

Data wrangling im Tidyverse

Dr. Holger Steinmetz
Lehrstuhl für Unternehmensführung
Universität Trier
steinmetzh@uni-trier.de

Inhalt

DISCLAIMER:

Das Skript ist im Laufe der Jahre als ein Sammelsurium regelmäßig auftretender Aufgaben entstanden. Es dient v.a. dazu, über das Inhaltsverzeichnis schnell zu entsprechenden Funktionen zu kommen. Es ist nicht als ein zusammenhängender, gut geschriebener Fließtext verfasst.

1	Tibbles als dataframes	4
1.1	Import.....	4
1.2	Im Doc benutzte Daten	5
1.3	Umlaute bewahren und csv als UTF8 speichern	Fehler! Textmarke nicht definiert.
1.4	Über Rstudio auf googlesheets zugreifen.....	5
1.5	Import und Export von Daten.....	6
2	Daten auswählen: filter, select und group_by	6
2.1	Filtern	6
2.2	Select.....	8
2.3	Group_by: Gruppenspezifische Berechnungen	10
3	Deskription: Count, and summarise	11
3.1	Typische Analysen bei einer EDA (exploratory data analysis)	11
3.2	count() und n(): Häufigkeits- und Kreuztabellen	12
3.3	Deskriptive Statistiken	16
3.4	Arrange	19
3.5	Missing data: Analyse und treatment	19
4	Daten verändern: Mutate, rename, separate, unite	20
4.1	Verbinden von mutate mit eigenen Funktionen	21
4.2	Anlegen einer Sequenz.....	21
4.3	Composite bilden	21
4.4	Variablen rekodieren.....	22
4.5	Scientific notation ändern	22
4.6	Veränderungen von Kategorien: fct_lump, fct_reorder & fct_recode.....	23
4.7	Bedingtes Berechnen	24
4.8	Umbenennen	25
4.9	Variablen zerlegen und fusionieren mit separate und unite	25
4.10	Datensätze joinen	27
4.11	pivot_longer() und pivot_wider: Format der Daten ändern	29
4.12	Programming und Funktionen	32
5	Zeit.....	42
5.1	Parsing: Umwandlung in das Format date oder datetime	42
5.2	Extraktion von Teilelementen	45
5.3	Runden.....	46
6	Grafiken	48
6.1	Scatterplot	48
6.2	Linien und Kurven	54
6.3	Darstellen von Statistiken.....	56
6.4	Barplots.....	60
6.5	Histogramme und Polygone	65
6.6	Density plots	67
6.7	Ridges plots	68
6.8	Boxplots	69
7	Grafikparameter und Generelles	71
7.1	Zwei Grafiken überlagern	71
7.2	Farben.....	71
7.3	Achsen	71

7.4	Text und Legenden	73
7.5	Themes	75
7.6	Grafiken speichern	76
8	Broom	77
8.1	Videos	77
8.2	Funktionen	77
8.3	Genestete Daten	78
8.4	Nutzen.....	79

1 Tibbles als dataframes

1.1 Import

Import aus Excel

- readxl package (von Wickham & Bryan)
- Funktion read_excel

```
read_excel("Codebook.xlsx")
```
- Nimmt man die Funktion so wie oben wird das erste sheet eingelesen--mit dem sheet-Argument kann/muss man hintere sheets auswählen:

```
read_excel("test.xlsx", 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 explizit sein, z.B. „C://data//test.xlsx“)

Aus der Zwischenablage

```
library("clipr")  
topic_labels <- read_clip_tbl()
```

Import via vroom

- Anstatt mit readr::read_csv2() kann man mit dem vroom package Daten einlesen. Das ist v.a. bei großen Datensätzen deutlich schneller
- Erkennt auch den delimiter automatisch—das heißt man kann die vroom-Funktion für alle files nehmen!

```
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

Rownames als explite Variablen transformieren

ACHTUNG: Wenn man eine Matrix mit rownames als input hat, geht das Argument

```
as_tibble(rownames="Name")
```

Damit werden die rownames als character/string-Variable eingefügt und mit „name“ überschrieben (es geht auch jeder andere Begriff)

Man kann auch Outputs anderer Funktionen (wenn sie rownames produziere) in ein tibble mit expliziten rownames transformieren—allerdings ist das etws umständlich weil der neue Befehl „rownames_to_column“ einen Dataframe braucht

```
data %>%  
  select(KA, SA, EE, NAG, LW1, CW, CMPL) %>%  
  psych::describe() %>%  
  as.data.frame() %>%  
  tibble::rownames_to_column(var = "variable") %>%
```

```
select(variable, n, min, max, mean, sd, median)
```

Variablen auf einen bestimmten Modus zwingen

Wenn in einer metrischen Variable ein fehlerhafter Eintrag ist (z.B. ein Buchstabe) kann man ihn manuell auf missing setzen, oder beim Import die Variable z.B. auf double zwingen.

```
data <- read_csv2("data.csv",  
  col_types = cols(fer = col_double(),  
                   exa = col_integer() )
```

Ich weiß allerdings nicht, ob man dann alle Variablen des Datensatzes aufführen muss

Daten exportieren

```
write_csv2(semdata, "semdata_neu.csv")
```

1.2 Im Doc benutzte Daten

- **Flights:** Die Daten, die im folgenden genutzt werden, sind aus dem nycflights-Paket. Der Datensatz heißt „flights“. Sie werden durch `library(nycflights13)` automatisch geladen.

1.3 Über Rstudio auf googlesheets zugreifen

```
install.packages("googlesheets4")  
library("googlesheets4")
```

1.3.1 API-Zugriff

Prompted die API. Hier muss man den google-account auswählen und tidyverse Zugriff gestatten
`gs4_auth()`

1.3.2 Zugriff auf sheets

Grundfunktion ist

```
data <- read_sheet(URL)
```

Ein paar Aspekte:

- 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, mann muss das für alle durchdeklinieren

```
df <- read_sheet("...",  
  col_types = 'cccdccccdddddcccccccc')
```

Möglichkeiten für col_types=

- c = character
- d = double
- i = integer
- ? = educated guess (was default ist)

- 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")
```
- Die Daten werden direkt als tibble dargestellt und können mit dplyr-Funktionen bearbeitet werden. Das sheet kann dann mittels write_sheet **überschrieben** werden

```
write_sheet(df, ss = "<URL>", sheet="Study level")
```
- Wenn man deutliche Veränderungen macht, kann man diese in ein neues sheet (innerhalb des selben gesamten googlesheets-Files schreiben

```
write_sheet(df, ss = "<URL>", sheet="test")
```

→ Legt ein neues sheet namens test an

1.4 Import und Export von Daten

- **Import als Excelfile**

```
readxl::read_excel("rawdata.xlsx", "", sheet = "data")
```
- **Export als csv**

```
readr::write_excel_csv2(total_data_rdx, "studycheck.csv")
```

(Hier gibt es natürlich die klassischen write.csv2() und write_csv2(), aber ich hatte mal den Fall, dass ich Text in den Daten hatte und die o.g. Funktion war die einzige, die das UTF8-encoding hinbekam!)

2 Daten auswählen: filter, select und group_by

2.1 Filtern

2.1.1 Positive Auswahl

Nach zwei Bedingungen filtern

```
flights %>%
  filter(month == 1, day == 1)
```

→ Filtert den 01. Januar. Alternativ geht auch „&“

```
flights %>%
  filter(dep_delay > 0 & year == 2013)
```

→ Filtert die Flüge mit einer Verspätung aus dem Jahr 2013. "|" geht genauso für „oder“

Level eines Faktors oder Characters filtern. Hier: Wähle nur die destinations mit „CLT“ aus

```
flights %>%
  filter(dest == "CLT")
```

Mehrere Kategorien eines Faktors filtern

```
flights %>%
  filter(carrier %in% c("AA", "UA")) %>%
  select(1:4, carrier)
```

Nur mal als Veranschulichung, dass es geklappt hat

Mehrere Werte EINER Variable filtern mittels des `%in%`-Operators

```
flights %>%  
  filter(dep_time %in% c(600,605))
```

→ Nur `dep_time = 600` ODER `605`. Alternative wäre, `dep_time` zweimal zu nennen mit `"|"`

Will man das Gegenteil und zig Werte ausschließen, macht man das mit dem Ausrufezeichen vor der Variable:

```
filter( ! dep_time %in% c(600,605))
```

→ Nimmt alle außer 600 und 605

Bereich einer Variable filtern

```
flights %>%  
  filter(dep_time %in% (600:605))
```

Der Doppelpunkt ist der Bereichsoperator

Bestimmte Zeilennummern rausziehen

```
flights %>%  
  slice(1000:1005)
```

Zieht die Zeilen 1000 bis 1005

Kann interessant sein, um sich bestimmte Teile eines tibbles anzuschauen

Kann v.a. interessant sein, innerhalb von Gruppen (v.a. wenn sie groß sind)

Beispielhafte Zeilen rauszufiltern

```
flights %>%  
  group_by(month) %>%  
  slice(1:3)
```

→ Zeigt immer nur die ersten 3 Tage in jedem Monat an.

Alternativ kann man aus jeder Gruppe eine Zufallsauswahl treffen:

```
flights %>%  
  group_by(month) %>%  
  sample_n(3)
```

Und man könnte sich die höchsten 3 Werte in jeder Gruppe ausgeben lassen

```
flights %>%  
  group_by(month) %>%  
  top_n(3, dep_delay)
```

"Zeige in jedem Monat die 3 Flüge mit den krassesten Verpätungen an"

2.1.2 Negativ Auswahl

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

- **Negativ filtern:**

```
flights %>%  
  filter(!dep_time==600) Alle dep_time-Werte ausser 600
```

- **Mehrere Werte rausschmeißen**

```
flights %>%  
  filter(!dep_time %in% c(600,605))
```

- **Einen Bereich rauswerfen**

```
flights %>%  
  filter(!dep_time %in% c(600:605))
```

- **Missings eliminieren**

```
flights %>%  
  filter(!is.na(dep_time))
```

- **Alle löschen, die in X=0 sind**

```
data %>%  
  filter(x !=0)
```

- **Zufallsstichprobe aus dem Datensatz ziehen**

```
mtcars %>%  
  slice_sample(n=10) #Achtung, „n“ ist essential, weil es ja auch prop sein kann  
  
mtcars %>%  
  slice_sample(prop = .5)
```

Geht auch pro Gruppe (→ zieht 50% aus jeder Gruppe vs)

```
mtcars %>%  
  group_by(vs) %>%  
  slice_sample(prop = .5) %>%  
  ungroup()
```

2.2 Select

2.2.1 Basis

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

```
flights %>%  
  select(1:4, carrier, dep_time:dep_delay)
```

Selektiert die ersten 4 Variablen plus "carrier" plus einem Bereich

Reihenfolgen ändern: Einfach Variablen in der gewünschten Reihenfolge auflisten

```
flights %>%  
  select(dep_time, month, carrier:tailnum)
```

Will man nur ein paar nach vorne holen und den Rest lassen geht „everything()“

```
flights %>%  
  select(time_hour, air_time, everything())
```

Negativ-Auswahl

```
flights %>%  
  select(-month, -day)
```

Auch das geht mit Bereichsoperator

Bereich von Variablen auswählen

```
flights %>%  
  select(month:dep_delay)
```

Bereich von Variablen ausschließen

```
data %>%  
  select(-(variable_a:variable_d) )
```

2.2.2 Auswahl aufgrund von Namens-Elementen

Ausschluss von Variablen, die ein Wortstamm haben

```
flights %>%  
  select(-contains("time"))
```

Schmeißt alle raus, die "time" enthalten. Mehrere Begriffe gehen mit c():

```
-contains(c("min", "max"))
```

```
flights %>%  
  select(contains("Delay"))
```

Wählt alle Variablen, die im Namen "Delay" haben

```
permits_raw %>%  
  select(-ends_with("change"))
```

Schmeißt alle Variablen raus, die den Wortstamm change haben.

Analog: Anfangs-Teile eines Namens selektieren

```
permits_raw %>%  
  select(starts_with("fl"))
```

2.2.3 Nach Variableneigenschaften selektieren

- Geht mit `select_if()`
- Z.B. Variablen nach Klasse selektieren

```
flights %>%  
  select_if(is.numeric)
```

Analog `is.factor`, `is.character`

2.2.4 Sonstiges

- **Auswahl von unique-Werten** einer Variable oder der Kombination

```
flights %>%  
  select(origin,dest) %>%  
  distinct()
```

Zeigt alle Kombinationen von Abflug und Ziel an (--> Es gibt 215 Kombinationen)

Geht mit `unique()` auch, `distinct` sei aber effizienter.

- **Identifizieren von Duplicates:** Geht mit `get_dupes()` aus dem janitor package. **Dies listet alle duplicates auf**

```
mtcars %>%  
  slice(1:5) %>%  
  bind_rows(mtcars) %>%  
  janitor::get_dupes()
```

Man kann mit `get_dupes()` auch bestimmte Variablen ansprechen, z.B. `get_dupes(mpg, cyl, disp)`

- **Duplicates automatisch löschen:**

- `distinct()` löscht alle identischen Zeilen (bzgl. aller Variablen im Datensatz)

```
mtcars %>%  
  slice(1:5) %>%  
  bind_rows(mtcars) %>%  
  distinct()
```

- Will man die Variablen bestimmen, hinsichtlich derer die Duplicates gelöscht werden (wobei mir gerade kein Anwendungsfall einfällt, dann muss „.keep_all“ rein, sonst löscht es nicht nur die Dups sondern auch die nicht-benannten Variablen)

```
mtcars %>%
  slice(1:5) %>%
  bind_rows(mtcars) %>%
  distinct(mpg, cyl, disp, .keep_all = TRUE)
```

- Nimmt man nur `distinct()` ohne Argument, schmeißt R **“duplicate rows”** raus!
- `janitor::get_dupes()` soll dasselbe machen:
`janitor::get_dupes(data, id)`

- **select(everything())**

`everything()` ist eine Bezeichnung, die alle Variablen im tibble betrifft. Gut, wenn man z.B. zwei Variablen nach vorne bringen möchte und anschließend den Rest (Reihenfolge ändern):

```
flights %>%
  select(time_hour, air_time, everything())
```

2.3 Group_by: Gruppenspezifische Berechnungen

`group_by` kann man in eine pipe einbauen; man kann aber auch einen neuen Datensatz mit `group_by` generieren, der identisch aussieht wie das Original—aber für jede Operation, die man damit ausführt, gruppierten output ausgibt.

Analysen für Subgruppen

```
flights %>%
  group_by(month) %>%
  summarize(Mean=mean(dep_delay, na.rm=TRUE))
```

Hier mal einfach der Mittelwert der Verspätungen für die Monate "Mean=" ist optional und fügt eine Überschrift hinzu

Hier mit mehreren Statistiken

```
flights %>%
  group_by(month) %>%
  summarize(Mean=mean(dep_delay, na.rm=TRUE),
            Min=min(dep_delay, na.rm=TRUE),
            Max=max(dep_delay, na.rm=TRUE),
            Range=(max(dep_delay, na.rm=TRUE)-min(dep_delay, na.rm=TRUE)))
```

Bemerkenswert: Innerhalb des commands wird eine neue Variable (`range`) erzeugt

- **Kombinationen von Gruppen**

```
flights %>%
  group_by(month, day) %>% #Kombination → Zellenmittelwerte
  summarize(Mean=mean(dep_delay, na.rm=TRUE))
```

- **Regression in einer pipeline**

```
data %>%
  lm(edf~ series_length, .) %>%
  summary(.)
```

3 Deskription: Count, and summarise

3.1 Typische Analysen bei einer EDA (exploratory data analysis)

- **Summary table aller Variablen**

```
data %>%
  select(KA, SA, EE, NAG, LW1, CW, CMPL) %>%
  psych::describe() %>%
  as.data.frame() %>%
  tibble::rownames_to_column(var = "variable") %>%
  select(variable, n, min, max, mean, sd, median)
```

variable	n	min	max	mean	sd	median
KA	149	1	3.857143	2.247363	0.7174891	2.285714
SA	149	1	5.000000	3.313199	0.8640537	3.333333
EE	149	1	5.000000	2.753915	1.0210958	2.666667
NAG	149	1	5.000000	2.008949	0.9299378	2.000000
LW1	149	1	5.000000	2.805369	0.9276376	3.000000
CW	149	1	5.000000	2.979866	1.0524733	3.000000
CMPL	149	1	5.000000	2.177852	0.9727257	2.000000

- **Einfache Häufigkeiten:** `count(x, sort=TRUE)`
- Zwischendurch immer auch mal wieder über `View()` die Daten anschauen. Dies geht ganz nett durch `data %>% View()`
- 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

```
-- Data Summary -----
Name                               Values
Number of rows                     32
Number of columns                   11

Column type frequency:
  numeric                           11

Group variables                     None

-- Variable type: numeric -----
# A tibble: 11 x 11
  skim_variable n_missing complete_rate mean    sd    p0    p25    p50    p75   p100 hist
  <chr>          <int>         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
1 mpg            0             1  20.1   6.03  10.4  15.4  19.2  22.8  33.9  
2 cyl            0             1   6.19   1.79   4     4     6     8     8  
3 disp          0             1 231.   124.   71.1  121.  196.  326  472  
4 hp            0             1 147.   68.6   52    96.5  123   180  335  
5 drat          0             1   3.60   0.535  2.76   3.08   3.70   3.92  4.93  
6 wt            0             1   3.22   0.978  1.51   2.58   3.32   3.61  5.42  
7 qsec          0             1  17.8   1.79  14.5  16.9  17.7  18.9  22.9  
8 vs            0             1   0.438  0.504   0     0     0     1     1  
```

9	am	0	1	0.406	0.499	0	0	0	1	1	
10	gear	0	1	3.69	0.738	3	3	4	4	5	
11	carb	0	1	2.81	1.62	1	2	2	4	8	

Alternative zu `str()`

`glimpse(flights)`

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

Deskriptive Statistiken mit nettem Histogramm (paket `skimr`)

```
flights %>%
  select_if(is.numeric) %>%
  skimr::skim()
```

Anm. `p0` = min, `p100`=max

3.2 count() und n(): Häufigkeits- und Kreuztabellen

3.2.1 Count

- **Einfache Häufigkeitstabelle**

```
mtcars %>%
  count(cyl, sort=TRUE, name="No of cars")
```

Anzahl der Fälle (hier: verschiedene Auto-Zylinder); sortiert nach Anzahl

	cyl	No of cars
1	8	14
2	4	11
3	6	7

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

- **X aggregieren**

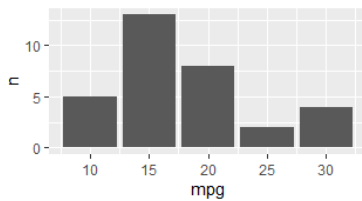
Man kann das Ausmaß der Aggregation innerhalb des `count`-Befehls vergrößern, was v.a. bei vielen differenzierten Werten sinnvoll ist. Hier ist `mpg` eine numerische Variablen mit einer Dezimalstelle. Das kann man z.B. auf 5 Kategorien aggregieren:

```
mtcars %>%
  count(mgp = 5*(mpg %/% 5))
      mpg      n
<dbl> <int>
1    10       5
2    15      13
3    20       8
4    25       2
5    30       4
```

Dann kann man auch noch schnell ein `geom_col()` oder `geom_line()` dranhängen:

```
mtcars %>%
```

```
count(mpg = 5*(mpg %/% 5)) %>%
  ggplot(aes(mpg, n)) +
  geom_col()
```



(mehr zu `geom_col` unter Grafiken)

- **Original-Datensatz bewahren mit `add_count()`**
- Der einfache `count`-Befehl aggregiert die Daten. Dies will man oft nicht und kann es mit `add_count` verhindern. Der Befehl legt eine `n`-Variable an, die die Anzahl enthält und für jeden analogen Fall einfach wiederholt. Hier am Beispiel `cyl`

Mit `count()`:

```
mtcars %>%
  count(cyl)
```

```
cyl      n
<dbl> <int>
1      4    11
2      6     7
3      8    14
```

vs. mit `add_count`

```
mtcars %>%
  add_count(cyl) %>%
  select(mpg, cyl, drat, n)
```

```
# A tibble: 32 x 4
mpg   cyl  drat     n
<dbl> <dbl> <dbl> <int>
1    21     6   3.9     7
2    21     6   3.9     7
3    22.8   4   3.85    11
4    21.4   6   3.08     7
5    18.7   8   3.15    14
6    18.1   6   2.76     7
```

etc.

➔ Der Datensatz bleibt erhalten!

- **Double counts**

Man kann an einen `count`-Befehl einen weiteren (`count(n)`) anschließen. Dieser zählt dann, wieviel die im ersten Schritt gezählten Häufigkeiten vorkommen

Beispiel: Häufigkeiten der PS-Zahlen im `mtcars`-Datensatz (single count)

```
mtcars %>%
  count(hp, sort=TRUE)
```

```
# A tibble: 22 x 2
```

	hp	n
	<dbl>	<int>
1	110	3
2	175	3
3	180	3
4	66	2
5	123	2
6	150	2
7	245	2
8	52	1

Beim double count wird gezählt, wie oft die 1er, 2er und 3er-Häufigkeiten vorkamen

```
mtcars %>% count(hp, sort=TRUE) %>% count(n)
```

	n	nn
	<int>	<int>
1	1	15
2	2	4
3	3	3

- **Bivariate Häufigkeits- oder Kreuztabelle**

Hier: Kreuzung von Zylinder-zahl und Getriebe (0 = Automatik, 1 = Manuell)

```
mtcars %>%
  count(cyl, am, sort=TRUE) %>%
  spread(cyl, n)
```

Die in spread zuerst genannte Variable bestimmt die Spalten, was man sieht, dass „am“ die Spaltenbezeichnung der ersten Spalte ist

	# A tibble: 2 x 4			
	am	`4`	`6`	`8`
	<dbl>	<int>	<int>	<int>
1	0	3	4	12
2	1	8	3	2

- **Prozentwerte in Kreuztabellen**

Das hab ich aus dem Nobelprize Video von Dave Robinson

Ziel: Eine Kreuztabelle aus dem Getriebe (am) und der Zylinderzahl (cyl) und dann zu berechnen, wieviel Prozent der automatik-Autos welche Zylinderzahl haben. Angehängt noch ein schicker bar_col-Plot

```
mtcars %>%
  count(am, cyl) %>%
  group_by(am) %>%
  mutate(percent = n / sum(n) )
```

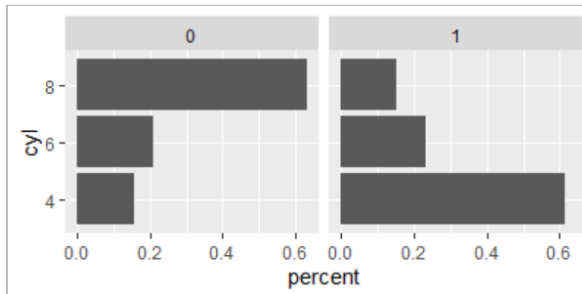
cyl wird vorher noch zum Faktor gemacht (braucht man nicht, sieht aber im plot besser aus)

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

Inkl. Plot:

An die o.g. code anschließen

```
%>%  
  ggplot(aes(cyl, percent)) +  
  geom_col() +  
  facet_wrap(~am) +  
  coord_flip()
```



60% der Automatik-Autos sind 8-Zylinder; bei den Gang-getriebenen sind das nur 20%

Schicker (inkl. Prozentwerten) geht es mit der `tabyl()`-Funktion aus dem `janitor` package ([hier](#) ein kleines tutorial)

```
tbl.tmp <- data %>%  
  tabyl(edct, treatment) %>%  
  adorn_percentages("col") %>%  
  adorn_pct_formatting(digits = 2) %>%  
  adorn_ns()
```

edct	0	1
Kein Schulabschluss	0.00% (0)	0.00% (0)
Hauptschulabschluss	1.06% (1)	2.20% (4)
Realschulabschluss	8.51% (8)	8.24% (15)
(Fach-)Abitur	40.43% (38)	25.82% (47)
Abgeschlossenes Studium	27.66% (26)	52.75% (96)
Promotionsstudium	2.13% (2)	1.65% (3)

Funktion sieht kompliziert aus. Der Kern ist aber erst mal `tabyl(x,y)`, der Rest addiert nacheinander, die Prozentinformation, die Prozent-Schreibweise (%) und die absoluten Zahlen in Klammern

3.2.2 Zählen in Gruppen

- `summarise(sum_miss = sum(is.na(x)))`: Fehlende Werte zählen (bzw. mit `!is.na()` die nicht-fehlenden)
- `summarise(n_distinct(variable))`: Zählen der unique Elemente

Beispiel: Wieviele Getriebe-Varianten gibt es für die verschiedenen Zylinder:

```
mtcars %>%
```

```
group_by(cyl) %>%
  summarise(gear_per_cyl = cars / n_distinct(am))
  cyl gear_per_cyl
  <dbl>         <int>
1      4           2
2      6           2
3      8           2
```

Antwort: Jeweils zwei (Automatik, manuell)

- **Anzahl der Fälle in Gruppen zählen**

```
summarise(n = n())
```

- `n_groups()`: Anzahl der Gruppen

3.3 Deskriptive Statistiken

3.3.1 Grundform

```
flights %>%
  summarise(M = mean(arr_delay, na.rm=TRUE))
```

Achtung: Wenn die Fehlermeldung "argument "by" fehlt" kommt, hast du „summarize“ (mit z) geschrieben!

3.3.2 Summentabellen

Aufsummierung einer Variable (hier Distanz) über zwei gekreuzte andere Variablen

```
flights %>%
  group_by(month, carrier) %>%
  count(wt=distance) %>%
  spread(carrier, n)
```

(wt = weighted tally; Keine Ahnung, was das macht (?))

3.3.3 Bedingte summaries

In den summary-Befehl kann man einen filter einfügen.

```
data %>%
  group_by(type) %>%
  summarise(prayer = sum(prayer == "yes"))
```

Das hatte Dave in einem TT Video wo type das Dessert von Leuten war. Er wollte den Zusammenhang mit Gebeten rausfinden

3.3.4 Summaries über Variablen

```
mtcars %>%
  summarise(across(displ:wt, mean, na.rm=TRUE))
```

```
      displ      hp      drat      wt
1 230.7219 146.6875 3.596563 3.21725
```

3.3.5 Mittelwertstabellen

Für mehrere Variablen gleichzeitig

```
flights %>%  
  summarise_at(vars(dep_time, arr_delay), mean, na.rm = TRUE)
```

Bereichsoperator geht auch:

```
flights %>%  
  summarise_at(vars(dep_time:arr_delay), mean, na.rm = TRUE)
```

Nach Charakteristika der Variablen

```
flights %>%  
  summarize_if(is.numeric, mean, na.rm=TRUE) %>%
```

→ Gibt für alle numerischen Variablen den mean aus

Kreuztablen mit Mittelwerten

Hier werden Klasse der KFZ und ihre Zylinder-Anzahl gekreuzt. Dann Mittelwerte von cty (was immer das auch ist), berechnet; so gekreuzt, dass die Zylinder die Spalten sind (weil das weniger sind)

```
mpg %>%  
  group_by(class, cyl)%>%  
  summarise(mean_cty=mean(cty))%>%  
  spread(cyl, mean_cty)
```

	class	`4`	`5`	`6`	`8`
1	2seater	NA	NA	NA	15.4
2	compact	21.4	21	16.9	NA
3	midsize	20.5	NA	17.8	16
4	minivan	18	NA	15.6	NA
5	pickup	16	NA	14.5	11.8
6	subcompact	22.9	20	17	14.8
7	suv	18	NA	14.5	12.1

3.3.6 Korrelationen

```
flights %>%  
  select(dep_delay, distance, dep_time) %>%  
  cor(., use="pair") %>%  
  round(3)
```

Wenn man einen Faktor drin hat

```
data.tb %>%  
  select(x,y,f1,f2) %>%  
  mutate(f1=as.numeric(f1),f2=as.numeric(f2)) %>%  
  cor(., use="pair") %>%  
  round(3)
```

- Die tidymodels Version ist correlate()

```
library(corr)
```

```
cor_mat <- mtcars %>%  
  as_tibble() %>%
```

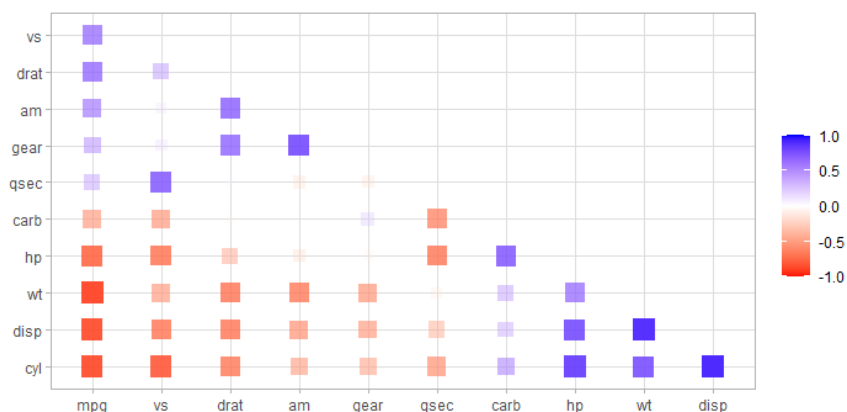
```
correlate() %>%
rearrange() %>% #Sortieren nach Höhe
shave() #Eliminieren der upper triangle
```

(In correlate ist der default Pearson & pairwise. Es gibt aber auch Kendall und Spearman)

term	mpg	vs	drat	am	gear	qsec	carb	hp	wt	disp	cyl
<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1 mpg	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
2 vs	0.664	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
3 drat	0.681	0.440	NA	NA	NA	NA	NA	NA	NA	NA	NA
4 am	0.600	0.168	0.713	NA	NA	NA	NA	NA	NA	NA	NA
5 gear	0.480	0.206	0.700	0.794	NA	NA	NA	NA	NA	NA	NA
6 qsec	0.419	0.745	0.0912	-0.230	-0.213	NA	NA	NA	NA	NA	NA
7 carb	-0.551	-0.570	-0.0908	0.0575	0.274	-0.656	NA	NA	NA	NA	NA
8 hp	-0.776	-0.723	-0.449	-0.243	-0.126	-0.708	0.750	NA	NA	NA	NA
9 wt	-0.868	-0.555	-0.712	-0.692	-0.583	-0.175	0.428	0.659	NA	NA	NA
10 disp	-0.848	-0.710	-0.710	-0.591	-0.556	-0.434	0.395	0.791	0.888	NA	NA
11 cyl	-0.852	-0.811	-0.700	-0.523	-0.493	-0.591	0.527	0.832	0.782	0.902	NA

- Grafik damit

```
cor_mat %>%
rplot(shape = 15, colours = c("red", "white", "blue")) +
theme_light()
```



3.3.7 Statistiken für mehrere Variablen ausgeben

```
flights %>%
summarize_each(funs(mean(.,na.rm=TRUE)), dep_delay, distance)
```

Wendet die Funktion mean auf die aufgeführten Variablen an

Dem kann man auch einen group_by-Befehl vorschalten

3.3.8 Mehrere Statistiken

```
flights %>%
summarize_each(funs(mean(.,na.rm=TRUE), sd(.,na.rm=TRUE)), dep_delay)
```

3.3.9 Gruppierte Statistiken mit group_by

```
flights %>%
group_by(year,month,day) %>%
summarize(delay = mean(dep_delay, na.rm = TRUE))
```

→ Man bekommt die mittleren Verspätungen für jeden Tag

3.4 Arrange

```
flights %>%  
  arrange(year, month)
```

Sortiert zuerst nach Jahr, dann nach Monat

Reihenfolge umkehren

```
flights %>%  
  arrange(desc(Month))
```

Alternative

```
flights %>%  
  arrange(-Month)
```

3.5 Missing data: Analyse und treatment

3.5.1 Analyse

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

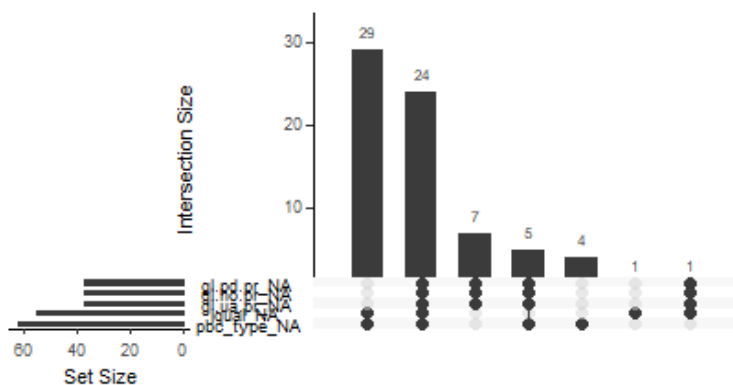
<https://bookdown.org/mwheymans/bookmi/>

Schneller Überblick

```
data %>%  
  skimr::skim()
```

Übersicht über pattern

```
data %>%  
  naniar::gg_miss_upset()
```



Anzeige der Missings pro Variable:

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

Anteil der missings bekommt man durch die Subtraktion mit den Fällen

```
studyData4 %>%  
  purrr::map_df(~sum(is.na(.)/length(.))) %>%
```

```
unlist() %>%  
round(.,2)
```

Mal checken. Dave macht es über `summarise(mean(is.na(y)))`

3.5.2 NAs löschen

Alle Fälle löschen, die überall missing sind

```
data %>%  
  filter_all(any_vars(!is.na(.)))
```

3.5.3 Umwandeln von NaN in NA

```
df %>%  
  mutate(Year2 = ifelse(is.nan(Year2), NA, Year2) )
```

...bei mehreren Variablen

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

3.5.4 Werte in NA umwandeln

Bestimmten Wert bei EINER Variable in NA umwandeln

```
mutate(sex = na_if(sex, "divers"))
```

Bestimmten Wert bei MEHREREN Variablen in NA umwandeln

```
mutate(across(c(height, weight, cig), na_if, -99))
```

oder bei einem Bereich:

```
mutate_at(vars(lp1:tf12), na_if, -99)
```

3.5.5 NA in Werte umwandeln

NAs alle Variablen ersetzen (Achtung!)

```
mutate_if(is.numeric, ~replace(., is.na(.), -99))
```

Missings durch mean imputieren

```
data %>%  
  mutate(mTenure= ifelse(is.na(mTenure), mean(mTenure, na.rm=TRUE), mTenure))
```

Missings in Werte umwandeln (z.B. NA → -99)

- Bei einer Variablen
`mutate(x1 = replace_na(x1, -99))`
- Bei mehreren Variablen
`mutate_at(vars(x1:x12), ~replace_na(., -99))`

4 Daten verändern: Mutate, rename, separate, unite

4.1 Verbinden von mutate mit eigenen Funktionen

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))  
}
```

2) Auf die dann in mutate_if verwiesen werden

```
feat_scl <- features %>%  
  mutate_if(is.numeric, rscl)
```

4.2 Anlegen einer Sequenz

```
data %>%  
  mutate(t = full_seq(1:nrow(data), 1))  
Alternative (z.B. wenn man eine ID anlegen will)  
data %>%  
  mutate(id = row_number())
```

4.3 Composite bilden

Mittelwerts-Composite:

```
data %>%  
  rowwise() %>%  
  mutate(KA = mean(c(KA1, KA2, KA3, KA4, KA5, KA6, KA7), na.rm=TRUE)) %>%  
  ungroup()
```

(ungroup ist wichtig, sonst bleibt das tibble im „rowwise-Modus“)

Man kann mit `c_across()` auch über einen range von Variablen mitteln

```
data %>%  
  rowwise() %>%  
  mutate(KA = mean(c_across(KA1:KA7), na.rm=TRUE)) %>%  
  ungroup()
```

4.3.1 McDonalds Omega

```
data %>%  
  select(v1, v2, v3) %>%  
  psych::omega(., plot=FALSE)
```

4.3.2 Splitten von kontinuierlichen Variablen

```
flights %>%  
  mutate(delay_cat = factor(dep_delay>12.6, labels=c("small", "strong"))) %>%  
  group_by(delay_cat) %>%  
  summarise(Mean = mean(dep_time, na.rm=TRUE))  
#Der Rest ist nur Veranschaulichung: Mittel für beide Gruppen
```

4.4 Variablen rekodieren

```
data <- data %>%
  mutate(time01r.t1 = recode(time01.t1, "0"=6, "1"=5, "2"=4, "3"=3, "4"=2, "5"=1,
                              "6"=0 ))

df %>% mutate(sex=recode(sex,
                          `1`="Male",
                          `2`="Female"))
(`1` = Ursprung, „male“ = Ziel)
```

Klasse rekodieren für eine Variable

```
data <- data %>%
  mutate(z2 = as.character(z))
```

Klasse rekodieren für mehrere Variablen

```
data <- data %>%
  mutate_at(.vars = c("Survived", "Sex", "Pclass"), .funs = factor)
```

bzw.

```
mutate_at(.vars = c("rtng_ov", "rtng_equipment", "rtng_equipment"),
          .funs = as.numeric )
```

Klasse rekodieren für alle Variablen

```
data %>%
  mutate_if(is.character, factor)
```

Erstes ist Bedingungsprüfung

Eine ähnliche Funktion ist `map_dbl()`

Damit kann man über jede Spalte im tibble eine Funktion laufen lassen, z.B. wieviel missings es gibt. Die gewählte Funktion muss aber kompatibel sein mit den Klassen der Variablen. Wenn nicht, die passenden selektieren

Dummies aus metrischen Variablen bilden

```
data %>%
  mutate(tms_d = case_when(n_tms == 0 ~ 0,
                           TRUE ~ 1))
```

Dummies aus kategorialen Variablen bilden / dummy-Variablen bilden

```
data %>%
  sjmisc::to_dummy(age, suffix = "label") %>%
  bind_cols(data) %>%
  as_tibble() %>%
  janitor::clean_names()
```

Grundlage ist eine kategoriale Altersvariable. Der `to_dummy`-Befehl bildet automatisch k dummies—in der Regression muss man die erste Variable dann rauslassen. Das `labels`-Argument fügt die Kategorien dann an die dummy-Variable an; `clean_names` verschönert den Namen etwas, so dass er nicht in Anführungszeichen steht.

4.5 Scientific notation ändern

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

```
mutate_if(is.numeric, round, 5)
```

4.6 Veränderungen von Kategorien: fct_lump, fct_reorder & fct_recode

4.6.1 Kategorien zusammenlegen: fct_lump()

Mit `fct_lump()` aus dem **forcats** package kann man gering besetzt Faktorlevel zusammenlegen zu einer „others“-Kategorie. Sehr nice bei barplots, die ansonsten eine Masse von uninteressanten Balken zeigen würde. **Das n-Argument legt die Anzahl der gewünschten Kategorien fest**

```
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

starwars %>%
  mutate(skin_color = fct_lump(skin_color, n = 5)) %>%
  count(skin_color, sort = TRUE)

# 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
```

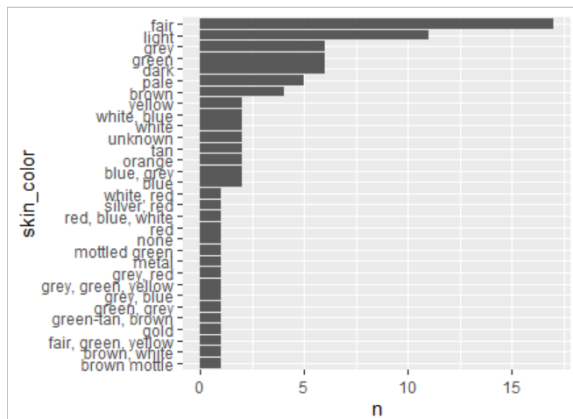
```
mutate(edu_cat = recode_factor(edu_cat, "7"="Ohne Abschluss" ,
  "1"="Haupt- oder Volksschule",
  "2"="Realschule",
```

4.6.2 Faktorenlevel neu ordnen: fct_reorder und fct_relevel

Das ist v.a. dann schick wenn man sie entlang einer Statistik in der Reihenfolge anordnen will. Diese Statistik kann im einfachsten Fall ein count sein (n) oder der Mittelwert in Y etc. Dave nimmt das immer um bei einem `geom_col` die Faktor-levels entlang der Häufigkeit anordnen will.

```
starwars %>%
  count(skin_color, sort = TRUE) %>%
  mutate(skin_color = fct_reorder(skin_color, n)) %>%
```

```
ggplot(aes(x=skin_color, n)) +
  geom_col()+
  coord_flip()
```



```
data %>%
  mutate(age = fct_relevel(age, ">60", after = Inf))
```

Inf führt dazu dass die 60 ans Ende kommt. Alternativ kann man eine Zahl reinschreiben, die die gewünschte Position angibt)

Man kann sie aber auch einfach in die gewünschte Reihenfolge bringen:

```
data %>%
  mutate(educ = fct_relevel(educ, "Secondary school certificate",
                             "High School Degree",
                             "Certificate for university entrance",
                             "Bachelor",
                             "Master",
                             "Magister",
                             "Diplom",
                             "Promotion"))
```

4.6.3 rct_recode(): Faktorlevels/Kategorien umbenennen

```
data%>%
  mutate(factorname = fct_recode(factorname, "neu" = "alt"))
```

Hier ist alle ein level des Faktors

4.6.4 Kontinuierliche Variable kategorisieren

```
age_data %>%
  mutate(age_cat = cut(age, c(18, 25, 40, 65, 90)))
```

(Achtung, die Endbegrenzungen (Min und Max) müssen mit rein)

4.7 Bedingtes Berechnen

Es gibt zum einen die if_else()-Funktion. So wie ich das sehe, ist die v.a. für simple Berechnungen:

```
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 ist "else" (wenn „else“ der Fall ist). Lässt man das einfach weg, wird es missing.
- „TRUE ~ X“ bedeutet, dass er die originalen Werte der X-Variable nehmen soll
- Wenn man sagen will, dass für die sonsteigen ein bestimmter Wert genommen werden soll, schreibt man `TRUE ~ 3` (oder ähnliches)

4.7.1 In missings umwandeln

Wenn man einen numerischen Wert in ein Missing umwandeln will geht das mit `na_if()`

```
data %>%  
  mutate(eye_color = na_if(eye_color, "unknown"))
```

Bei mehreren Variablen

```
mutate(across(c(height, weight, cig), na_if, -99))
```

4.7.2 Missings durch mean imputieren

```
data %>%  
  mutate(mTenure= ifelse(is.na(mTenure), mean(mTenure, na.rm=TRUE), mTenure))
```

Schönes Beispiel für die Einbindung von `ifelse` in `mutate`

4.8 Umbenennen

```
data <- data %>%  
  rename(Neu=alt, B=y, F1=f1, F2=f2)
```

Links stehen die Zielvariablen, rechts die Ursprungsvariablen

4.9 Variablen zerlegen und fusionieren mit `separate` und `unite`

4.9.1 Separate

Separate trennt eine Variable, die aus 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.

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

Achtung: R ersetzt die Datum durch die Aufsplittung

Besser die Originalvariable erst kopieren dann trennen:

```
chicago %>%  
  mutate(date2=date) %>%  
  separate(date, c("year", "month", "day"), sep="-")
```

Trennen bei Fehlen eines separators, z.B. m23

```
data %<%  
  separate(gender_age, sep=1)
```

Trennen von mehreren Zahl in einer Zelle mit `separate_rows()`

```
# A tibble: 3 × 3  
  subject_id visit_id measured  
    <int> <chr> <chr>  
1     1001 1,2, 3 9,0, 11  
2     1002 1|2 11, 3  
3     1003 1 12
```

```
data %>%  
  separate_rows(visit_id, measured, convert=TRUE)
```

```
# A tibble: 6 × 3  
  subject_id visit_id measured  
    <int> <int> <int>  
1     1001 1 9  
2     1001 2 0  
3     1001 3 11  
4     1002 1 11  
5     1002 2 3  
6     1003 1 12
```

- Für jeden Eintrag wird eine neue Zeile aufgemacht (Nestung)
- 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 `visit_id` von 1, 2 und 3 und NA in der `measured`-Variable (d.h. R orientiert sich am Fall mit der maximalen Anzahl)

```
  subject_id visit_id measured  
    <int> <int> <int>  
1     1001 1 9  
2     1001 2 0  
3     1001 3 11  
4     1002 1 11  
5     1002 2 3  
6     1002 3 NA  
7     1003 1 12  
8     1003 2 NA  
9     1003 3 NA
```

4.9.2 Unite

Unite macht das Gegenteil und verbindet Variablen zu einer neuen

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

Hier ist „date“ die kombinierte Zielvariable

Das ist auch `str_c` überlegen, wenn man einen Bereichsoperator nehmen will (was bei `str_c()` wohl nicht geht).

```
unite("top_freex_words", word.1:word.5, remove = TRUE, sep=", ") %>%
```

4.9.3 Variablen zusammen-pasten

```
flights %>%
  mutate(data = paste0(month, "-", day, "-", year))
```

	year	month	day	dep_time	sched_dep_time	date
	<int>	<int>	<int>	<int>	<int>	<chr>
1	2013	1	1	517	515	1-1-2013
2	2013	1	1	533	529	1-1-2013
3	2013	1	1	542	540	1-1-2013
4	2013	1	1	544	545	1-1-2013
5	2013	1	1	554	600	1-1-2013
6	2013	1	1	554	558	1-1-2013
7	2013	1	1	555	600	1-1-2013
8	2013	1	1	557	600	1-1-2013
9	2013	1	1	557	600	1-1-2013
10	2013	1	1	558	600	1-1-2013

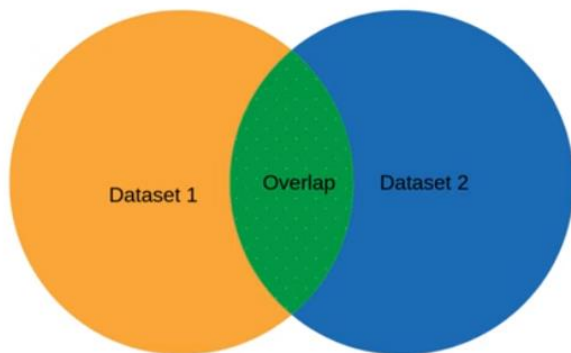
→ Verbindet die month, day-und year-Variablen in eine und verbindet sie mit einem dash

4.10 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



Inner join: **GREEN**

Left join: **YELLOW + GREEN**

Right join: **BLUE + GREEN**

Full join: **YELLOW + GREEN + BLUE**

Semi join: **GREEN - BLUE**

Anti join: **YELLOW - GREEN**

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

Sie hat 3 Datensätze:

Studentenfile

	Student_ID	First_Name	Last_Name	Gender	Major	Ethnic_Code
1	1001	Ryan	Williams	M	Computer Science	1
2	1002	Melissa	Smith	F	Biology	1
3	1003	Sam	Held	M	Literature	4
4	1004	Kyle	Zhang	M	Astronomy	3
5	1005	Reena	Kapoor	F	Social Science	3

Location file

	Student_ID	Location
1	1001	Phoenix
2	1002	Tempe
3	1003	Chandler
4	1004	Glendale
5	1008	Gilbert

Ethnicity file

	Ethnic_Code	Ethnicity_Description
1	1	White
2	3	Asian
3	5	Hispanic

Wichtig: Die Venn-Diagramme beziehen sich 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 enthaltene 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. Ist super um im Vorfeld nicht-passende Key-Worte zu identifizieren.
Zwei Besonderheiten
1) Es braucht keine ID
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?
```

4.11 pivot_longer() und pivot_wider: Format der Daten ändern

4.11.1 Pivot_longer

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
```

```
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

Beispiel: billboard

```

# A tibble: 317 x 79
  artist track date.entered wk1 wk2 wk3 wk4 wk5 wk6 wk7 wk8
  <chr>   <chr>   <date>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 2 Pac   Baby~ 2000-02-26 87 82 72 77 87 94 99 NA
2 2Ge+h~ The ~ 2000-09-02 91 87 92 NA NA NA NA NA
3 3 Doo~ Kryp~ 2000-04-08 81 70 68 67 66 57 54 53
4 3 Doo~ Loser 2000-10-21 76 76 72 69 67 65 55 59
5 504 B~ Wobb~ 2000-04-15 57 34 25 17 17 31 36 49
...

```

- Der count ist die Chart-Platzierung
- Die wk's gehen bis wk76
- Es soll jetzt eine wk-Spalte geben

```

billboard %>%
  pivot_longer(wk1:wk76, names_to="wk", values_to="count")

  artist track date.entered wk count
  <chr>   <chr>   <date>   <chr> <dbl>
1 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk1 87
2 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk2 82
3 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk3 72
4 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk4 77
5 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk5 87
6 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk6 94
7 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk7 99
8 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk8 NA
9 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk9 NA
10 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk10 NA
...

```

- Es gibt massig missings—die kann man mit `values_drop_na = TRUE` löschen
- Anstatt alle Variablen aufzuzählen, kann man—v.a. wenn sie einen gemeinsamen Stamm haben diesen mit `cols = starts_with("wk")` ansprechen.

4.11.2 Pivot_wider

- Macht das Gegenteil und überführt einen longformat-Datensatz in wide format, so dass ein Fall = eine Zeile wird
- Beispiel: Mittelwerte und Varianzen einer Cluster-Analyse; diese stehen für jeden Cluster (Klasse) untereinander. Ziel soll sein, dass Mittelwert und Varianz 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) genestet 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()`: Woraus 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_Variiances se_Means se_Variiances
<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
```

4.12 Programming und Funktionen

4.12.1 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 tidyverse 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 man „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](#)):

```
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)
...
```

4.12.2 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)
```

4.12.3 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 Fibonacci-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 unnest
```

```
(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 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)` verändern
- 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 rumgezockt, 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 Beispiel 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 durch die Umwandlung in ein tibble der Parameter diesen hässliche column name bekommt
- Mittels `print()` wird das Ergebnis zwischendurch ausgegeben (return scheint zu bewirken, dass danach nix mehr ausgegeben wird)

4.12.4 Map und anonyme Funktionen

4.12.4.1 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 Element 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 Struktur 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. Wickham 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 verschiedenen Bedeutungen)
- Mittels `map()` cycles 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 geschrieben wird) nicht vordefinieren muss. Verglichen mit `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	<code>results = numeric(10)</code>	<code>V = c(1:10)</code>	<code>V = c(1:10)</code>
Funktion	<code>for(i in 1:10) { results[i] = i*2 }</code>	<code>map(V, function(x) { x*2 })</code>	<code>lapply(V, function(x) { x*2 })</code>

	<code>print(results)</code>	<code>}}</code>	<code>}}</code>
		Alternative <code>map(V, ~{. *2})</code>	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-Schreibweise (`~`) 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
...
```

4.12.4.2 Formel-Schreibweise

- Während man mit map die standardgemäße 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})
```

4.12.4.3 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
```

4.12.4.4 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 unsinnig)
```

- **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 filenames 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 charakteristisches 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.

- (2) **Files einlesen**

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

- (3) **ID aus den filenames extrahieren**

Dies geht natürlich nur, wenn der filename 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 filenames enthält

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

- (2) 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

## 5 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](#))
- **Tutorial:** <https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html>
- **Original 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** (<dtm>): kombiniert Datum und Zeitpunkt (z.B. 2013-04-02 1:05:32).
  - Im base R heißen die POSIXct

### 5.1 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

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

Oft ist eine Datumsvariable als String formatiert. Diese muss durch **ymd()**, **mdy()**, **dmy()** in ein date umgewandelt werden. 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
```

Z.B. muss bei „13. Mai 18“ R gesagt bekommen, was die Zahlen bedeuten. Hier also “day-month-year”

Entsprechend ist die Umwandlung

```
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. angeht als auch, was die Nennung von Elementen angeht: Enthält die Klammer Minuten, muss man auch `_hm` wählen (und nicht `_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:

- **Aus einem deutschen Datum („20.02.20“)**

```
tmp$time <- as.Date(tmp$time, format="%d.%m.%Y")
```

Nimmt man `as.POSIXct()` wird eine Uhrzeit von 1:00:00 überall hinzugefügt—welche Funktion man nutzt, hängt also davon ab, ob es in den Rohdaten eine Uhrzeit gibt (wenn nicht, kann man auch `as.Date()` nehmen)

- **Wenn die Uhrzeit mit dran hängt** und man ein `datetime` Objekt haben möchte

```
time
<chr>
1 01.01.2020 00:00
2 01.01.2020 00:01
3 01.01.2020 00:02
4 01.01.2020 00:03
5 01.01.2020 00:04
```

```
data %>%
 mutate(time = as.POSIXct(time, format="%d.%m.%Y %H:%M"))
time
<dtm>
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
```

- **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>

### 5.1.2 Zusammensetzung von Datumsangaben aus Komponenten

Manchmal man die einzelnen Komponenten separat gespeichert

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

```
#> # A tibble: 336,776 × 5
#> 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
#> # ... with 3.368e+05 more rows
```

#### Zusammenfügen mit `make_date()` oder `make_datetime`

```
flights %>%
 select(year, month, day, hour, minute) %>%
 mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
#> # A tibble: 336,776 × 6
#> year month day hour minute departure
#> <int> <int> <int> <dbl> <dbl> <dtm>
#> 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
#> # ... with 3.368e+05 more rows
```

#### Zusammenfügen aus Jahr und Monat zu einem Date.

Ausgangslage war ein tibble in dem nur das Jahr und der Monat vorhanden war (Bier-[Video](#) 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
```

## 5.2 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 dt

```
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))
```

Restuliert in einer reinen Zahl der dbl-Klasse

- Mit der yearweek()-Funktion  

```
weekly_data <- day_data %>%
 mutate(week = yearweek(time))
```

Restuliertr 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. anfangen). 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)
```

## 5.3 Runden

---

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

### 5.3.1 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
 <dtm>
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. Monat oder Tag) ab, ist die Zeitkomponente damit hinfällig und mit 00:00:00 angezeigt

### 5.3.2 `ceiling_date()`: Aufrunden zur nächst oberen Einheit

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

```
data_ts %>%
 mutate(time = ceiling_date(time, unit="month"))
 time
 <date>
1 2020-03-01
```

### 5.3.3 `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 die 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 Februar, wird daher auf 01.03. aufgerundet

## 6 Grafiken

### 6.1 Scatterplot

---

#### 6.1.1 Grundform

```
ggplot(mpg, aes(x = displ, y = hwy)) +
 geom_point()
```

aes = mapping

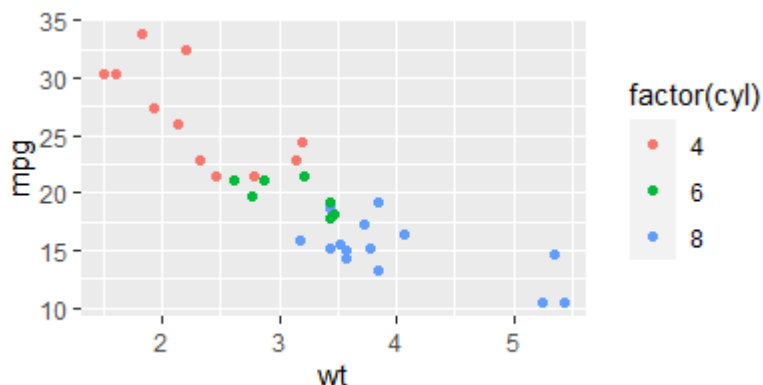
- Zuweisung der Variablen PLUS
- Aesthetics, wie size, shape, colour zur *Datenstrukturierung* (d.h. wenn aesthetics auf Variablen-Werte bezogen werden)

#### 6.1.2 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).

##### 6.1.2.1 Farbe nach einer kategorialen Drittvariablen

```
mtcars %>%
 ggplot(aes(wt, mpg)) +
 geom_point(aes(colour = factor(cyl)))
```

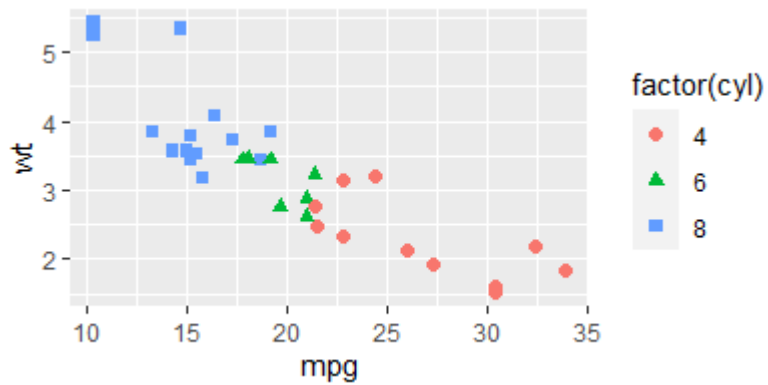


##### 6.1.2.2 Form nach einer kategorialen Drittvariablen

(hier mal in Kombi mit der Farbe und Vergrößerung der Symbole)

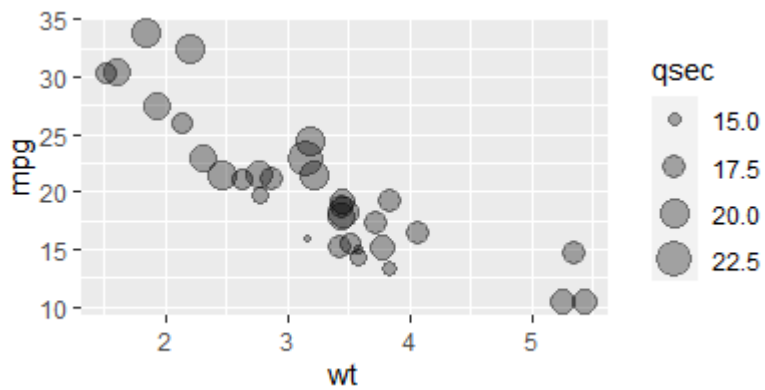
```
mtcars %>%
 ggplot(aes(mpg, wt, shape = factor(cyl))) +
 geom_point(aes(colour = factor(cyl)), size = 2)
```





### 6.1.2.3 Größe der bubbles nach einer kontinuierlichen Drittvariablen

```
mtcars %>%
 ggplot(aes(wt, mpg)) +
 geom_point(aes(size = qsec), alpha=1/3)
```



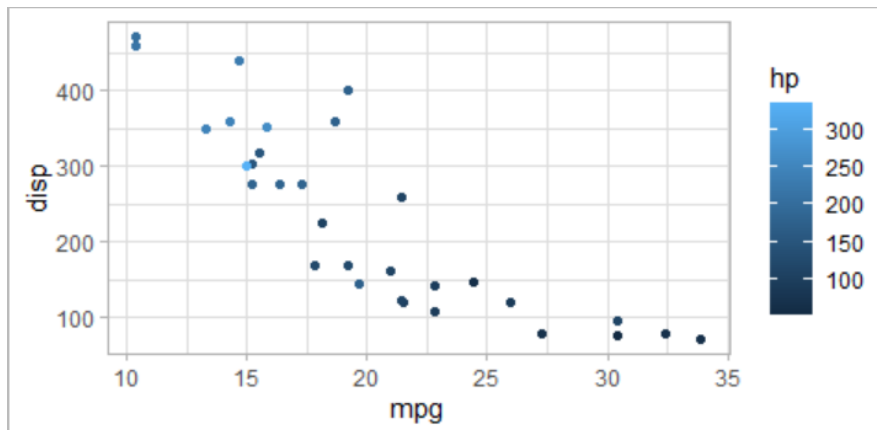
- **shape = ...** (Symbole): z.B.

- 1: Kreis
- 2: Dreieck
- 3: Plus etc.

Interessant kann shape sein, wenn man damit Gruppen kennzeichnet (aes)

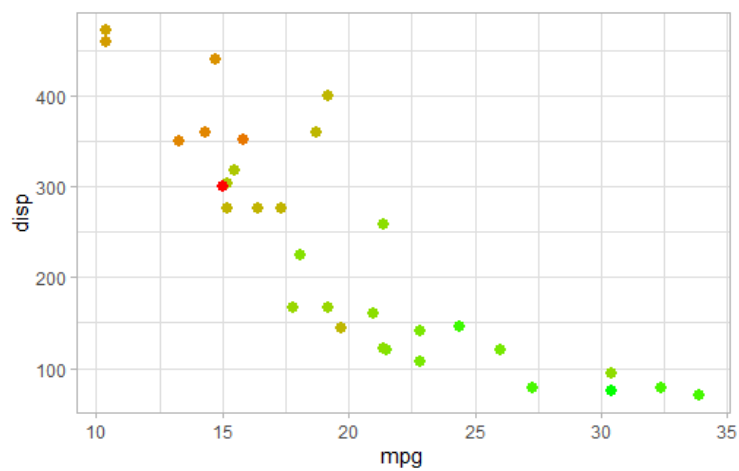
### 6.1.3 Farbe nach einer kontinuierlichen Drittvariablen

```
mtcars %>%
 ggplot(aes(x=mpg, y=disp)) +
 geom_point(aes(colour=hp))
```



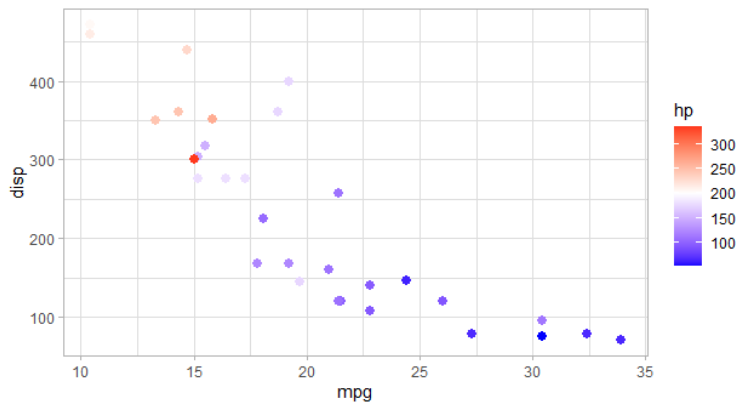
Die Abstufungen und Farben kann man frei wählen:

1. Version: Kontinuierlicher Übergang von einer Farbe zur nächsten mittels `scale_color_gradient()`



```
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: Wahl eines neutralen Midpoint



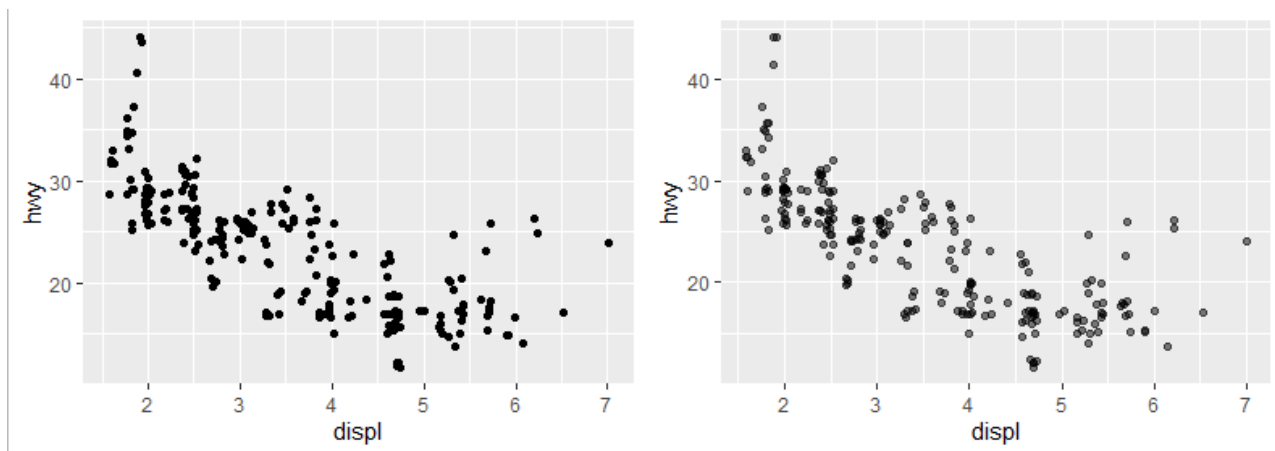
```
mtcars %>%
 ggplot(aes(x=mpg, y=disp)) +
 geom_point(aes(colour=hp), size=2.5) +
 scale_color_gradient2(low="blue",
 high = "red",
 midpoint=200) +
 theme_light()
```

### 6.1.4 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 etwas 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)**  

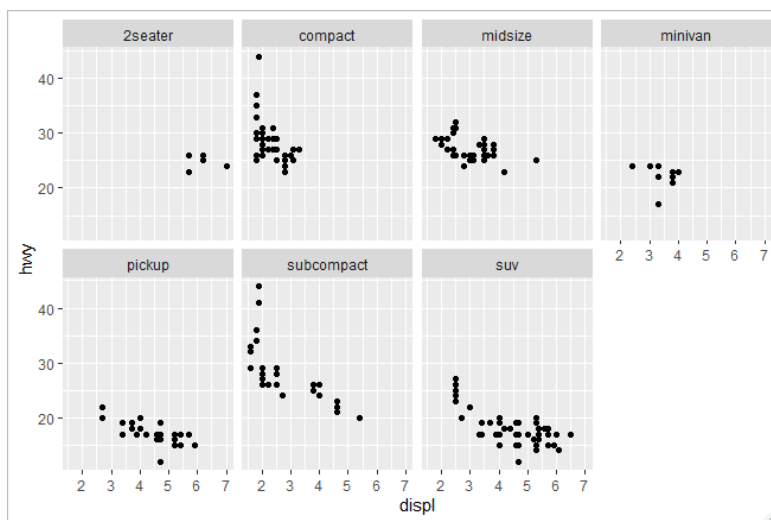
```
geom_point(shape = 1)
```
- **Reduktion auf Pixel**  

```
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)
```

### 6.1.5 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))
```

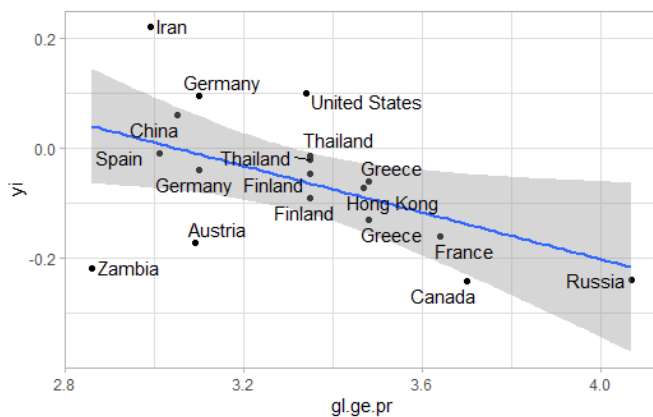
### 6.1.6 Punkte labeln

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()
```



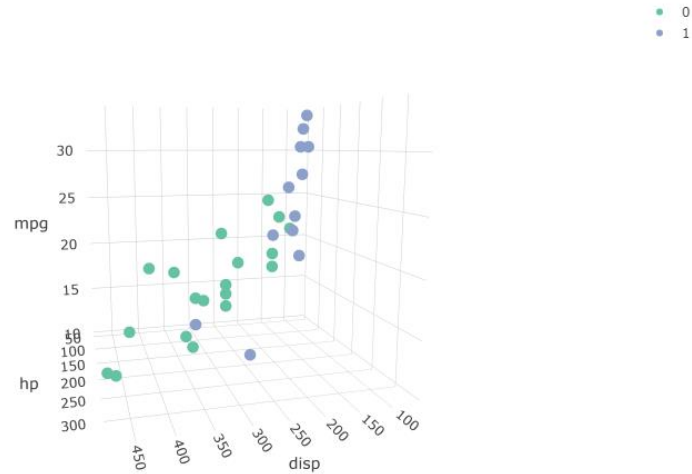
- alternative ist `geom_text(check_overlap = TRUE, hjust = "inward")` was die labels direkt an die Punkte flanscht, was die Zuordnung einfacher macht

### 6.1.7 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
```

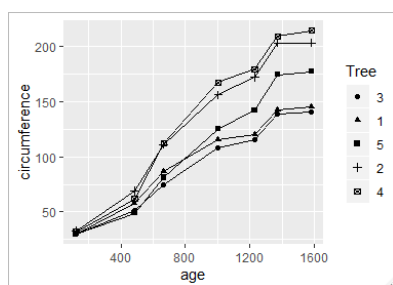
**#Achtung: Die Z-Achse ist die Y-Variable!**



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

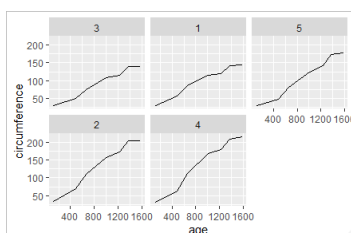
## 6.2 Linien und Kurven

```
Orange %>%
 ggplot(aes(age, circumference, shape=Tree)) +
 geom_point() +
 geom_line()
```



Tree ist hier die Gruppierungsvariable, die durch 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

### 6.2.1 Spaghetti plots / growth

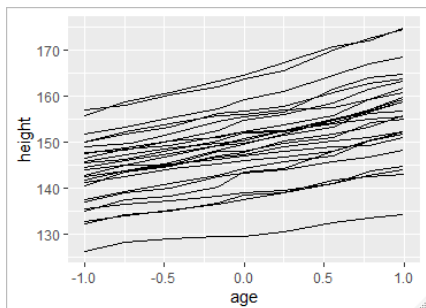
- Die sind eine Alternative für das facet-wrap. Hier werden alle Units abgebildet
- Die Daten sind hier in Multi-level-Struktur / long format. Wenn sie im wide format vorliegen, müssen sie mit `gather()` (siehe im entspr. Abschnitt) umgeschichtet werden. Das geht alles in der selben pipeline
- Beispiel 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)

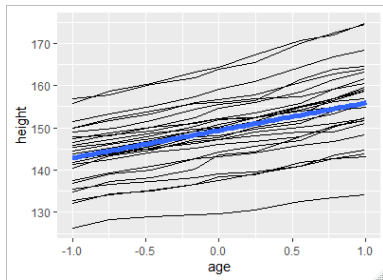
A tibble: 234 x 4
 Subject age height Occasion
 <ord> <dbl> <dbl> <ord>
1 1 -1 140. 1
2 1 -0.748 143. 2
3 1 -0.463 145. 3
4 1 -0.164 147. 4
5 1 -0.0027 148. 5
6 1 0.247 150. 6
7 1 0.556 152. 7
8 1 0.778 153. 8
9 1 0.994 156. 9
10 2 -1 137. 1
11 2 -0.748 139. 2
12 2 -0.463 140. 3
13 2 -0.164 143. 4
14 2 -0.0027 143. 5
15 2 0.247 144. 6
```

```
ggplot(Oxboys, aes(age, height)) +
 geom_line(aes(group = Subject))
```

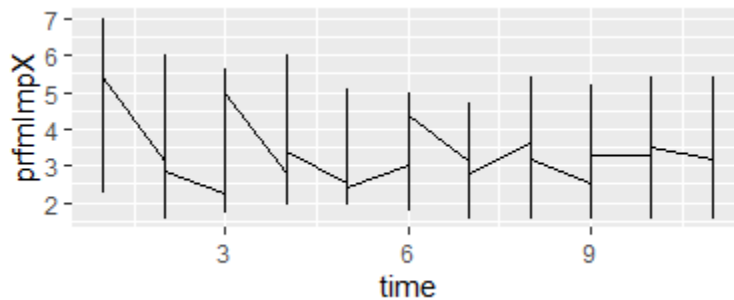


Hinzufügen eines mittleren trends (method=gam als default). Dann sieht man ja ob er linear ist

```
ggplot(Oxboys, aes(age, height)) +
 geom_line(aes(group = Subject)) +
 geom_smooth(se = FALSE)
```



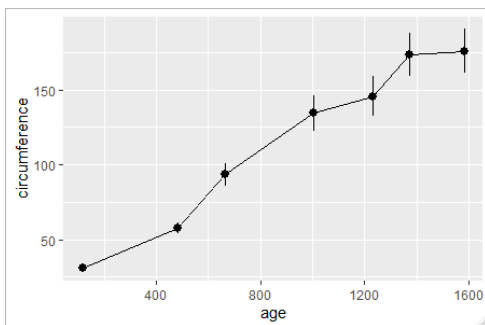
**Hinweis:** Wenn der plot so aus sieht, fehlt in `geom_line` die Subject ID



## 6.3 Darstellen von Statistiken

- Mittelwertsverlauf über das Alter (inkl. SE des Mittelwerts)  

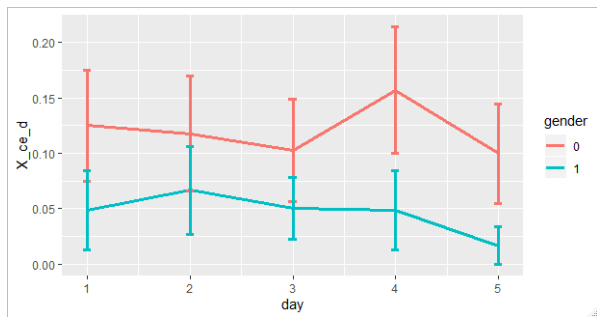
```
ggplot(Orange, aes(age, circumference)) +
 stat_summary(fun.y=mean, geom="line") +
 stat_summary(fun.data=mean_se, geom="pointrange")
```



- Die SE kann man auch mit „errorbar“ machen, hier in Kombi mit einer Gruppierungsvariable aus einer Tagebuchstudie:  

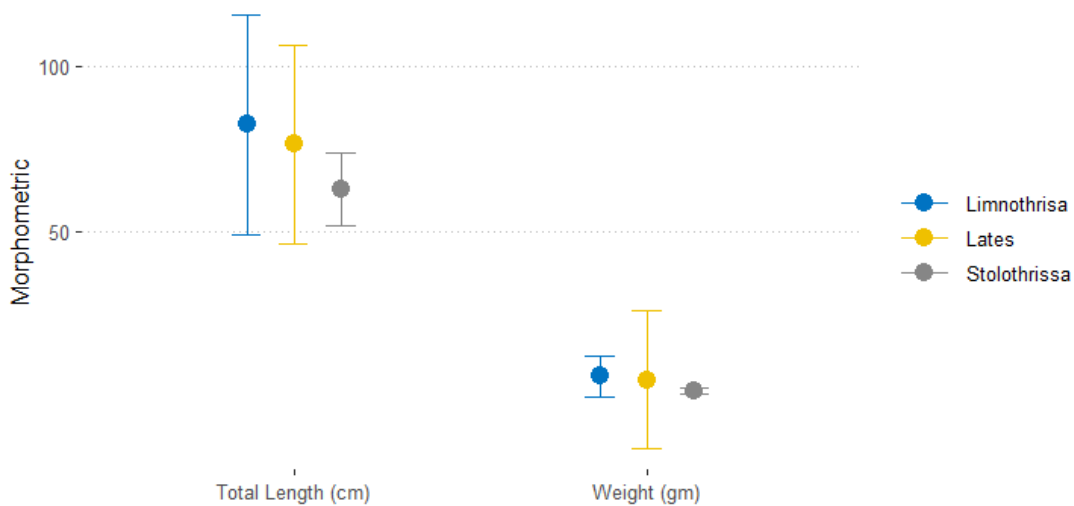
```
ggplot(global_dt, aes(day, X_ce_d, color=gender)) +
 stat_summary(fun.y=mean, geom="line", size=1.3) +
 stat_summary(fun.data = mean_se, geom="errorbar", size=1.1, width=.08)
```





### Anpassungen:

- `Size = #` ändert die Dicke der Linien
- `width = #` verkleinert 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 variablen upper und lower angelegt werden, bei der vom Mittelwert 1 SD addiert oder subtrahiert 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 = "data") %>%
 group_by(species, variable) %>%
 summarise(data.mean = mean(data),
 data.sd = sd(data),
 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("Limnothrissa", "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)"))
```

### 6.3.1 Einfügen von Regressionsgeraden in scatterplots

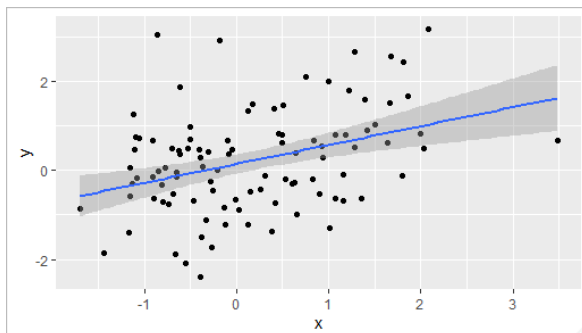
Linien und Kurven werden nach Regressionen eingefügt

- `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:

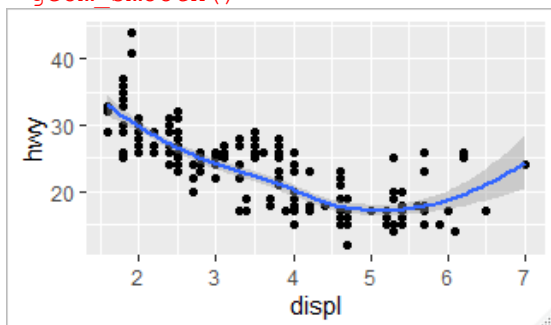
- **Regressionslinie**

```
data.tb %>%
 ggplot(aes(x, y)) +
 geom_point() +
 geom_smooth(method="lm")
```



- **Kurve**

```
ggplot(mpg, aes(x = displ, y = hwy)) +
 geom_point() +
 geom_smooth()
```



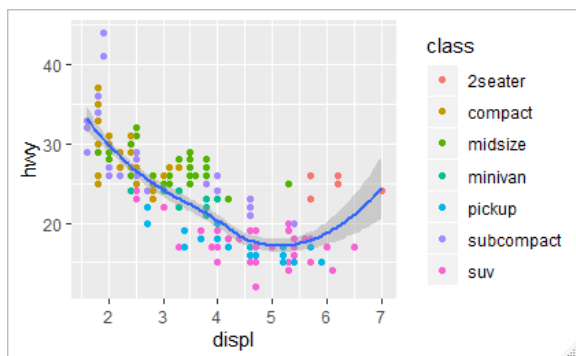
- Bei  $N < 1000$  wird die Kurve durch loess geschätzt (local regression), d.h. `method=loess` (default); bei  $N > 1000$  ist es ein generalized additive model `method="gam"`. In dem Fall muss

man aber eine Formel hinter `method = ....` Einfügen, die entweder `formula = y ~ s(x)` or `y ~ s(x, bs = "cs")`

- Die wiggleness wird durch das Argument „`span = .2`“ geregelt; Bereich ist 0-1. Default ist 1
- **ACHTUNG.** Manchmal ist die smooth weniger wiggly, als eine aus einem plot. Das kann man umgehen in dem man erst ein gam rechnet, die fitted values in die Daten schreibt (geht leider nicht mit `augment`) und dann die line mit den fitted values macht  
`geom_line(aes(y ~ .fitted))`

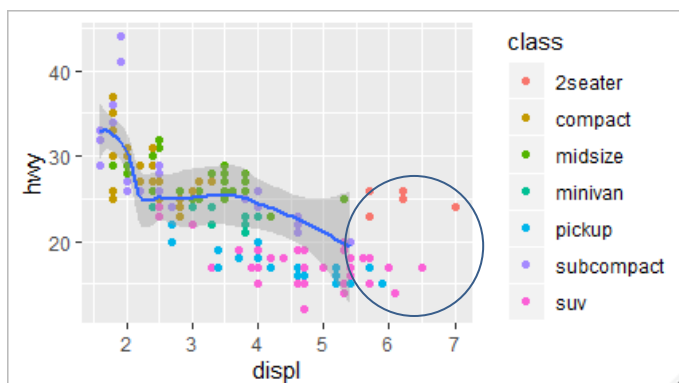
Will man nur die Punkte nach Gruppen aufteilen, aber nicht die Kurve, ist das ein local mapping der Punkte

```
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"))
```

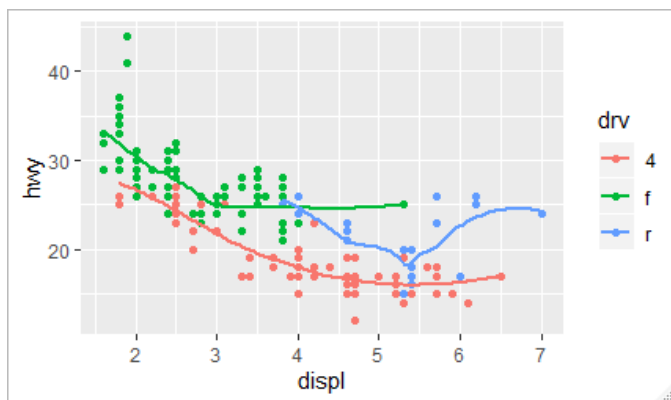


Hier wird die smoothing line nur für die Subgruppe der "subcompact" cars angezeigt. Macht hier nicht wirklich Sinn; es geht aber ums Prinzip

So bekommt man Punkte und Kurven für Subgruppen (→ Moderatoreffekte)

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
 geom_point() +
```

```
geom_smooth(se = FALSE)
```



## 6.4 Barplots

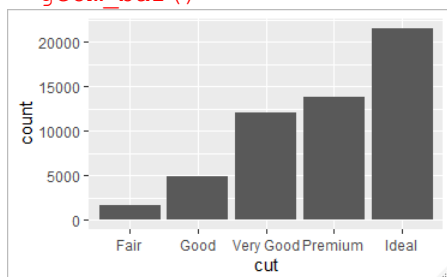
- Es gibt 2 Arten von Barplots
  - **geom\_bar**: Anzeige der Häufigkeiten
  - **geom\_col**: Geht zwar auch mit Häufigkeiten (wenn count() vorgeschaltet ist, eignet sich aber besser für die Anzeige von Werten (z.B. Stats)

### 6.4.1 geom\_bar

Datensatz für die Beispiele:

```
diamonds <- ggplot2::diamonds
```

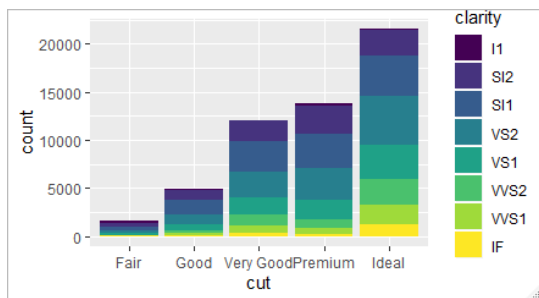
```
ggplot(diamonds, aes(x = cut)) +
 geom_bar()
```



Hier ist cut eine kategoriale Variable.

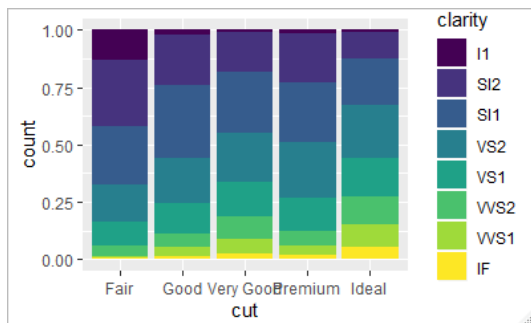
Säulen schachteln nach einer anderen Variablen:

```
ggplot(diamonds, aes(x = cut, fill = clarity)) +
 geom_bar()
```



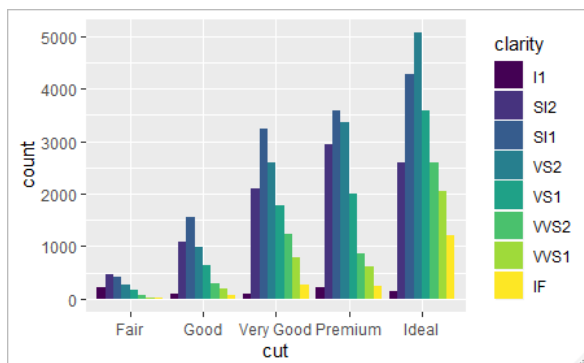
Diese Form kombiniert absolute Häufigkeit mit relativer \*pro Säule\*. Eine Vergleichbarkeit der Säulen bekommt man mit "position=fill", die für jede Säule 100% veranschlagt (hier muss aes, bzw. zumindest position in den geom-part).

```
ggplot(diamonds) +
 geom_bar(aes(x = cut, fill = clarity), position = "fill")
```



## Nebeneinander

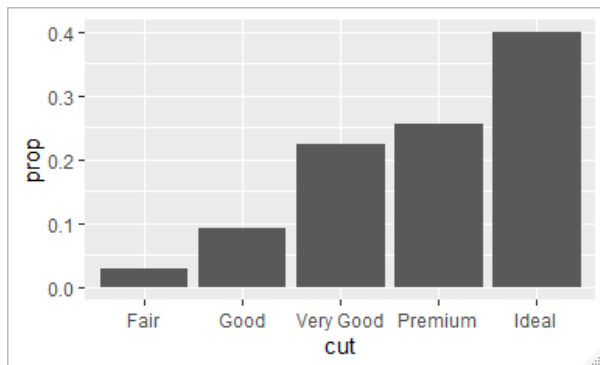
```
ggplot(diamonds, aes(x = cut, fill = clarity)) +
 geom_bar(position="dodge")
```



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

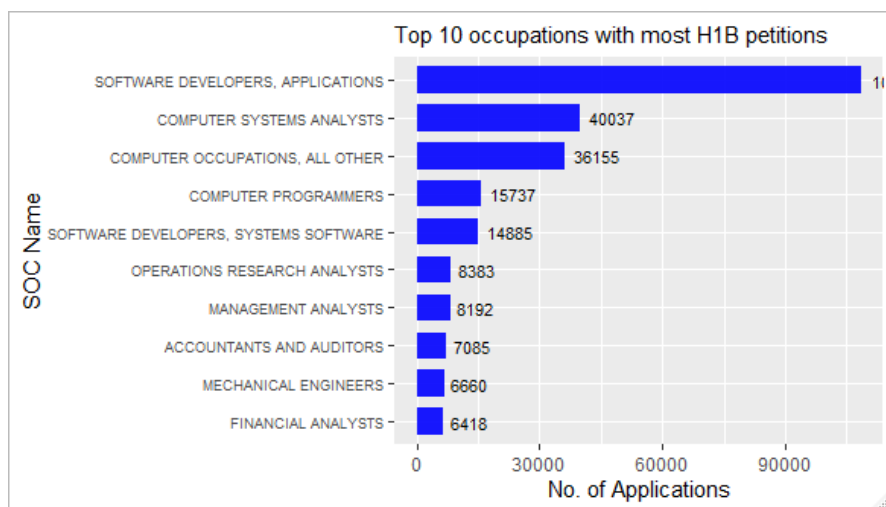
```
ggplot(diamonds) +
 geom_bar(aes(x = cut, y = ..prop.., group = 1)
)
```



(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)`

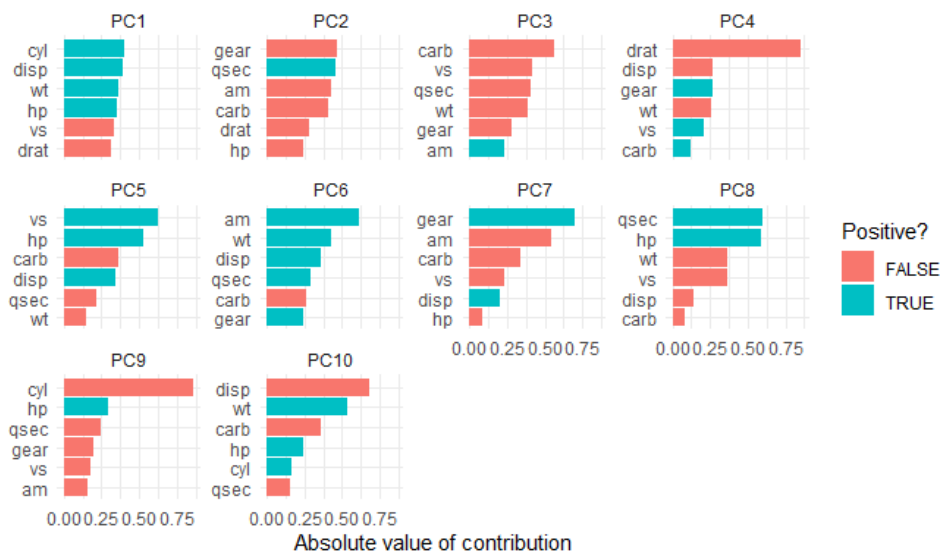
- Betrifft die Beschriftungen (Zahl der Bewerbungen)
- Label=Was für Werte sollten angezeigt werden (→ num\_apps; Zahl der Bewerbungen)
- hjust=Steuert, wie nah die Werte an den Balken sind. Weiß im Moment nicht, ob die absolut (im Raum) oder relativ zu den Balken sind. Einfach rumprobieren
- size=Größe 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)

6.4.2 geom_col

- Das Beispiel zeigt eine Faktorladungsmatrix (pca)

```
pca %>%
  mutate(component = fct_inorder(component)) %>% #Ordnet 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:

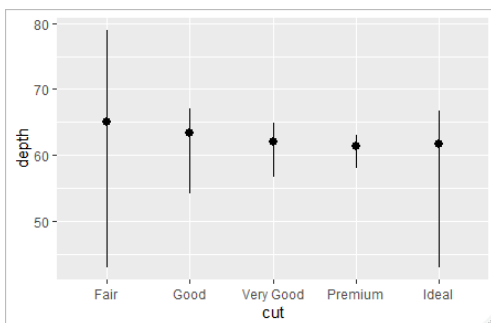


6.4.3 stat_summary: Anzeige von Statistiken anstelle Häufigkeiten

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
  )
```

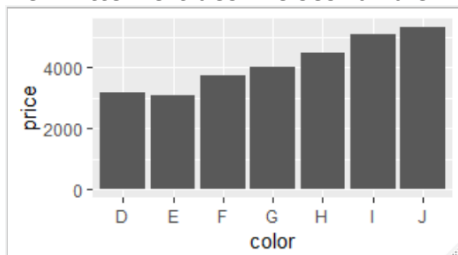
Das dürfte m.E. identisch sein zu einem gruppierten boxplot...Könnte aber noch andere Anwendungen geben.



Das kann man auch im bar plot machen:

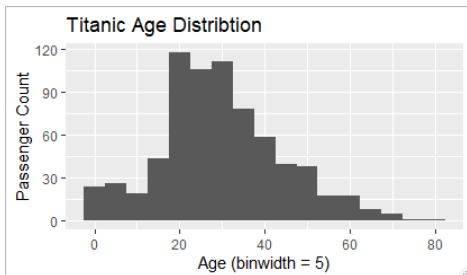
```
ggplot(diamonds, aes(color, price)) +
  geom_bar(stat = "summary_bin", fun.y = mean)
```

Der Mittelwert des Preises für die verschiedenen Diamanten-Farben („color“) wird angezeigt



6.5 Histogramme und Polygone

```
ggplot(titanic, aes(x = Age)) +  
  geom_histogram(binwidth = 5) +  
  labs(y = "Passenger Count",  
       x = "Age (binwidth = 5)",  
       title = "Titanic Age Distribution")
```

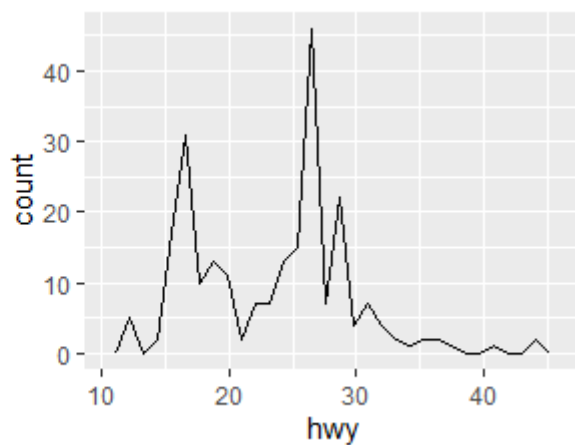
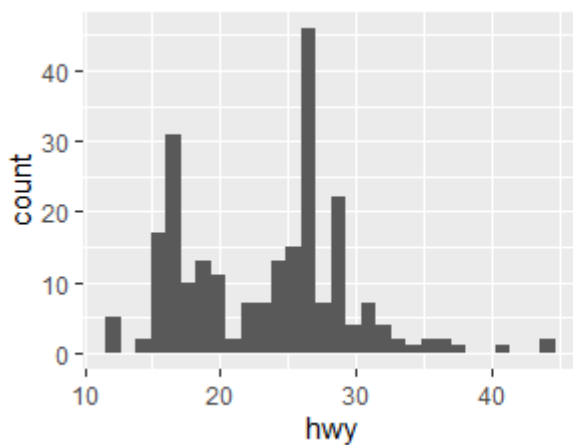


- Die Säulenbreite wird mit dem Argument `binwidth = 1` gesteuert
- Alternativ kann man mit `bins = 10` die Anzahl der säulen determinieren
- Mit `breaks` kann man die exakten cuts bestimmen
- Es ist wichtig damit zu experimentieren. Der default-Wert ist simpel und teilt alle Werte in 30 Säulen

Eine Alternative zu Histogrammen 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)
```



6.5.1 Aufsplittung nach Gruppen

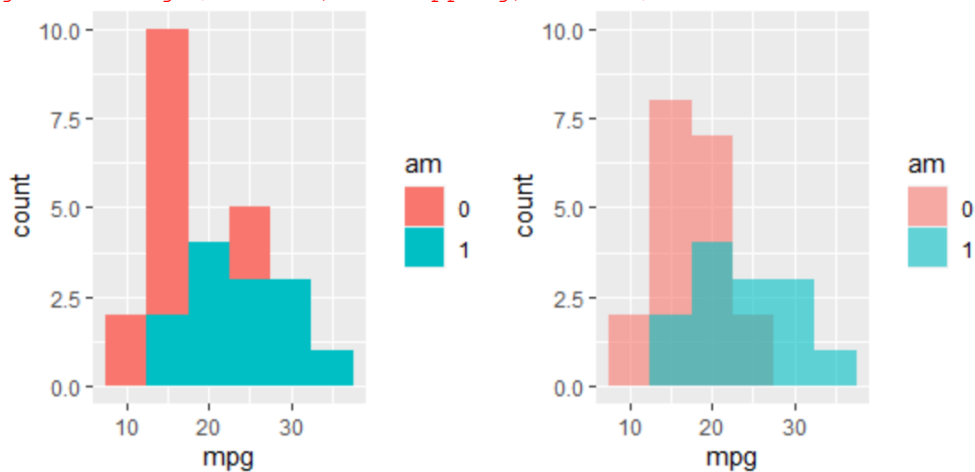
Hier bei gibt es zwei Wege: Das Histogramm der einen Gruppe wird auf das der andren oben drauf gesattelt (Rohversion) oder sie liegen hintereinander (`position = „identity“`)

```
library(gridExtra)

stacked <- mtcars %>%
  mutate(am=as.factor(am)) %>%
  ggplot(aes(x = mpg, fill = am)) +
  geom_histogram(binwidth = 5) +
  ylim(0, 10)

overlapping = mtcars %>%
  mutate(am=as.factor(am)) %>%
  ggplot(aes(x = mpg, fill = am)) +
  geom_histogram(binwidth = 5, position="identity", alpha=.6) +
  ylim(0, 10)

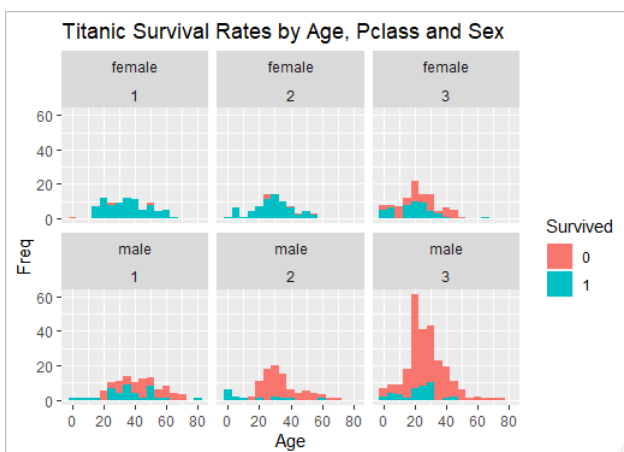
grid.arrange(stacked,overlapping, ncol=2)
```



Links der gestackte—rechts der überlappende

6.5.2 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")
```

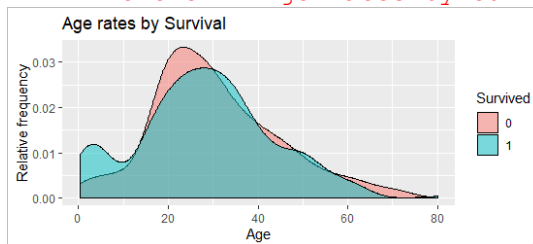


(Achtung: Die Balken sind übereinandergeschachtelt, nicht hinter-/voreinander. So haben in der 1. und Klasse nahezu alle Frauen überlebt und nur ganz wenige (die roten) nicht. Vgl. dazu den passenden density-plot unten, der andere Informationen trägt).

6.6 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 Unterschiede 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 interpretieren

```
ggplot(titanic, aes(x = Age, fill = Survived)) +
  geom_density(alpha = 0.5) +
  labs(y = "Relative frequency",
       x = "Age",
       title = "Age rates by Survival")
```



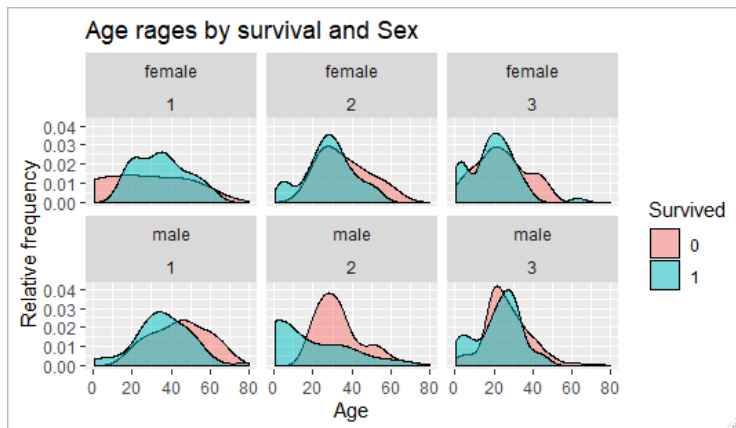
- Alpha ist die Transparenz; je höher um so niedriger
- Zusätzlich zum fill-command kann man color=<Gruppe> hinzufügen, dann werden die Ränder auch farbig. Size=... steuert die Dicke der Linien

6.6.1 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 rates by survival and Sex")
```



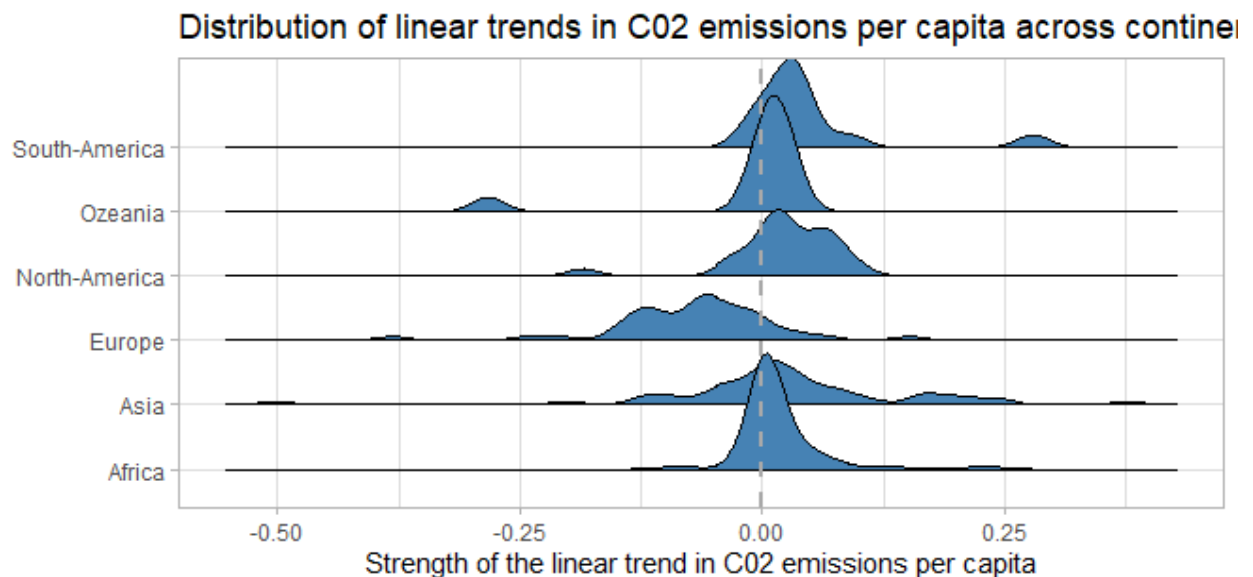
```
ggplot(titanic, aes(x = Age, fill = Survived)) +
  facet_wrap(Sex ~ Pclass) +
  geom_density(alpha = 0.5) +
  labs(y = "Relative frequency",
       x = "Age",
       title = "Age rages by survival and Sex")
```



6.7 Ridges plots

- Hierfür müssen die Daten ins longformat verwandelt werden

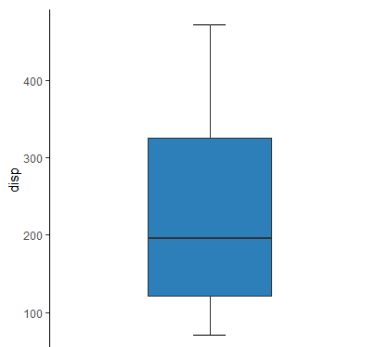
```
library(ggbridges)
reg_coefs1 %>%
  ggplot(aes(x = trend, y = continent, group = continent)) +
  geom_density_ridges2(fill="steelblue")+
  geom_vline(xintercept = 0, linetype = "dashed", color="darkgrey", size=1)+
  labs(x = "Strength of the linear trend in C02 emissions per capita",
       y="",
       title = "Distribution of linear trends in C02 emissions per capita
across continents")
```



6.8 Boxplots

6.8.1 Einfacher boxplot

```
mtcars %>%  
  ggplot(aes(y=disp)) +  
    stat_boxplot(geom="errorbar", width=.2) + #Querlinien an den Antennen  
    geom_boxplot(fill="#2c7fb8") +  
    theme_classic() +  
    theme(legend.position = "none") +  
    scale_x_discrete()
```



- **Optionen**

- Breite: `geom_box(width = .3)`
- Punkte hinzufügen: `geom_jitter(width = .2)` als weiteres geom

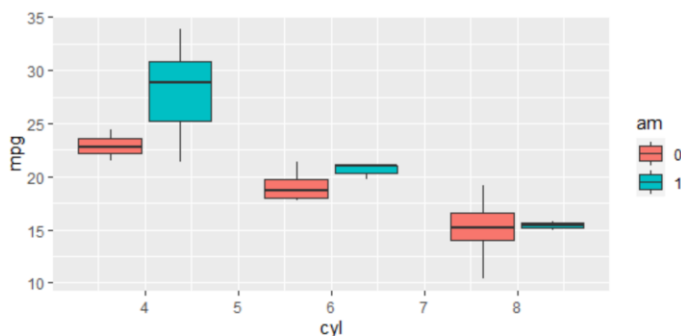
6.8.2 Gruppierte Boxplots

```
ggplot(mpg, aes(x = class, y = hwy)) +  
  geom_boxplot()
```

x = ist die Gruppierungsvariable. Ist die kontinuierlich, kann man `as.factor(class)` reinnehmen

Eine **Interaktion** bekommt man über

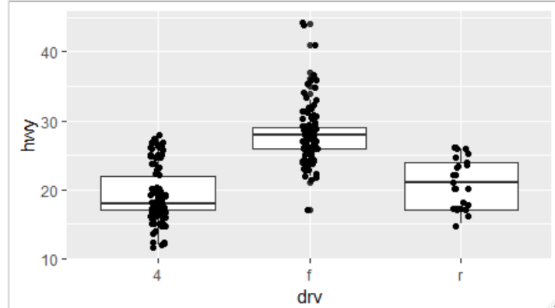
```
mtcars %>%  
  mutate(am = as.factor(am)) %>%  
  ggplot(aes(cyl,mpg, fill=am, group=interaction(am, cyl)))+  
  geom_boxplot()
```



(Ich hab keine Ahnung, was das bedeutet)

6.8.3 Hinzufügen der Punkte

```
ggplot(mpg, aes(drv, hwy)) +  
  geom_jitter(width = 0.05)
```



Das geht auch mit `geom_point`, aber dann hat man alle Punkte in überlappenden Linien

6.8.4 Optionen

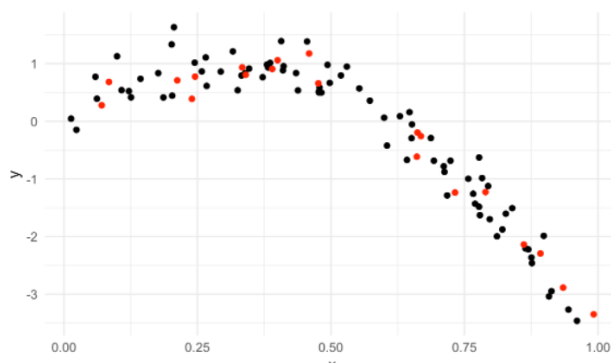
- Mit `coord_flip()` kann man ihn kippen (einfach als neue Zeile nach `geom_boxplot`)
- Gruppierte Boxplots kann man auch mit kontinuierlichen X-Variablen machen. Man muss sie nur zerstückeln: `geom_boxplot(aes(group = cut_width(carat, 0.1)))`
- Violin plots sind ähnlich: `geom_violin()`

7 Grafikparameter und Generelles

7.1 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")
```



7.2 Farben

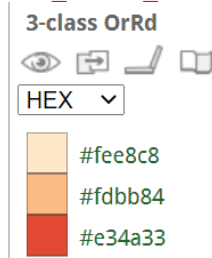
- Liste mit coolen Farben: <https://www.datanovia.com/en/blog/awesome-list-of-657-r-color-names/>

Hiermit kann an jede Grafik einfach ein Farbenname angehängt werden (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 antworten:

```
scale_color_manual(values = c("#74a9cf", "#d7301f"))
```



Da kann man natürlich Farben aus völlig unterschiedlichen Paletten zusammenstellen

7.3 Achsen

- Limits festlegen

...lim als weitere Zeile

```
xlim("f", "r") #kategoriale X-Achse  
ylim(20, 30)
```

Beispiel

```
ggplot(mtcars, aes(mpg, wt)) +  
  geom_point() +  
  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 Befehl fügt mit der Brechstange die Wert 1-10 ein. Wenn die x-Dimension von 1-20 geht, wird ab 11 nix angezeigt

- **Die Anzahl der angezeigten X-Werte auf der X-Achse erhöhen**

```
scale_x_continuous(breaks = scales::pretty_breaks(10))
```

- **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 gewünschten Wert

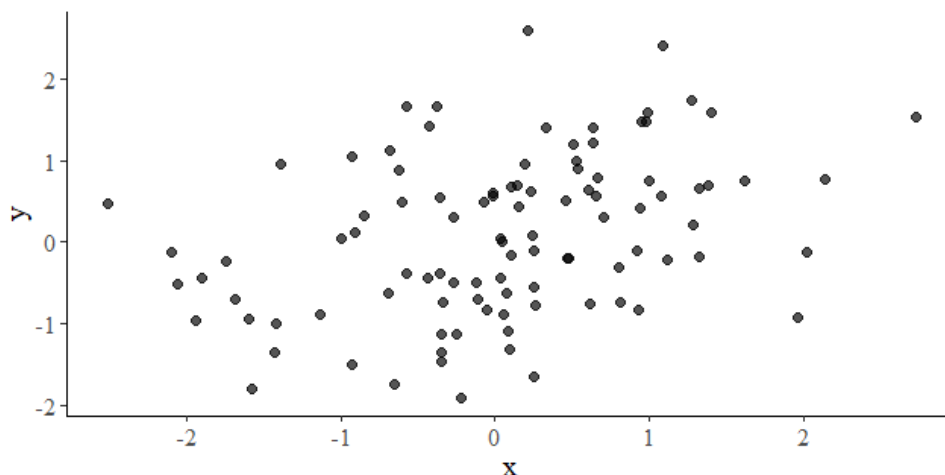
- **X-Achsen-Beschriftung um 90° rotieren**

```
...  
geom_line() +  
theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

7.4 Text und Legenden

7.4.1 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 = "A", size=15))
```



7.4.2 Unter- und Überschriften

Weiteres Argument in einer eigenen Zeile

```
geom_bar() +
labs(y = "Passenger Count",
     title = "Titanic Survival Rates",
     subtitle = "WTF")
```

7.4.3 Gemeinsame Überschrift bei Multiplots mit `gridExtra::grid.arrange`

```
gridExtra::grid.arrange(p1,p2, nrow=1, top = "Number of firms")
```

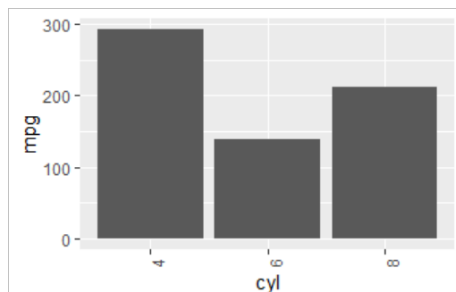
7.4.4 Weiterer Text

- <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 noch 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"))
```

### 7.4.5 Text rotieren

Wenn man Labels an einer X-Achse z.B. schräg oder vertikal 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!)

### 7.4.6 Legende entfernen

```
theme(legend.position = "none")
```

### 7.4.7 Text in der Legende ändern (Legendennamen und Kategorie-Namen

```
p + scale_color_discrete(name = "Type of Retweet",
 labels = c("Number of conspiracy retweets",
 "Overall number of retweets"))
```

Dieser command setzt voraus, dass die Gruppeneinteilung durch „color“ angefordert wurde. Alternativ `scale_fill_discrete()`

Man kann Legendentext und -Farbe auch zusammen adressieren:

```
scale_color_manual(values = c("#2c7fb8", "#d7301f"),
 name = "Traffic volume", labels = c("Low", "High"))
```

### 7.4.8 Position der Legende ändern

```
theme(legend.position = c(.3, .85))
```

- Linker Wert: Horizontale Lage
- Rechter Wert: Vertikale Lage

## 7.5 Themes

<https://www.datanovia.com/en/blog/ggplot-themes-gallery/>

```
library(ggthemes)
windowsFonts(Times=windowsFont("Times New Roman"))
```

- **Mehrere Grafiken nebeneinander oder untereinander:**

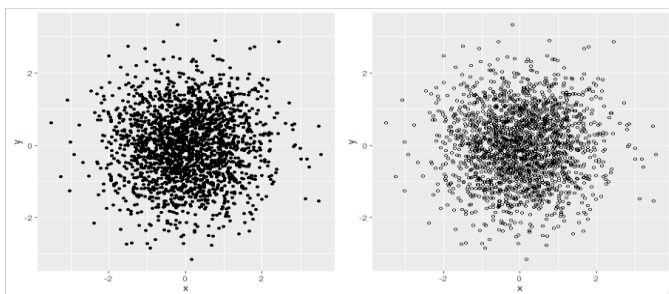
```
p1 = ggplot(mpg, aes(hwy)) + geom_histogram()
p2 = ggplot(mpg, aes(hwy)) + geom_freqpoly()
```

```
library(gridExtra)
grid.arrange(p1,p2,nrow=1)
```

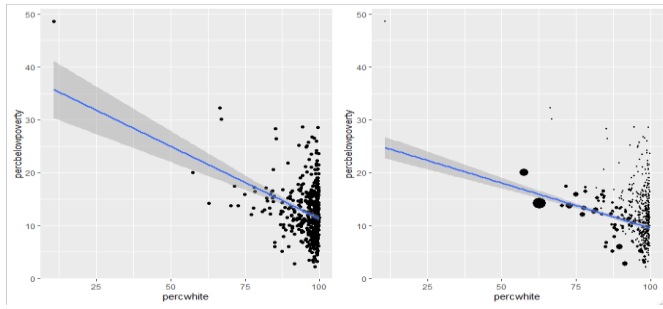
- **Zwei plots neben- oder untereinander**

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.

## 7.6 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)

## 8 Broom

### 8.1 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.

### 8.2 Funktionen

---

Broom ist dafür da, den output eines Models in einen tibble zu transformieren. Das passiert mit 3 Funktionen

#### 8.2.1 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

#### 8.2.2 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 hinzugefügt—wenn nicht, gibt es einen abgespeckten nur mit den Prädiktoren und dem Kriterium.

### 8.2.3 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()-Funktionen können alle im selben genesteten Datensatz enthalten sein!

| Country     | Data | Model | Glance | Tidy | Augment |
|-------------|------|-------|--------|------|---------|
| Afghanistan | <df> | <lm>  | <df>   | <df> | <df>    |
| Albania     | <df> | <lm>  | <df>   | <df> | <df>    |
| Algeria     | <df> | <lm>  | <df>   | <df> | <df>    |
| ...         | ...  | ...   | ...    | ...  | ...     |

Quelle: Wickhams [Video](#)

## 8.3 Genestete Daten

Mit den broom-Funktionen gehen auch meist genestete tibbles einher. Die obere Abbildung zeigt eine—in dieser können Daten, Modell-Merkmale, Koeffizienten 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>
```

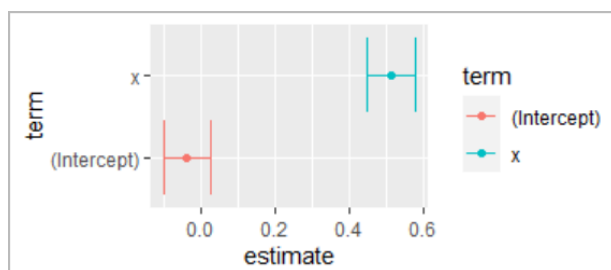
```
1 1 <tibble [487 x 1]>
2 2 <tibble [262 x 1]>
3 0 <tibble [251 x 1]>
```

## 8.4 Nutzen

### 8.4.1 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)

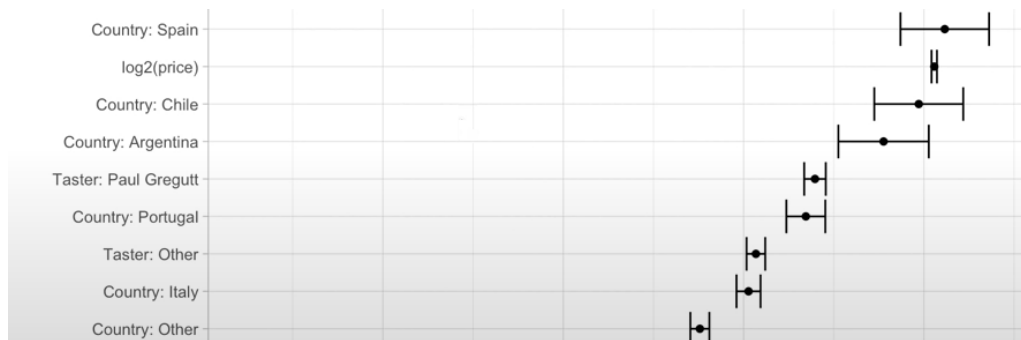
- In diesem Video macht er so einen plot aber verändert mit `str_replace()` die Begriff der Prädiktoren, was in dem Fall, wo er dummies benutzt sehr schick ist:

<https://www.youtube.com/watch?v=AQzZNIyWWM> 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_reorder(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 Regressionsmodells enthält) entlang des estimates ge-re-levelt (wobei das Käse ist weil er völlig unterschiedliche Effekte sortier, aber egal).

## 8.4.2 Kombinieren und Vergleichen verschiedener Modelle

### 8.4.2.1 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](#))



### 8.4.2.2 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

### 8.4.2.3 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 bedeutet, das eine Funktion über „etwas“ wiederholt durchgeführt wird:

- Im ersten Schritt 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, führt 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 Variablen (x,y) und 5 Koeffizienten

Durch unnest() werden die jetzt entpackt:

```

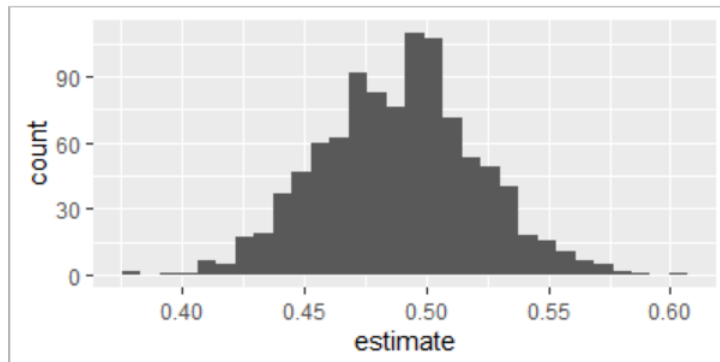
boot_coefs <- boot_models %>%
 unnest(coef.info)
A tibble: 2,000 x 8
 splits id model term estimate std.error statistic p.value
 <list> <chr> <lis> <chr> <dbl> <dbl> <dbl> <dbl>
1 <split [1~ Bootstrap~ <lm> (Inte~ 0.0222 0.0308 0.719 4.72e- 1
2 <split [1~ Bootstrap~ <lm> x 0.456 0.0313 14.6 1.07e-43
3 <split [1~ Bootstrap~ <lm> (Inte~ 0.00698 0.0315 0.221 8.25e- 1
4 <split [1~ Bootstrap~ <lm> x 0.464 0.0325 14.3 2.94e-42
5 <split [1~ Bootstrap~ <lm> (Inte~ -0.0264 0.0312 -0.846 3.98e- 1
6 <split [1~ Bootstrap~ <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)

#### 8.4.2.4 Manuelles bootstrapping indirekter Effekte

Hier mal der Code für ein bootstrapping eines indirekten Effekts aus zwei hintereinander geschalteten 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-Effekten

```
boot_models <- data_boot %>%
 mutate(modell1 = map(splits, ~ lm(y ~ m, data=.)),
 coef.info1 = map(modell1, 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)
```

