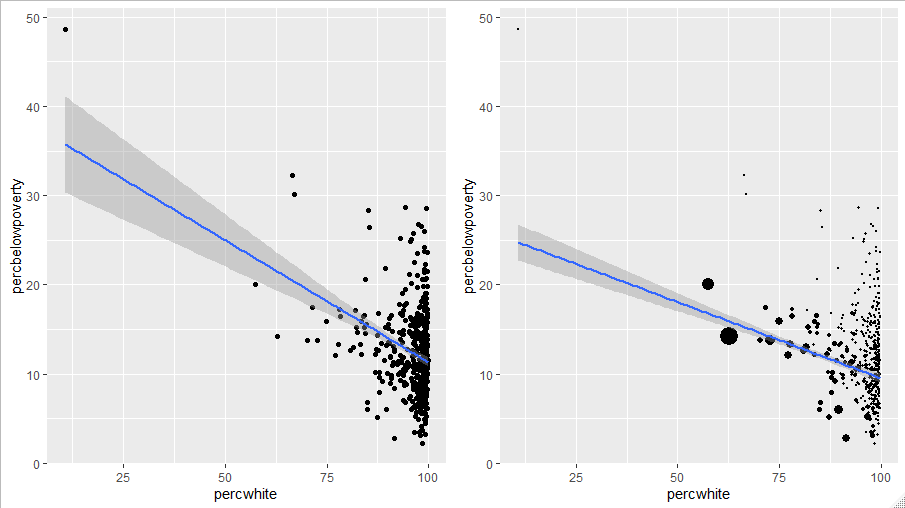
* **Zwei plots neben- oder untereinander**

Dafür braucht man das Paket „gridExtra“. Dann legt man einfach nacheinander zwei plots an und kombiniert sie:

grid.arrange(plot1, plot2, ncol=2) #nrow geht genauso



* **Gewichtung durch Drittvariablen**. Ändert nicht nur den look der Grafik (z.B. Darstellung der bubble-size in scatterplot



Aber nicht nur scatterplots können so gewichtet werden sondern auch Histogramme, density plots, boxplots etc.

## Grafiken speichern

Nach der Generierung des plot (ohne, dass der Plot als Objekt gespeichert wurde):

dev.print(file="test2.png", device=png, width=4700, height=2500, res=500)

Ist ein ziemlicher pain. Ich habs nur hinbekommen, wenn ich extrem große Breiten und Höhenwerte eingesetzt hab (rumprobieren)

# Broom

## Videos

Video1: <https://www.youtube.com/watch?v=7VGPUBWGv6g&t=776s>

**Tutorial** von David Robinson (der Video1 macht):

Robinson, D. (2014). broom: An R package for converting statistical analysis objects into tidy data frames. arXiv preprint arXiv:1412.3565.

## Funktionen

Broom ist dafür da, den output eines Models in einen tibble zu tranformieren. Das passiert mit 3 Funktionen

### tidy()

* Speichert die Koeffizienten in einen tibble. Beispiel

x=rnorm(1000)

y = .5\*x + rnorm(1000)

data = tibble(x,y)

linReg <- lm(y ~ x)

td <- tidy(linReg, conf.int=TRUE)

td

# A tibble: 2 x 7

term estimate std.error statistic p.value conf.low conf.high

<chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 (Intercept) -0.0360 0.0322 -1.12 2.64e- 1 -0.0992 0.0272

2 x 0.514 0.0327 15.7 9.56e-50 0.449 0.578

🡪 Anzahl der Zeilen = Anzahl der Koeffizienten

### augment()

* Speichert fallweise Implikationen (fitted values, residuals, cooks distance) in ein tibble:

augment(linReg, data)

# A tibble: 1,000 x 9

y x .fitted .se.fit .resid .hat .sigma .cooksd .std.resid

<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 -0.736 -2.31 -1.22 0.0821 0.486 0.00649 1.02 7.49e-4 0.479

2 -0.741 -0.987 -0.543 0.0456 -0.198 0.00200 1.02 3.80e-5 -0.195

3 0.231 2.16 1.07 0.0777 -0.841 0.00581 1.02 2.00e-3 -0.828

4 -2.24 0.199 0.0661 0.0329 -2.31 0.00104 1.02 2.68e-3 -2.27

5 -1.77 -1.26 -0.681 0.0522 -1.09 0.00262 1.02 1.51e-3 -1.07

6 0.953 -1.78 -0.952 0.0666 1.90 0.00428 1.02 7.54e-3 1.87

7 -1.58 0.556 0.249 0.0370 -1.83 0.00132 1.02 2.13e-3 -1.79

* + Wie man sieht, werden sie den Rohdaten hinzugefügt (hier x und y)
  + Wenn man den Originaldatensatz in die Funktion einfügt, werden die Residuen etc. dort hinzufügt—wenn nicht, gibt es einen abgespeckten nur mit den Prädiktoren und dem Kriterium.

### glance()

* Speichert schließlich die Gesamt-Modell-Informationen

glance(linReg)

# A tibble: 1 x 11

r.squared adj.r.squared sigma statistic p.value df logLik AIC BIC

<dbl> <dbl> <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl>

1 0.198 0.197 1.02 246. 9.56e-50 2 -1436. 2879. 2894.

# ... with 2 more variables: deviance <dbl>, df.residual <int>

🡪 Anzahl der Zeilen = Anzahl der Modelle

BTW: Die Ergebnisse der tidy(), glance()- und augment()-Funkiton können alle im selben genesteten Datensatz enthalten sein!



Quelle: Wickhams [Video](https://www.youtube.com/watch?v=rz3_FDVt9eg&t=3134s)

## Genestete Daten

Mit den broom-Funktionen gehen auch meist genestete tibbles einher. Die obere Abbildung zeigt eine—in dieser können Daten, Modell-Merkmale, Koefizienten und Residuen enthalten sein.

Eine einfache Methode, einen genesten tibble herzustellen ist einfach über group\_by() + nest(), wobei die Gruppierungsdatei irgendeine Variable sein kann. Beispiel

Daten-Simulation

F = round(runif(1000, 0,2)); x = rnorm(1000); data = tibble(F,x)

data %>%

group\_by(F) %>%

nest()

# A tibble: 3 x 2

# Groups: F [3]

F data

<dbl> <list>

1 1 <tibble [487 x 1]>

2 2 <tibble [262 x 1]>

3 0 <tibble [251 x 1]>

## Nutzen

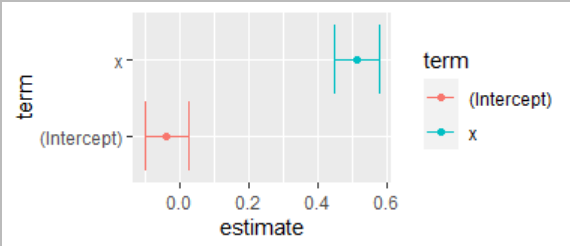
### Veranschaulichung

Damit kann man dann z.B. Modellkoeffizienten grafisch darstellen

ggplot(td, aes(estimate, term, color=term))+

geom\_point()+

geom\_errorbarh(aes(xmin=conf.low, xmax=conf.high))



(Dadurch könnte man auch intuitiv Differenzen zwischen Koeffizienten bzgl. ihrer sign. Differenz ablesen können)

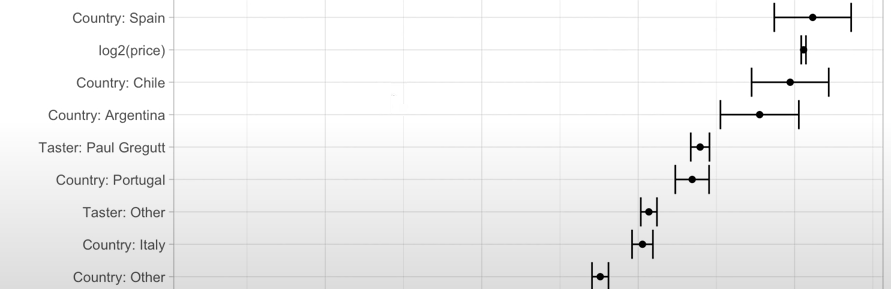
* In diesem Video macht er so einen plot aber verändert mit str\_replace() die Begriff der Prädiktoren, was in dem Fall, wo er dummies benutz sehr schick ist:

<https://www.youtube.com/watch?v=AQzZNIyjyWM> by min. 28:00

Vorher



Nachher



Befehl dazu ist

model %>%

tidy(conf.int = TRUE) %>%

filter(term != " (Intercept)") %>%

mutate(term = str\_replace(term, "country", "Country: "),

term = str\_replace(term, "taster\_name", "Taster: "),

term = fct\_rorder(term, estimate)) %>%

ggplot(.....)

fct\_reorder() sortiert die level eines Faktors entlang der Werte einer anderen Variable, hier wird der Faktor „term“ (der Prädiktor im model-Datensatz, der die Koeff. eines Regressionsmodels enthält) entlang des estimates ge-re-levelt (wobei das Käse ist weil er völlig unterschiedliche Effekte sortier, aber egal).

### Kombinieren und Vergleichen verschiedener Modelle

#### Stacken über Gruppen

🡪 Man kann verschiedene Modelle **stacken**. Hier Beispieldatensatz mtcars; es wird ein Modell für zwei Gruppen gemacht („am“) und gestackt

mtcars %>%

group\_by(am) %>%

do(tidy(lm(mpg ~ wt, .)))

# A tibble: 4 x 6

# Groups: am [2]

am term estimate std.error statistic p.value

<dbl> <chr> <dbl> <dbl> <dbl> <dbl>

1 0 (Intercept) 31.4 2.95 10.7 0.00000000601

2 0 wt -3.79 0.767 -4.94 0.000125

3 1 (Intercept) 46.3 3.12 14.8 0.0000000128

4 1 wt -9.08 1.26 -7.23 0.0000169

Man kann damit auch bootstraps machen (irgendwie mit replicate) und jeden einzelnen grafisch darstellen

[Es gibt noch das paket tidymodels dass Julia Silge oft nutzt. Das ist ein Sammelpaekt, das broom und andere hilfreiche pakete umfasst, siehe [link](https://cran.r-project.org/web/packages/tidymodels/index.html))

#### Stacken über Variablen / Regressionsmodelle

Ziel ist hier, verschiedene Regressionen zu stacken

mtcars %>%

pivot\_longer(names\_to = "predictor", values\_to = "measure", -mpg) %>%

group\_by(predictor) %>%

do(tidy(lm(mpg ~ measure, .)))

# Groups: predictor [10]

predictor term estimate std.error statistic p.value

<chr> <chr> <dbl> <dbl> <dbl> <dbl>

1 am (Intercept) 17.1 1.12 15.2 1.13e-15

2 am measure 7.24 1.76 4.11 2.85e- 4

3 carb (Intercept) 25.9 1.84 14.1 9.22e-15

4 carb measure -2.06 0.569 -3.62 1.08e- 3

5 cyl (Intercept) 37.9 2.07 18.3 8.37e-18

6 cyl measure -2.88 0.322 -8.92 6.11e-10

7 disp (Intercept) 29.6 1.23 24.1 3.58e-21

8 disp measure -0.0412 0.00471 -8.75 9.38e-10

9 drat (Intercept) -7.52 5.48 -1.37 1.80e- 1

10 drat measure 7.68 1.51 5.10 1.78e- 5

11 gear (Intercept) 5.62 4.92 1.14 2.62e- 1

12 gear measure 3.92 1.31 3.00 5.40e- 3

13 hp (Intercept) 30.1 1.63 18.4 6.64e-18

* Mit pivot\_longer wird jedes X-Y-Segment für alles X’s übereinander-gestackt (geht natürlich auch für mehrere Y-Variablen und 1 X oder mehrere X-Y-Kombinationen.
* Predictor ist einfach der Name der ganzen Variablen; measure ist ihr Wert
* Dann wird gruppiert und die einzelnen Regressionen für die gruppierten Segmente berechnet

#### Bootstrapping

Video2 (von J. Silge): <https://www.youtube.com/watch?v=7LGR1sEUXoI>

library(rsample)

set.seed(123)

data\_boot <- bootstraps(data,

times=1e3,

apparant=TRUE)

🡪 Es werden 1000 subsamples gezogen

data\_boot

# A tibble: 1,000 x 2

splits id

<list> <chr>

1 <split [1K/359]> Bootstrap0001

2 <split [1K/389]> Bootstrap0002

3 <split [1K/389]> Bootstrap0003

4 <split [1K/370]> Bootstrap0004

5 <split [1K/359]> Bootstrap0005

6 <split [1K/362]> Bootstrap0006

7 <split [1K/360]> Bootstrap0007

8 <split [1K/381]> Bootstrap0008

Jedes der samples sind 1000 Ziehungen mit Zurücklegen, die Zahl danach ist der „remainder"—dieser Teil sind die Fälle die nicht gezogen wurden. Das ist bei Machine Learning-Ansätzen nötig, weil die Modelle im ersten Teil gefittet und am 2. Teil getestet werden.

Nun wird das Modell auf jeden der splits angewendet. Mittels map() werden die einzelnen Modellergebnisse in den bootstrap-tibble integriert:

boot\_models <- data\_boot %>%

mutate(model = map(splits, ~ lm(y ~ x, data=.)),

coef.info = map(model, tidy))

Map bedeudet, das eine Funktion über „etwas“ wiederholt druchgeführt wird:

* Im ersten Schritt wird wird die lm()-Funktion über die einzelnen splits wiederholt durchgeführt. Ergebnis ist das model. Dies wird boot\_data hinzugefügt
* Im zweiten Schritt wird über die einzelnen models gemaped und die tidy()-Funktion angewendet. Dadurch werden die Koeffizienten boot\_data hinzugefügt. Da das ja mehrere sind, fürht das dazu, dass jeder Eintrag jetzt ein eigenes tibble ist

# A tibble: 1,000 x 4

splits id model coef.info

<list> <chr> <list> <list>

1 <split [1K/359]> Bootstrap0001 <lm> <tibble [2 x 5]>

2 <split [1K/389]> Bootstrap0002 <lm> <tibble [2 x 5]>

3 <split [1K/389]> Bootstrap0003 <lm> <tibble [2 x 5]>

4 <split [1K/370]> Bootstrap0004 <lm> <tibble [2 x 5]>

5 <split [1K/359]> Bootstrap0005 <lm> <tibble [2 x 5]>

6 <split [1K/362]> Bootstrap0006 <lm> <tibble [2 x 5]>

7 <split [1K/360]> Bootstrap0007 <lm> <tibble [2 x 5]>

🡪 Wie coef.info zeigt, ist das ein tibble mit 2 Vairalben (x,y) und 5 Koeffizienten

Durch unnest() werden die jetzt entpackt:

boot\_coefs <- boot\_models %>%

unnest(coef.info)

# A tibble: 2,000 x 8

splits id model term estimate std.error statistic p.value

<list> <chr> <lis> <chr> <dbl> <dbl> <dbl> <dbl>

1 <split [1~ Bootstr~ <lm> (Inte~ 0.0222 0.0308 0.719 4.72e- 1

2 <split [1~ Bootstr~ <lm> x 0.456 0.0313 14.6 1.07e-43

3 <split [1~ Bootstr~ <lm> (Inte~ 0.00698 0.0315 0.221 8.25e- 1

4 <split [1~ Bootstr~ <lm> x 0.464 0.0325 14.3 2.94e-42

5 <split [1~ Bootstr~ <lm> (Inte~ -0.0264 0.0312 -0.846 3.98e- 1

6 <split [1~ Bootstr~ <lm> x 0.459 0.0322 14.2 6.00e-42

Die Zeilen haben sich jetzt verdoppelt. Das ist ein ganz normaler genesteter Datensatz und „id“ ist die ID des bootstraps

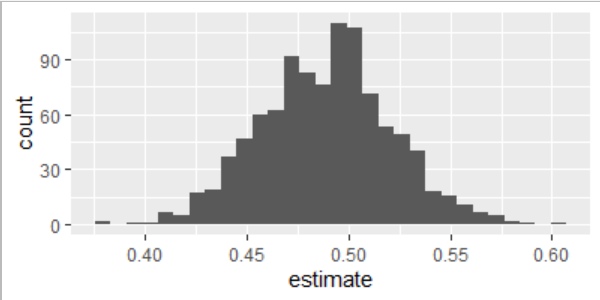
Und mit dem kann man ganz normal arbeiten, z.B. nur die Bs für „x“ extrahieren oder noch was nachschießen (hier ein nettes histogramm der Koeffizienten von x

boot\_coefs 2 %>%

filter(term=="x") %>%

ggplot(aes(x=estimate))+

geom\_histogram()



**Konfidenzintervalle** bekommt man mit

int\_pctl(boot\_models, coef.info)

# A tibble: 2 x 6

term .lower .estimate .upper .alpha .method

<chr> <dbl> <dbl> <dbl> <dbl> <chr>

1 (Intercept) -0.0706 -0.00952 0.0516 0.05 percentile

2 x 0.428 0.489 0.554 0.05 percentile

(Note. Den genesteten Original-bootstrap Datensatz nehmen)

#### Manuelles bootstrapping indirekter Effekte

Hier mal der Code für ein boostrapping eines indirekten Effekts aus zwei hintereinander geschaltetetn OLS-Regressionen. Hintergrund war die Idee, dass man auf die Art auch jedes SEM mittels Regression rechnen können müsste. Ergebnis: Es klappt, aber die SEs (SD der Verteilung) sind nicht 100% identisch mit denen aus lavaan. Für mich aber ausreichend. Müsste aber mal wissen, wie lavaan das intern genau macht

**Daten**

x = rnorm(100)

m = .5\*x + rnorm(100)

y = .5\*m + rnorm(100)

data = tibble(x,m, y)

**Bootstrapping**

library(rsample)

set.seed(123)

data\_boot <- bootstraps(data,

times=1e3,

apparant=TRUE)

**Rechnen der Modelle mit den Einzel-Efffekten**

boot\_models <- data\_boot %>%

mutate(model1 = map(splits, ~ lm(y ~ m, data=.)),

coef.info1 = map(model1, tidy)) %>%

mutate(model2 = map(splits, ~ lm(m ~ x, data=.)),

coef.info2 = map(model2, tidy))

**Unnesten und Multiplikation zum indirekten Effekt**

(das war etwas tricky, weil ich die Variablen in coef.info umbenennen musste)

boot\_models %>%

unnest(coef.info1) %>%

select(splits, id, term, estimate, model2, coef.info2) %>%

rename(term1 = term, estimate1 = estimate) %>%

unnest(coef.info2) %>%

rename(term2 = term, estimate2 = estimate) %>%

select(-std.error, -statistic, -p.value, -model2) %>%

filter(term1 !="(Intercept)" & term2 !="(Intercept)") %>%

select(-term1, -term2, -splits, -id) %>%

mutate(indirect = estimate1\*estimate2) %>%

summarise(mean\_ind = mean(indirect), sd = sd(indirect), z = mean\_ind/sd)