UNIVERSITÄT
LEIPZIG

Themenbezogene Tageskurse

Datenanalyse und Visualisierung mit R

Peter Kannewitz
peter.kannewitz@uni-leipzig.de

Christoph Mayer
christoph.mayer@uni-leipzig.de

25. Oktober 2024

Organisatorisches

Herzlich Willkommen im Kurs “Datenanalyse und Visualisierung mit R”!

- Alle Materialien finden ihr in unserem Moodle-Kurs
  - Folien
  - Projektordner für die Aufgaben
- für Uni-Externe stellen wir einen NextCloud share bereit
- Zusätzlich gibt es ein Handout zum Nachlesen

Organisatorisches

Uhrzeit	Inhalt
10:00 - 11:00	Wiederholung
10:00 - 10:15	Kaffeepause
11:15 - 12:15	Einführung Tidyverse
12:15 - 13:15	Mittagspause
13:15 - 14:15	Datenvorverarbeitung mit dplyr
14:15 - 14:30	Kaffeepause
14:30 - 16:00	Datenvisualisierung mit ggplot

Modulziele

- Paketfamilie `tidyverse` kennenlernen
- Datenbereinigung mit `dplyr` durchführen
- Funktionen mit Pipes aneinanderketten
- komplexe Abbildungen mit `ggplot2` erstellen
- mithilfe von geeigneten Ressourcen (Cheat Sheets, Handout, ...) selbst weiterlernen

Hinweise zu den PCs im Pool:

- Die PCs können mit dem Uni-Login und dem entsprechenden Passwort genutzt werden.
- **Empfehlung:** Eigene Laptops können gerne mitgebracht und genutzt werden.
- Speichert Eure Daten nicht **nur** auf den PCs im Pool.

Wiederholung

Was wir bisher kennen (sollten)

- Arbeiten mit RStudio
  - Skripte vs. Konsole
  - globale Umgebung
  - Speichern von Skripten
  - Projekte
- einfache Datentypen
  - Umwandlungen und Besonderheiten
  - logical > integer > numeric > character
- strukturierte Datentypen
  - **Vektor**, **Data Frame**, Matrix, Liste
  - Erstellen von Vektoren
- Objekte vs. Funktionen
  - Zuweisung von Objekten mit `<-`
  - Anwendung von Funktionen auf Objekte
- Funktionen
  - Argumente (Objekte + Optionen)
  - Argumente benennen / Reihenfolge von Argumenten
  - Hilfeseite
  - Default Werte von Argumenten
- Packages
  - Installieren und laden
- Daten einlesen
  - `read*` -Funktionen
- einfach statistische Kennwerte
  - `mean()`, `sd()`, (`t.test()`)

Funktionen aneinander ketten

Verschachtelung

Funktionen können miteinander verschachtelt werden. R arbeitet sich hierbei von *innen* nach *außen* vor.

```
round(log(abs(mean(c(-1,-2,-3,-4)))))
```

```
## [1] 1
```

Aufgabe

Gegeben sei ein Vektor, der die Zahlen von 1-20 enthält.

Erstellt diesen Vektor und berechnet die exponierte (`exp()`) gerundete (`round()`) Standardabweichung (`sd()`) des Vektors.

Lösung

```
x <- 1:20
exp(round(sd(x)))
```

```
## [1] 403.4288
```

Pipes |>

Alternativ zu geschachtelten Funktionen können Pipes verwendet werden.[1]

Prinzip:

- Jede Funktion gibt als Ergebnis ein Objekt aus.
- Das hieraus entstandene Objekt, soll wieder einer Funktion **als erstes Argument** übergeben werden.
- Auch das Ergebnis dieser Funktion soll wieder einer Funktion übergeben werden
- usw.

[1] Nativ verfügbar ab R Version 4.1.0. sonst im Paket `magrittr`

Pipes |>

Anwendungsbeispiel:

```
c(-1,-2,-3,-4) |>
  mean() |>
  abs() |>
  log() |>
  round()
```

```
## [1] 1
```

Diese Möglichkeit wird aufgrund ihrer Lesbarkeit präferiert!

- es existieren mittlerweile 2 so genannte Pipe-Operator:
  - |> ist die base Version, also ohne Pakete verwendbar
  - %>% ist aus dem tidyverse
- Pipes senden das Vorhergehende immer als erstes Argument in die nächste Funktion
- so enststehen “Piplines”

Aufgabe

Gegeben sei ein Vektor, der die Zahlen von 1-20 enthält.

Erstellt diesen Vektor und berechnet die exponierte (`exp()`) gerundete (`round()`) Standardabweichung (`sd()`) des Vektors **mithilfe von Pipes**.

Lösung

```
1:20 |>
  sd() |>
  round() |>
  exp()
```

```
## [1] 403.4288
```

15min Kaffeepause!

Die Paketsammlung Tidyverse

Das Tidyverse ist eine Sammlung von Packages, welche einer gemeinsamen Grammatik folgen.

Tidyverse laden

```
install.packages("tidyverse") ## nur falls nötig!

library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

Motivation

- komplexe statistische Berechnungen sind mit R meist sehr einfach umzusetzen
  - statistische Hypothesentests
  - Diagramme bzw. graphische Darstellungen
  - (lineare) Regressionsmodelle bzw. ANCOVA, ANOVA, ...
  - Mashine Learning Verfahren (Random Forest, Neural Networks, ...)
- Voraussetzung für alle weiteren Schritte ist jedoch **immer**, dass der Datensatz **bereinigt** wurde!
- die Datenbereinigung (auch data wrangling) mit dem `tidyverse` wollen wir uns in dieser Sitzung anschauen

Mit den `dplyr::` Funktionen lassen sich komplexe Datenbereinigungen einfach umsetzen. Die Datenaufbereitung nimmt in vielen Projekten die meiste Zeit in Anpsruch. Für das Paket existiert auch ein sehr gutes Cheatsheet.

Hinweis

Das Paket `dplyr::` erwartet, dass ihr sogenannte ”tidy data” vorliegen habt. Das bedeutet, dass **jede Spalte eine Variable** und **jede Zeile eine Beobachtung/Fall** ist.

Daten verfügbar machen

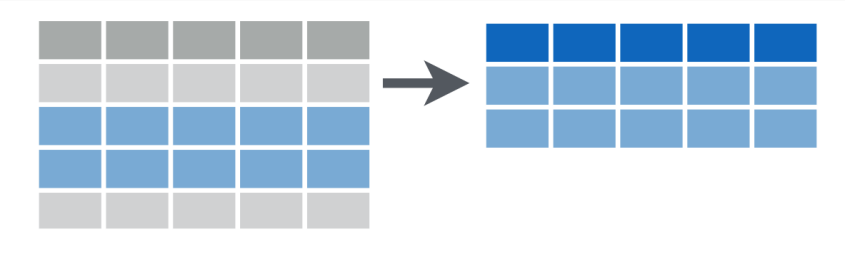
An diesem Datensatz werden wir die verschiedenen `dplyr::` Funktionen erproben.

```
starwars <- dplyr::starwars
glimpse(starwars)
```

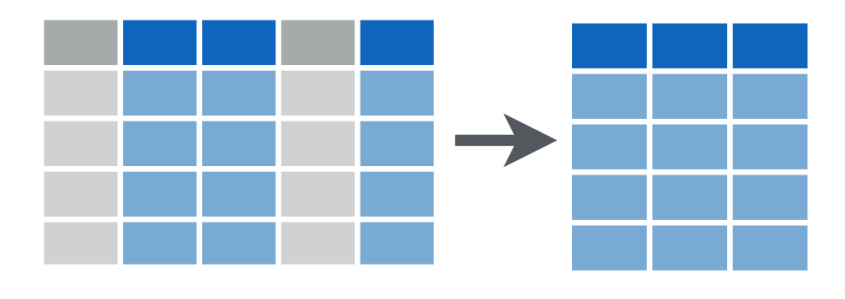
```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or~
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
## $ films      <list> <"A New Hope", "The Empire Strikes Back", "Return of the J~
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

`dplyr::filter()` und `dplyr::select()`

- mit `filter()` wählen wir Fälle nach bestimmten Bedingungen aus
- es werden **Zeilen** ausgeschlossen und der Datensatz wird “kürzer”



- mit `select()` wählen wir Variablen
- es werden **Spalten** ausgeschlossen und der Datensatz wird “schmaler”



Warum?

- oft starten wir in die Datenanalyse mit einem großen Datensatz
- meist interessieren uns nur einige Variablen und bestimmte Fälle (Personen)
- Ziel der Datenbereinigung ist es, einen Datensatz zu erstellen, der (nur) die Variablen und Fälle beinhaltet, die wir für die Analyse benötigen
    - bessere Übersichtlichkeit
    - weniger Speicherplatz nutzen
    - schnelleres und einfacheres Arbeiten
    - einheitliche Stichprobe

Beispiel

```
starwars |>
  dim()
```

```
## [1] 87 14
```

Beispiel

- Wir interessieren uns **nicht** für die Variablen ‘films‘, ‘vehicles‘ und ‘starships‘
- Wir wollen nur Charaktere, die größer als 1 m sind und über 80 kg wiegen

```
starwars |>
  select(-c(films, vehicles, starships)) |>
  filter(
    height > 100,
    mass > 80
    ) |>
  dim()
```

```
## [1] 21 11
```

Fälle filtern

```
starwars |>
  filter(!is.na(species)) |>
  dim()
```

```
## [1] 83 14
```

Frage

Wonach filtert dieser Code unseren Datensatz?

Übung

Angenommen, wir würden nur Charaktere untersuchen wollen, welche gelbe Augen haben **und** über 80 kg wiegen. Wie viele Fälle/Untersuchungseinheiten hätten wir dann noch übrig?

Lösung

```
starwars |>
  filter(eye_color == "yellow", mass > 80)
```

```
## # A tibble: 3 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Darth Va~    202   136 none       white      yellow          41.9 male  mascu~
## 2 Ki-Adi-M~    198    82 white      pale       yellow          92   male  mascu~
## 3 Dexter J~    198   102 none       brown      yellow          NA   male  mascu~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Tipps

- innerhalb von `filter()` können logische Operatoren (&, |, !, ==, !=, <, >, usw.) verwendet werden
- innerhalb von `select()` können einige hilfreiche Hilfsfunktionen verwendet werden (`?dplyr_tidy_select`)
  - oft verwendet werden z.B. `starts_with()` oder `where()`

```
starwars |>
  select(where(is.numeric)) |>
  dim()
```

```
## [1] 87  3
```

Variablen manipulieren mit `dplyr::mutate()`

- mit `mutate()` können wir Variablen in unserem Data Frame berechnen
- innerhalb von `mutate()` geben wir immer Folgendes an:
  - `neue_variable = berechnung(alte_variable)`
  - ist der Name der neuen Variable gleich einer existierenden Variable, so wird diese überschrieben
  - ist der Variablenname neu, so fügen wir unserem Data Frame eine Variable hinzu

```
starwars |>
  mutate(
    height_in_meters = height / 100, # Höhe in Meter umrechnen
    index = 1:n() # fortlaufende Nummer
    ) |>
  dim()
```

```
## [1] 87 16
```

Funktionen in `dplyr::mutate()`

Innerhalb von `mutate()` können viele Funktionen zur Erstellung einer Variable angewendet werden. Oft hilfreich sind `recode()` oder `case_when()`.

```
starwars |>
  mutate(
    gender = recode(
      gender,
      "masculine" = "männlich",
      "feminine" = "weiblich"
      ) |>
      as_factor(),
    age_at_by = case_when(
      birth_year >= 50 ~ "50 und älter",
      birth_year < 50 ~ "unter 50",
      is.na(birth_year) ~ NA_character_
    )
  ) |>
  select(gender, birth_year, age_at_by) |>
  head(n = 3)
```

```
## # A tibble: 3 x 3
##   gender   birth_year age_at_by
##   <fct>         <dbl> <chr>
## 1 männlich         19 unter 50
## 2 männlich        112 50 und älter
## 3 männlich         33 unter 50
```

Aufgabe

Was passiert, wenn ihr statt der Funktion `mutate()` die Funktion `transmute()` benutzt?

Wann könnte dieses Verhalten von Vorteil sein?

Lösung

```
starwars |>
  transmute(
    height_in_meters = height / 100,
    index = 1:n()
    ) |>
  dim()
```

```
## [1] 87  2
```

- `transmute()` funktioniert genau so wie `mutate()`
- bei `transmute()` werden nur die innerhalb des Befehls erstellten Variablen behalten
- `mutate()` fügt die neuen Variablen zu den alten hinzu

60min Mittagspause!

Fälle Gruppieren

Am wohl unscheinbarsten ist die Funktion `group_by()`, da sie unseren Datensatz oberflächlich betrachtet gar nicht verändert.

```
starwars |>
  group_by(homeworld) |>
  head(5) # nimm die ersten fünf Fälle
```

```
## # A tibble: 5 x 14
## # Groups:   homeworld [3]
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Luke Sky~    172    77 blond      fair       blue            19   male  mascu~
## 2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu~
## 3 R2-D2         96    32 <NA>       white, bl~ red             33   none  mascu~
## 4 Darth Va~    202   136 none       white      yellow          41.9 male  mascu~
## 5 Leia Org~    150    49 brown      light      brown           19   fema~ femin~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Im Hintergrund haben wir unseren Tibble aber um das Attribut **group** ergänzt, welches jeweils die Zeilennummern für die Fälle einer Gruppe als integer Vector speichert.

```
starwars |>
  group_by(homeworld) |>
  attr("group") |>
  head(5)
```

```
## # A tibble: 5 x 2
##   homeworld            .rows
##   <chr>          <list<int>>
## 1 Alderaan               [3]
## 2 Aleen Minor            [1]
## 3 Bespin                 [1]
## 4 Bestine IV             [1]
## 5 Cato Neimoidia         [1]
```

Fälle Gruppieren

Durch das nun unserem Datensatz beigefügte Attribut verhalten sich nun folgende Tidyverse Funktionen teilweise anders. Dies ermöglicht es uns, Gruppeneigenschaften zu ermitteln:

```
starwars |>
  group_by(homeworld) |>
  summarise(
    avg_height = mean(height, na.rm = TRUE),
    n = n()
  ) |>
  arrange(desc(n)) |> # absteigend sortiert nach Anzahl
  head(3)
```

```
## # A tibble: 3 x 3
##   homeworld avg_height     n
##   <chr>          <dbl> <int>
## 1 Naboo           177.    11
## 2 Tatooine        170.    10
## 3 <NA>            139.    10
```

Finale Aufgabe

Manchmal frage ich mich, welche drei Arten (species) in Star Wars wohl im Durchschnitt am schwersten sind, wenn ich nur die Fälle betrachten würde, welche eindeutig einer Art (species) und einem Gewicht zugeordnet werden können sowie mindestens zwei Fälle für die jeweilige Art vorhanden sind. Leider habe ich nie eine Antwort darauf gefunden.

Könnt ihr mir helfen?

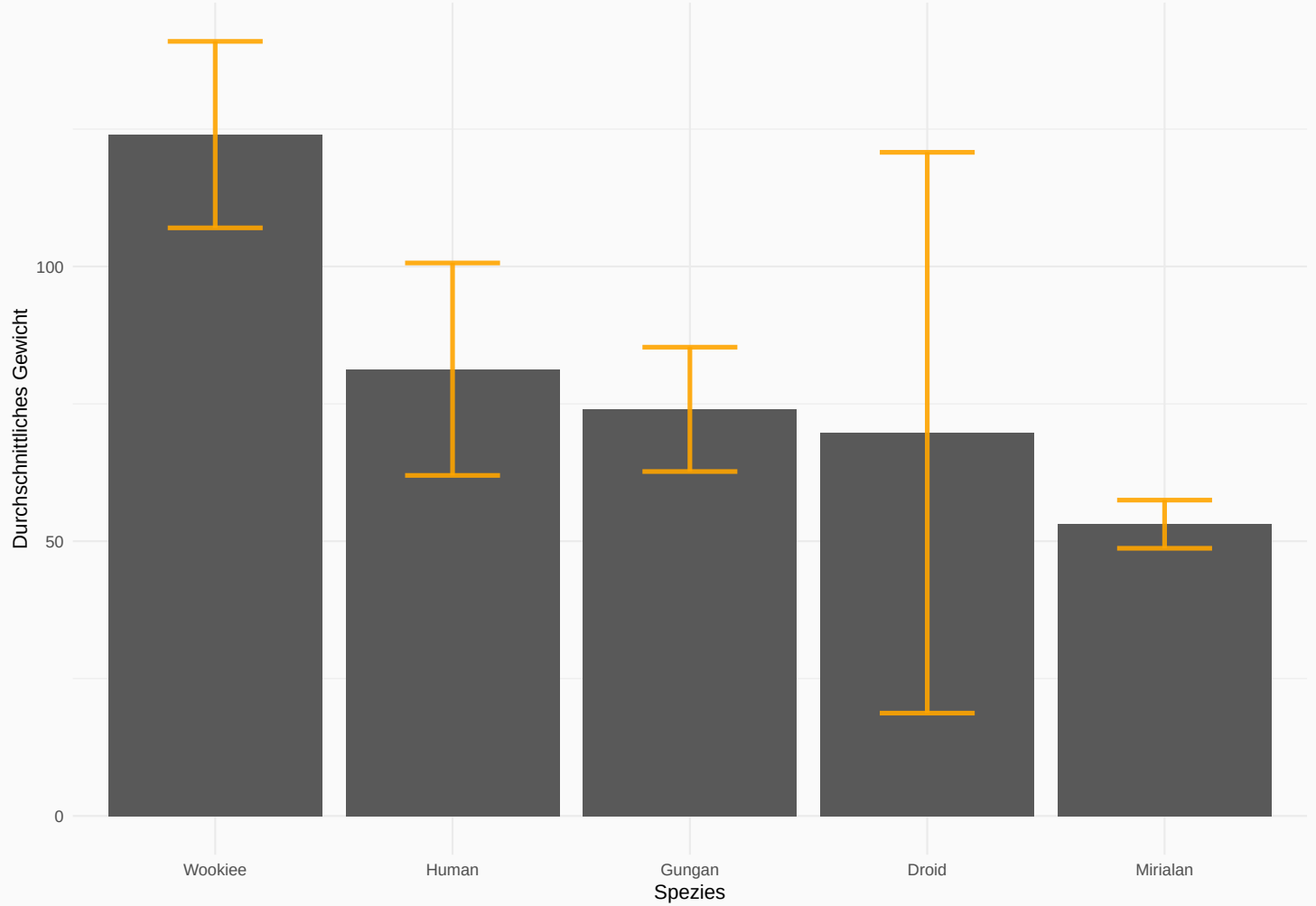
Lösung

```
starwars |>
  filter(!is.na(mass) & !is.na(species)) |>
  # "welche eindeutig einer Art (species) und einem Gewicht
  #  zugeordnet werden können"
  group_by(species) |>
  # "Arten (species) in Star Wars [...] im Durchschnitt"
  summarise(
    avg_weight = mean(mass, na.rm = TRUE),
    # "im Durschnitt am schwersten"
    n = n()
  ) |>
  filter(n >= 2) |>
  # "mindestens zwei für die jeweilige Art vorhanden sind"
  slice_max(avg_weight, n = 3) # "welche drei Arten"
```

```
## # A tibble: 3 x 3
##   species avg_weight     n
##   <chr>        <dbl> <int>
## 1 Wookiee      124       2
## 2 Human         81.3    20
## 3 Gungan        74       2
```

Ausblick

Nächste Sitzung wollen wir uns dann mit auseinandersetzen, wie wir die gerade gelernten Prinzipien nahtlos in wunderschöne und informative Grafiken übersetzen können.



Fragen?

15 min Kaffeepause!

Grafiken

Mit R lassen sich Grafiken auf ganz unterschiedliche Weisen erstellen:

1. Mit dem base R Grafiksystem lassen sich schnell und unkompliziert vollständige (High-Level) Grafiken erzeugen.
2. Mit dem base R Grafiksystem lassen sich jedoch auch (Low-Level) Grafiken von Grund auf aufbauen und können sehr individuell gestaltet werden.
3. Mit verschiedenen Packages lassen sich mit vertretbarem Aufwand ansprechende und beliebig komplexe (Medium-Level) Grafiken erstellen. Hierunter zählt z. B. `ggplot2` aus dem `tidyverse`.

Vorbereitung:

```
df_regio <- readRDS(file = "data/regio.rds") |>
  mutate(Region = if_else(condition = Regio_Code < 11000,
                          true = "Alte Bundesländer",
                          false = "Neue Bundesländer"),
         Region = as_factor(Region),
         Durchschnittsalter_cat = cut_interval(Durchschnittsalter, n = 3))
```

Je nach Skalennieveau der Variablen sind unterschiedliche Darstellungen sinnvoll.

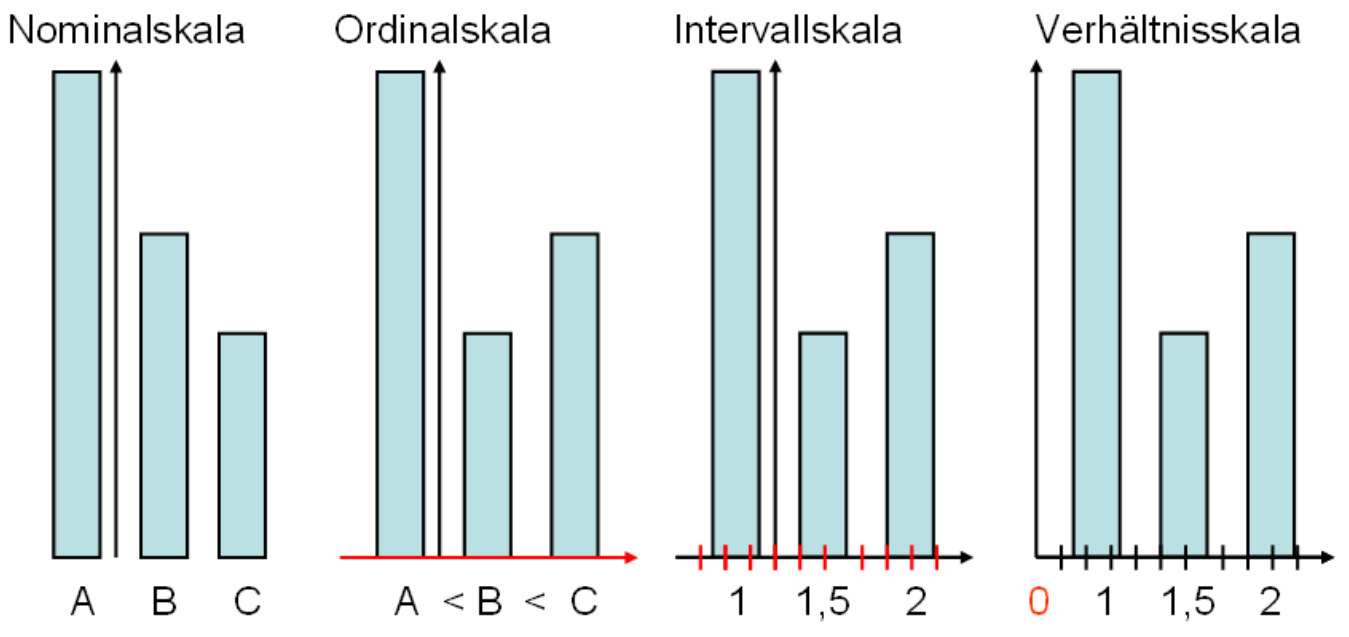


Figure 1: CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=724035

Skalenniveaus Operationen und Beispiele

Skalenniveau	Mögliche Operationen	Beispiele
Nominalskala	$=, \neq$	Postleitzahlen, Geschlechter
Ordinalskala	$=, \neq, <, >$	Schulnoten, Tabellenplatz in der Bundesliga
Intervallskala	$=, \neq, <, >, +, -$	Zeitskala (Datum), Intelligenzquotient
Verhältnisskala	$=, \neq, <, >, +, -, \times, \div$	Alter (in Jahren), Umsatz (in Euro)

Übernommen von https://de.wikipedia.org/wiki/Skalenniveau.

GGplot

`GGPlot2` ist ein mächtiges Grafik Package, um mit mittlerem Aufwand sehr ansprechende Plots zu produzieren.

Die Logik des Aufbaus von Grafiken lehnt sich an die Ideen von Leland Wilkinson (**The Grammar of Graphics**) an.

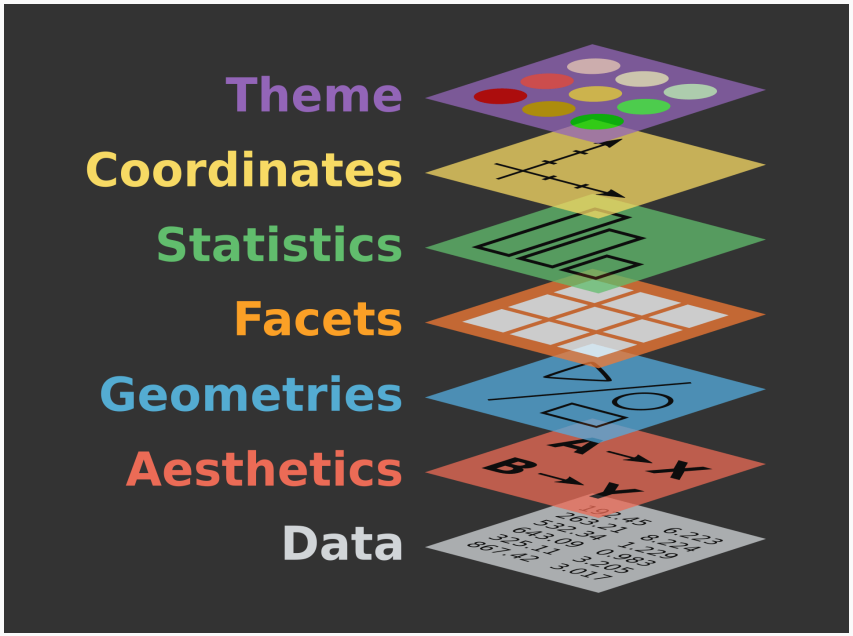


Figure 2: Quelle: http://r.qcbs.ca/workshop03/book-en/grammar-of-graphics-gg-basics.html

Daten

Zunächst übergeben wir den ersten “Layer” der Grafik: die **Daten**.

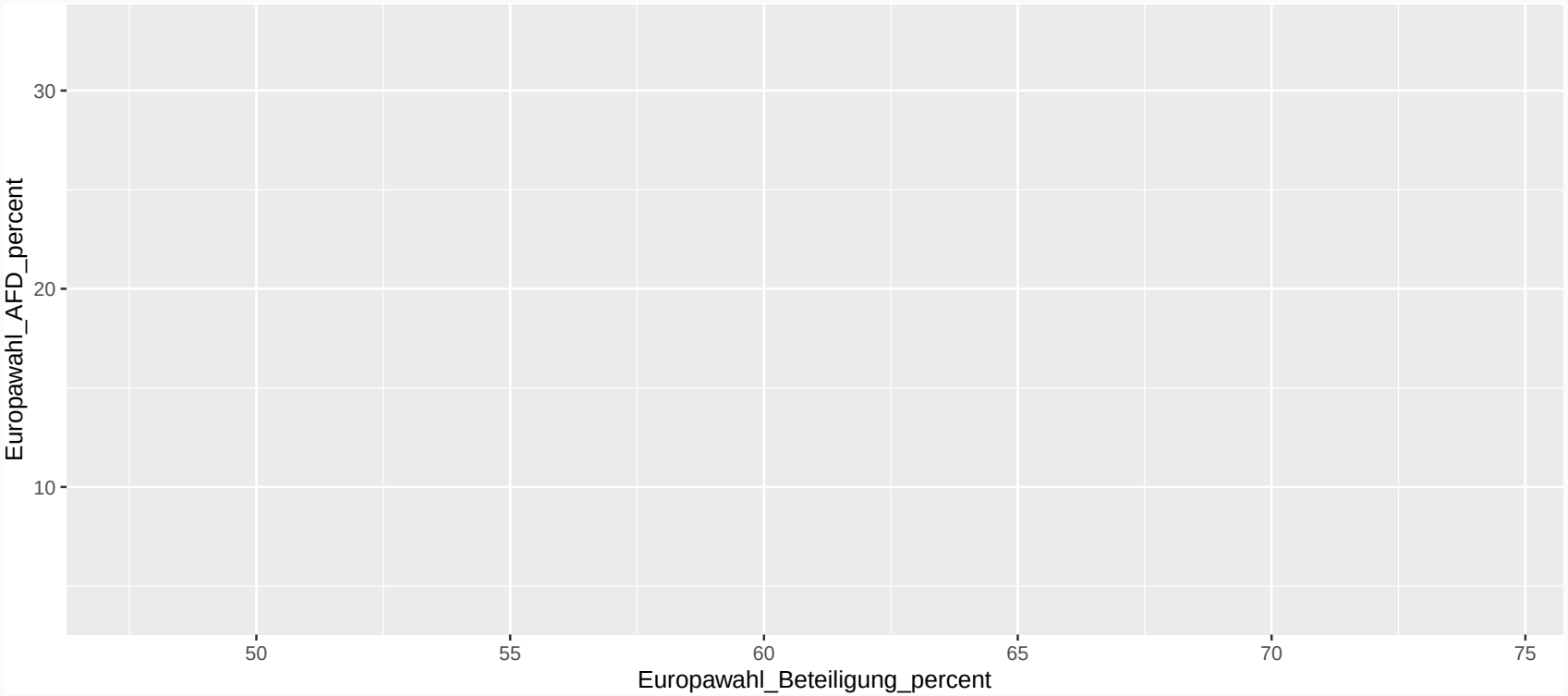
Es passiert augenscheinlich noch nichts. Tatsächlich werden aber die Daten bereits hinterlegt.

```
ggplot(data = df_regio)
```

Aesthetics

Im nächsten Schritt werden über das sogenannte *Mapping* die **Aesthetics** festgelegt. Hierbei wird definiert, welche Merkmale der Daten worauf projiziert werden sollen (z.B. Achsen, Farbe, Größe, etc.).

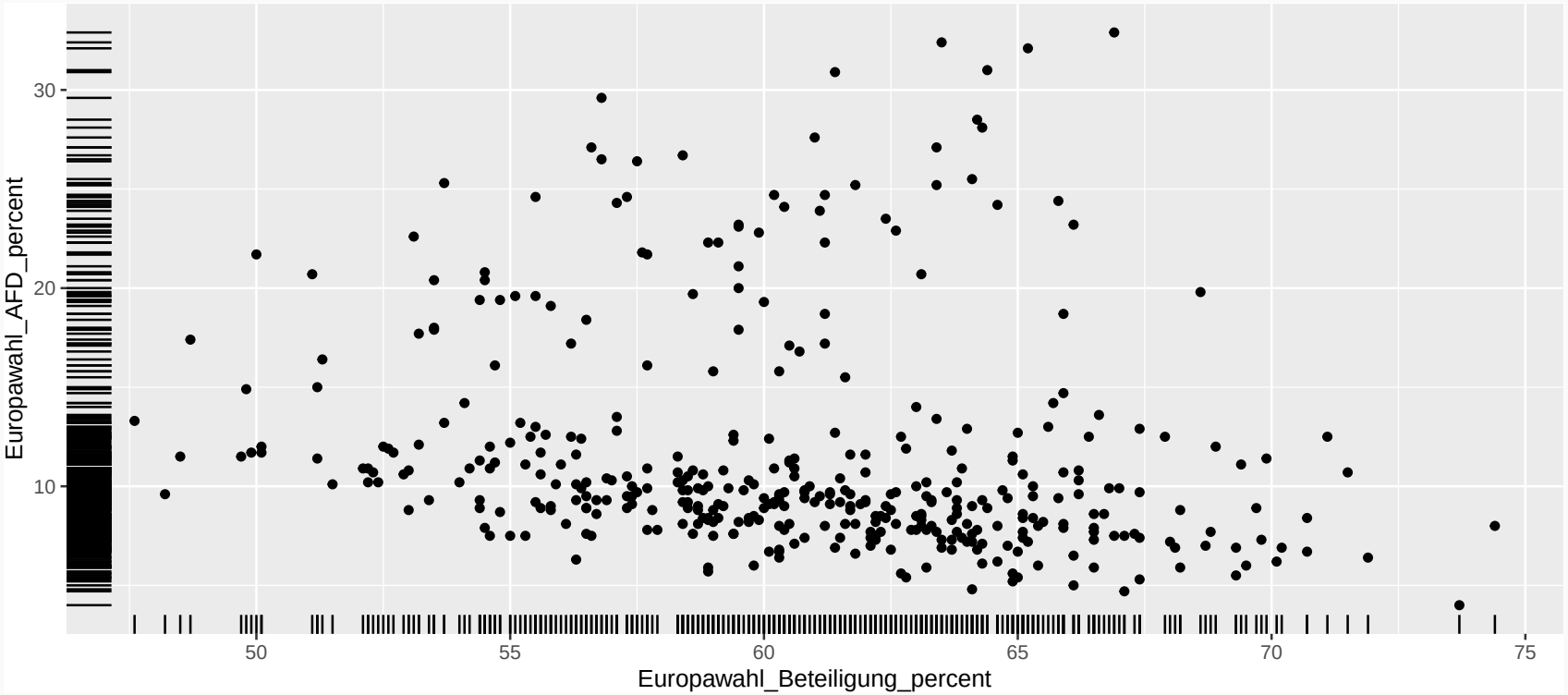
```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent))
```



Geome

- “Geome” bestimmen die konkrete Darstellungsform der Daten
- einzelne Geome werden mit einem “+” an das Basislayer gehangen
    - Reihenfolge ist dabei wichtig, weil Geome übereinander gelegt werden

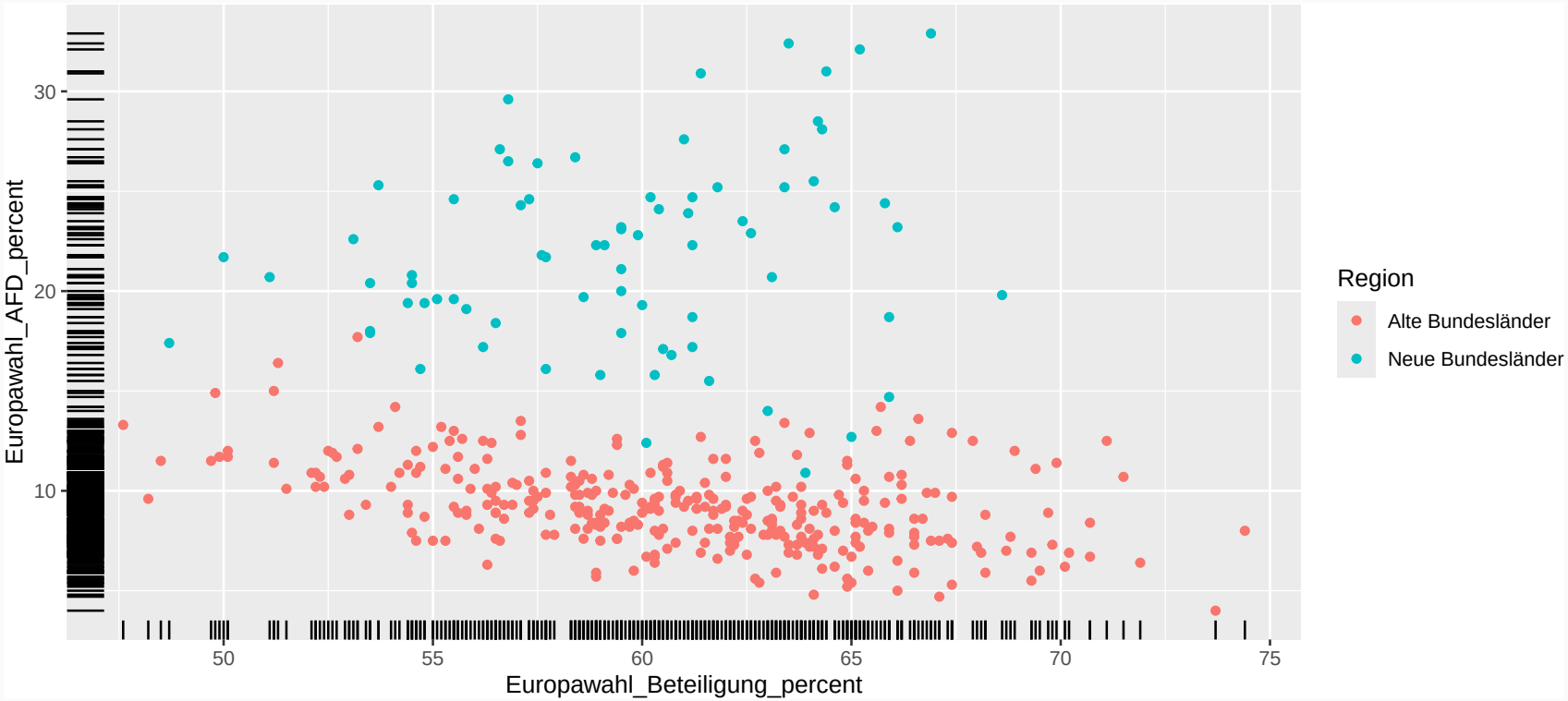
```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point() +
  geom_rug()
```



- Aesthetics können entweder **global*** für den gesamten Plot oder **lokal** für einzelne Geome festgelegt werden

Beispiel: Die Farbe der Punkte soll sich nach alten und neuen Bundesländern unterscheiden, aber nur für die Punkte, nicht für die Randverteilung.

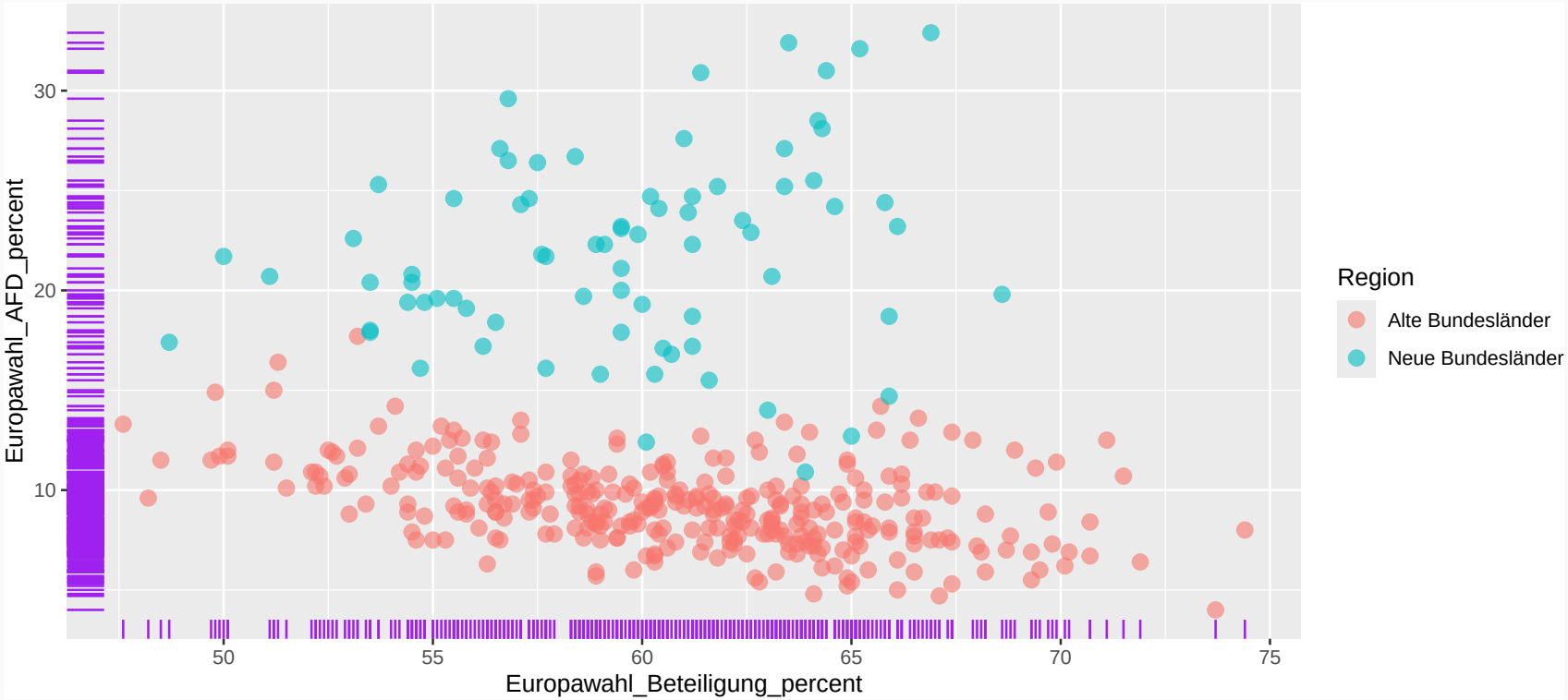
```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point(mapping = aes(color = Region)) +
  geom_rug()
```



- gleichzeitig können statische ästhetische Eigenschaften festgelegt werden, die nicht an Daten geknüpft sind

Beispiel: Die Randverteilung im Plot soll lila (`color`) dargestellt werden und die Punkte sollen durchsichtiger (`alpha`) und größer (`size`) werden.

```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point(mapping = aes(color = Region),
             alpha = 0.6,
             size = 3) +
  geom_rug(color = "purple")
```



Es existieren eine Vielzahl von Geomen. Hier eine kleine Übersicht über die wichtigsten:

Geom	Geometrische Objekt
`geom_points()`	Datenpunkte (Scatter-Plot)
`geom_jitter()`	Datenpunkte (Jitter-Plot)
`geom_bar()`	Balken
`geom_histogram()`	Histogramm
`geom_boxplot()`	Boxplot
`geom_smooth()`	Fit eines spezifizierten Zusammenhangs
`geom_rug()`	Marginale Verteilungen

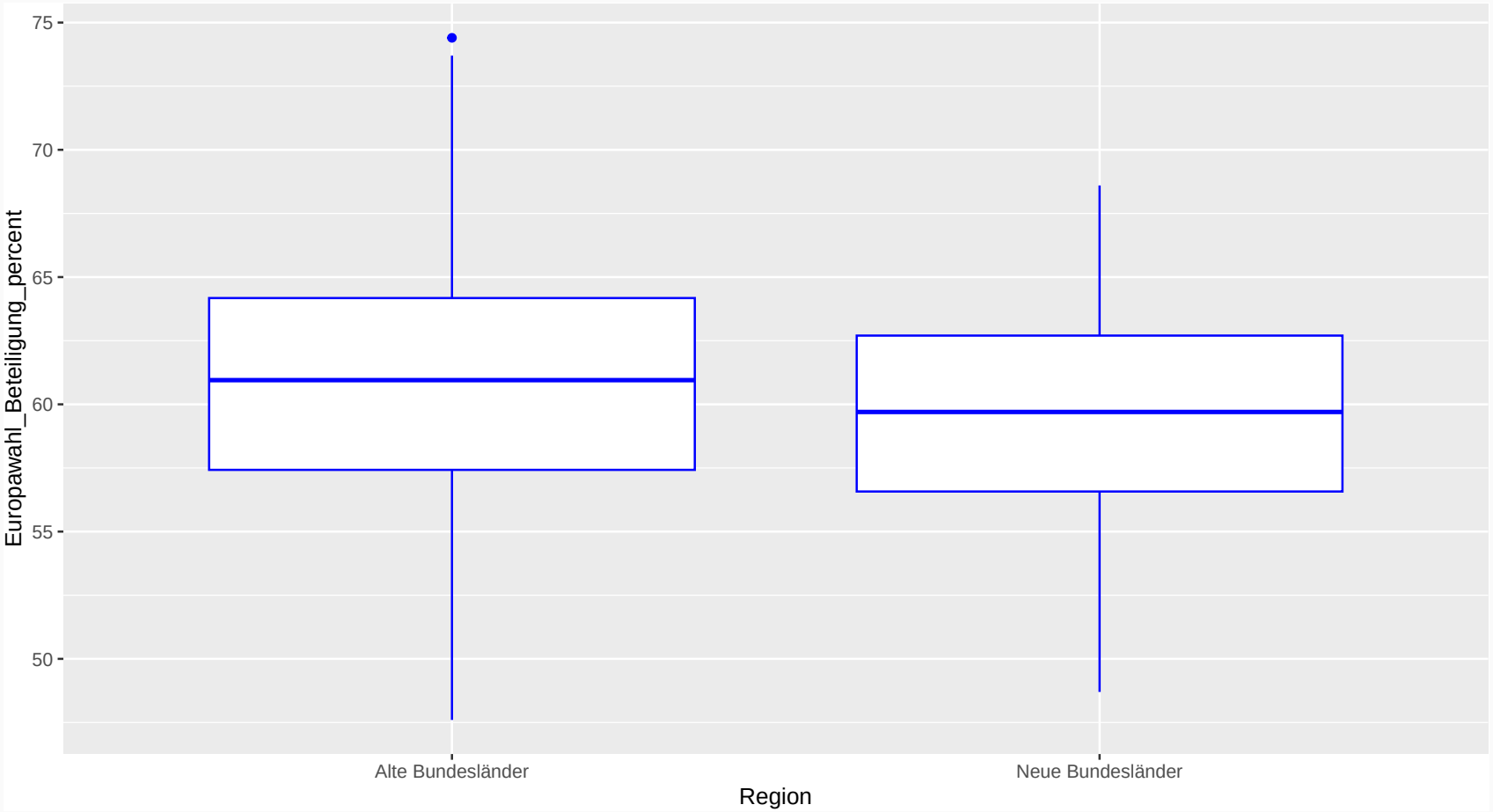
Für weitere Geome siehe Cheat Sheet.

Aufgabe

Erstellt eine Boxplotdarstellung (`geom_boxplot`) für die Wahlbeteiligung bei der Europawahl. Unterscheidet nach alten und neuen Bundesländern. Färbt zusätzlich die Boxplots blau.

Lösung

```
df_regio |>
  ggplot(mapping = aes(x = Region,
                       y = Europawahl_Beteiligung_percent)) +
  geom_boxplot(color="blue")
```



Achsenbeschriftung und Themes

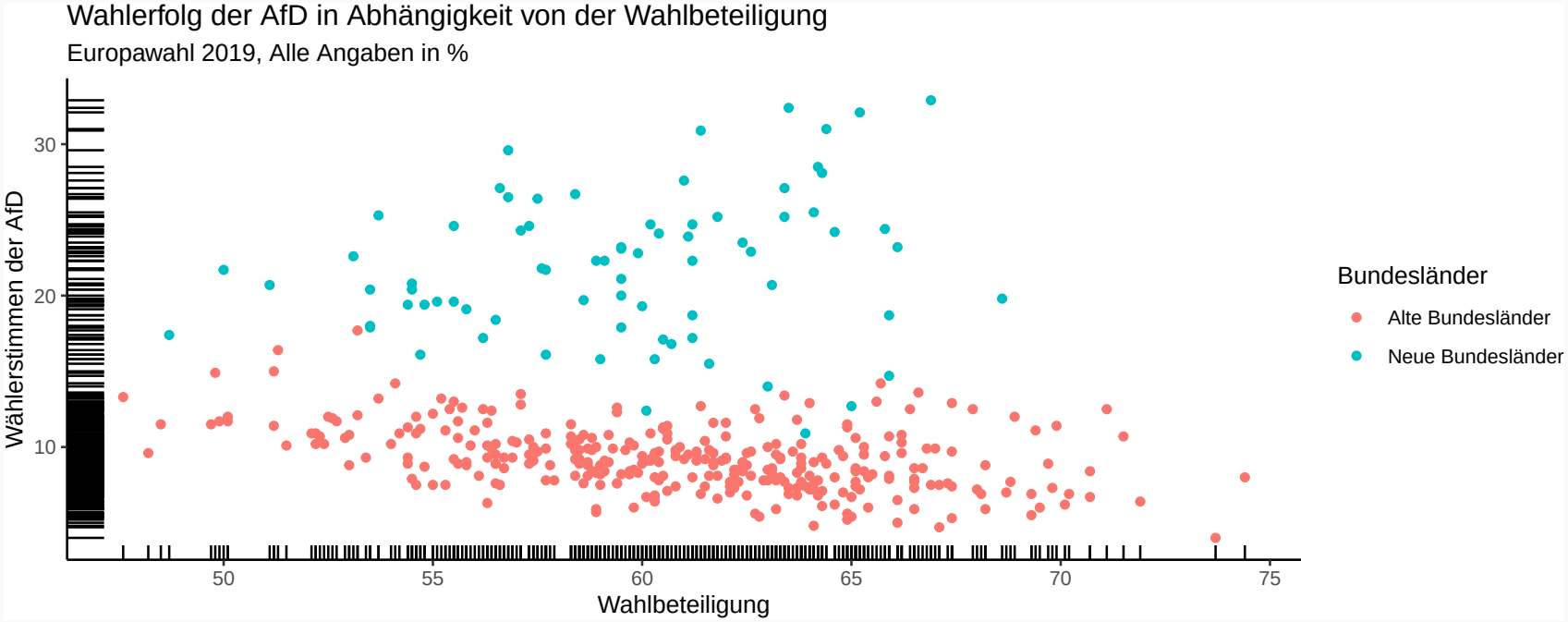
- die Beschriftung der Achsen stellt ein weiteres Layer dar
- mit `labs()` können zum Beispiel folgende Sachen angepasst werden:
  - Titel: `title = ""`
  - Untertitel: `subtitle = ""`
  - Legendenüberschrift: `color = ""` (Achtung! Hängt von den verwendeten Aesthetics ab)
  - Achsenbeschriftung: `x = "", y = ""`
  - ...
- allgemeine Erscheinungsbild lässt sich mit Themes verändern (das Paket `{ggtheme}` stellt Zahlreiche Themes bereit)

Hinweis:

Weitere Eigenschaften lassen sich über `theme()` manipulieren. Diese Vorgehensweise ist aber deutlich komplizierter und erfordert ein wenig Erfahrung im Umgang mit diversen Online-Suchmaschinen ;-).

Globale Einstellungen

```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point(mapping = aes(color = Region)) +
  geom_rug() +
  labs(title = "Wahlerfolg der AfD in Abhängigkeit von der Wahlbeteiligung",
       subtitle = "Europawahl 2019, Alle Angaben in %",
       x = "Wahlbeteiligung", y = "Wählerstimmen der AfD",
       color = "Bundesländer") +
  theme_classic()
```



Plots als Objekte

GGPlot Plots sind in R nichts anderes als umfangreiche Listen vom Typ “ggplot”.

```
ggplot(data = df_regio) |> class()
```

```
## [1] "gg"     "ggplot"
```

Daher können wir auch Plots an Namen binden und im Nachhinein entsprechend aufrufen:

```
base_plot <- ggplot(data = df_regio,
                    mapping = aes(x = Europawahl_Beteiligung_percent,
                                  y = Europawahl_AFD_percent))
```

Speichern von Grafiken mit GGSave

- mit der Funktion `ggsave()` können GGPlot Objekte auf *reproduzierbare* Art und Weise exportiert werden
- vor dem Speichern muss die Grafik als Objekt an einen Namen gebunden werden (oder entsprechend gepiped)
- die Endung des Files gibt das Dateiformat vor

Speichern von Grafiken mit GGSave

```
plot_save <-
  ggplot(data = df_regio,
         mapping = aes(x = Europawahl_Beteiligung_percent,
                       y = Europawahl_AFD_percent)) +
  geom_point()

ggsave(plot = plot_save,
       filename = "plots/plot.pdf",
       height = 7,
       width = 8,
       units = "in")
```

Aufgabe

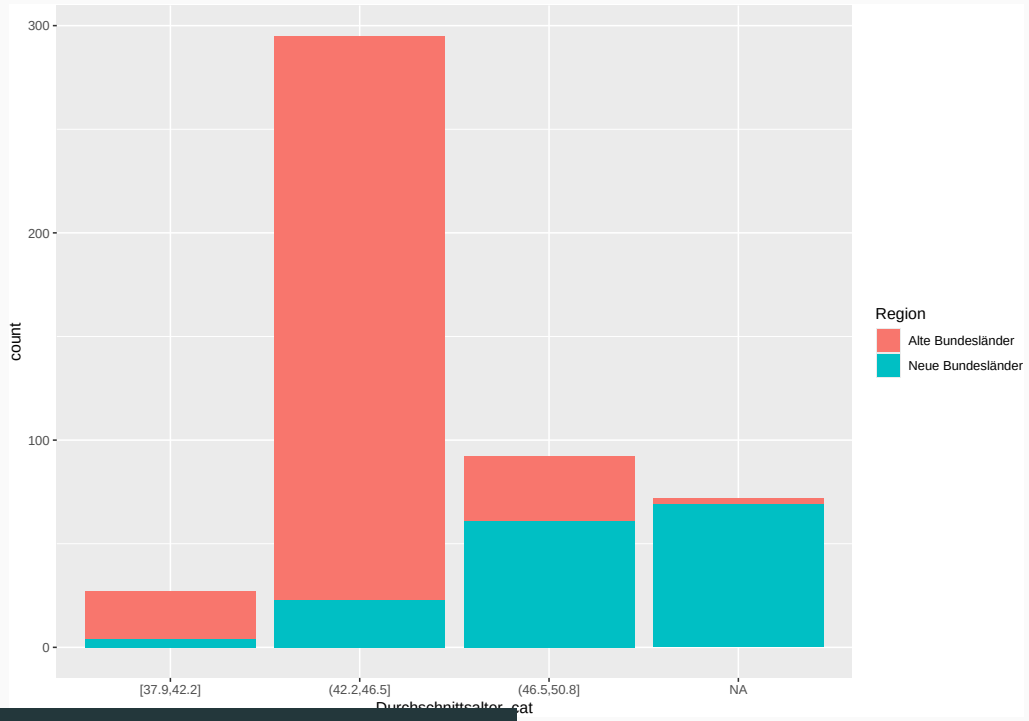
Erstellt ein Balkendiagramm für die Durschnittsalterkategorien (`Durchschnittsalter_cat`). Färbt die Balken nach der Region (`Region`).

Speichert diese Grafik im Anschluss in dem Ordner `plots` unter dem Namen `age_region.pdf`.

```
age_region <- df_regio |>
  ggplot(mapping = aes(x = Durchschnittsalter_cat)) +
  geom_bar(mapping = aes(fill = Region))

ggsave(plot = age_region,
       filename = "plots/age_region.pdf")

age_region
```

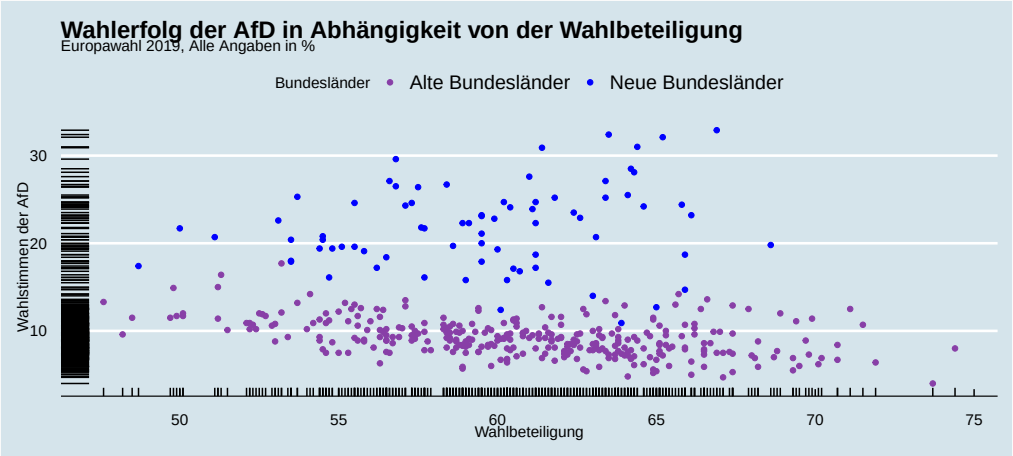


Zusatz: Farbwerte umdefinieren

Farbwerte manuell ändern

Mit `scale_color_manual()` können Farbwerte definiert werden. Dies kann sowohl über vordefinierte Namen als auch über Hex-Werte[2] geschehen:

```
library(ggthemes)
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point(mapping = aes(color = Region)) +
  geom_rug() +
  labs(title = "Wahlerfolg der AfD in Abhängigkeit von der Wahlbeteiligung",
       subtitle = "Europawahl 2019, Alle Angaben in %",
       x = "Wahlbeteiligung", y = "Wahlstimmen der AfD",
       color = "Bundesländer") +
  scale_color_manual(breaks = c("Alte Bundesländer", "Neue Bundesländer"), # Ausprägungen
                     values = c("#883fa7", "blue")) + # Farbwerte
  theme_economist()
```

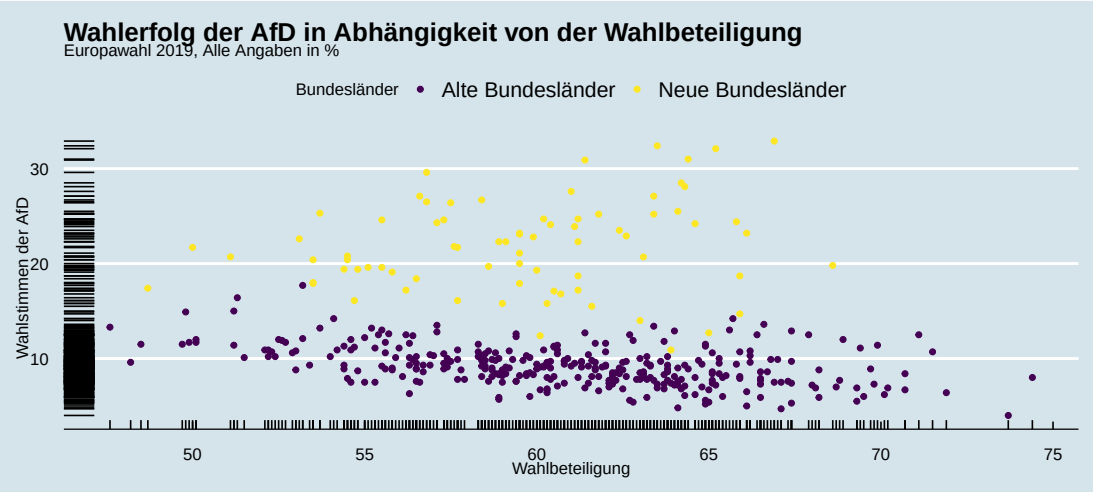


[2] https://www.color-hex.com/

Viridis

Für viele nicht sichtbar: die Standardfarben von GGPlot sind nicht geeignet für Personen mit Farbenblindheit. Hier hilft das Package `viridis`.

```
ggplot(data = df_regio,
       mapping = aes(x = Europawahl_Beteiligung_percent,
                     y = Europawahl_AFD_percent)) +
  geom_point(mapping = aes(color = Region)) +
  geom_rug() +
  labs(title = "Wahlerfolg der AfD in Abhängigkeit von der Wahlbeteiligung",
       subtitle = "Europawahl 2019, Alle Angaben in %",
       x = "Wahlbeteiligung", y = "Wahlstimmen der AfD",
       color = "Bundesländer") +
  scale_color_viridis_d() +
  theme_economist()
```



Letzter Hinweis

Das Erstellen und Anpassen von Grafiken kann durchaus noch viel komplexer werden. In dieser Sitzung gab es daher nur einen kleinen Einblick in die allgemeine Funktionsweise von `ggplot`. Im Zweifel sollte einfach solange auf Webseiten gesucht und rumprobiert werden, bis das gewünschte Ergebnis erzielt wurde.

Bei Inspirationslosigkeit liefert die R Graph Gallery Vorschläge für unterschiedliche Typen von Grafiken.

Viel Spaß beim Ausprobieren! :-)

Welche Fragen habt ihr?