Hands-On Exploratory Data Analytics

Nicolas Meseth

17. Februar 2026

Inhaltsverzeichnis

Vorwort

Dieses Buch führt in die Werkzeuge der explorativen Datenanalyse ein. Konkretes Werzeug, mit dem wir unsere Analysen erstellen, ist R in Kombination mit dem Tidyverse. Für manche Aufgaben im Buch verwenden wir auch Python, insbesondere für die Anwendung von Werkzeugen aus dem Bereich des maschinellen Lernens.

Teil I.

Projekt 1: Umfrage

1. Umfragedaten

In diesem Projekt geht es um die Analyse eines Umfragedatensatzes. Umfragen sind ein geeignetes Mittel für unterschiedliche Forschungsfragen und kommen in der Praxis häufig zum Einsatz, um etwas über Präferenzen von Menschen herauszufinden. Umfrageergebnisse analysieren zu können stellt deshalb eine wichtige Kompetenz dar. Die Werkzeuge, die wir dafür kennenlernen, lassen sich ohne Weiteres auf andere, anders geartete Daten, übertragen.

Lernziele in diesem Abschnitt

- Wie lade ich einen Datensatz mit R?
- Was sind Metainformationen eines Datensatzes?
- Woher weiß ich, wie viele Beobachtungen in meinem Datensatz stecken?
- Wie bekomme ich raus, welche Variablen es gibt?
- Worin unterscheiden sich Variablen?

1.1. Daten laden mit R

In jedem Projekt steht am Anfang das Laden der Daten. In diesem Buch gehen wir davon aus, dass ein Datensatz vorliegt, der bereits erhoben wurde. Die Datenerhebung ist nicht direkt Bestandteil dieses Buches, es geht primär um die Daten*analyse*.

Die Umfrageergebnisse wurden mit der Software Limesurvey erhoben und als CSV-Datei exportiert. Wenn ihr das GitHub-Repository für dieses Buch auf euren Rechner heruntergeladen habt, dann liegen die Daten im Ordner `/data` und können so geladen werden:

```
library(tidyverse)
survey <- read_csv("data/mds12_schoko_milch.csv")
```

Achtet auf den Unterstrich

Bitte achtet auf die Verwendung der richtigen Funktion `read_csv`. Es gibt in R auch eine Funktion mit dem Namen `read.csv`, die sehr ähnlich ist, aber keinen Tibble erzeugt. Wir verwenden in diesem Buch durchgehend das Tidyverse und Tibbles.

Nach Ausführung des Codeblocks stehen die Daten auf dem Objekt mit dem Namen `survey` als Tibble bereit. Es ist eine gute Idee, für Objekte, die wir häufig verwenden, möglichst kurze Namen zu vergeben. Gleichzeitig sollten Namen sprechend sein, damit man anhand des Namens schnell erkennt, worum es sich handelt. Beim Objektnamen `survey` wurde beides berücksichtigt.

Zum Laden der Daten gehört auch eine Prüfung, ob es alles geklappt hat. Insbesodere zu wissen, ob alle Beobachtungen und alle Variablen geladen wurden, ist essenziell, um mit den Daten weiterzuarbeiten. Beovr wir uns damit beschäftigen schauen wir uns an, was man abseits von den Möglichkeiten, mit R etwas über die Daten zu erfahren, sonst noch wissen sollte.

> **Daten laden**
>
> An dieser Stelle gehen wir nicht tiefer auf das Laden von Daten aus verschiedenen Formaten ein. Einen detaillierten Einblick findet ihr in Kapitel 7.

1.2. Metainformationen eines Datensatzes

Der verwendete Datensatz wurde am Fachgebiet Agrarökonomie der Hochschule Osnabrück unter Leitung von Prof. Dr. Ulrich Enneking im Jahr 2025 erhoben. In einer umfangreichen, mehrteiligen Online-Umfrage wurden deutschlandweit Menschen zu Einstellungen und Kaufverhalten bei Lebensmitteln befragt. An der Umfrage haben 2.811 Personen teilgenommen.

Der vorherige Absatz enthält wichtige Informationen über den vorliegenden Datensatz. Wir nennen diese Art von Information auch *Metainformation* (meta = über) oder Kontexinformation. Schauen wir mal drauf, welche Informationen in dem Absatz stecken. Tabelle 1.1 fasst das zusammen:

Tabelle 1.1.: Metainformationen zum Umfragedatensatz

W-Frage	Antwort
Wer hat die Daten erhoben?	Professor Enneking
Wie wurden die Daten erhoben?	Online-Umfrage
Wann wurden die Daten erhoben?	2025
Wo wurden die Daten erhoben?	online
Wer wurde befragt?	2.811 Menschen deutschlandweit
Was wurde erhoben?	Einstellungen und Kaufverhalten bei Lebensmitteln

Ihr seht es handelt sich um eine Reihe von W-Fragen, auf die sich die Kontextinformationen beziehen. Warum sind diese Informationen überhaupt wichtig? Dazu werden wir später noch mehr erfahren, aber hier verdeutlichen wir es uns anhand einer Frage, auf die wir in der obigen

Tabelle 1.1 noch keine Antwort finden, die aber extrem wichtig ist: *Was repräsentiert eine Zeile in den Daten?*

Was sehen wir eigentlich, wenn wir eine Zeile betrachten? Auf diese Frage benötigen wir eine Antwort, bevor wir mit der Datenanalyse beginnen. Zwar können wir oft durch scharfes Hinsehen diese Frage selbst beantworten, aber der Schein kann trügen. Wenn wir nicht selbst Urheber der Daten sind, dann sollten wir diese Information aus sicherer Quelle erfragen. Im vorliegenden Beispiel habe ich das getan und habe meinen geschätzen Kollegn gefragt. Ich weiß nun, dass eine Zeile im Datensatz den Antworten einer Person auf die Online-Umfrage entspricht. Allgemein sprechen wir in der empirischen Forschung von *Beobachtungen*. Eine Beobachtung in unserem Fall ist also eine ausgefüllte Umfrage, die uns in einer Zeile vorliegt. Systematisch zu beobachten und diese Beobachtungen für die Beantwortung von Forschungsfragen zu analysieren ist der Kern der emprischen Forschung.

> Empirical research is any research that uses structured observations from the real world to attempt to answer questions. (Huntington-Klein 2026)

Wenn wir wissen, was eine Zeile bedeutet, dann können wir auch herausfinden, wie viele Personen teilgenommen haben. Oder allgemein, wie viele Beobachtungen wir vorliegen haben. Wie? Mit der Funktion `count`, die einfach die Anzahl der Zeilen (oder Beobachtungen) im Datensatz zählt. Dazu muss man wissen, dass jede Zeile einer Person entspricht, die an der Umfrage teilgenommen hat.

```
survey |>
  count()                                                    ①
```

① Der Befehl `count` zählt die Beobachtungen (oder Zeilen) in einem Datensatz.

```
# A tibble: 1 x 1
      n
  <int>
1  2811
```

Es gibt weitere Möglichkeiten, die Anzahl der Beobachtungen zu ermitteln. Eine ganz einfache ist es, einfach den Namen des Objekts einzugeben, das den Datensatz in R repräsentiert:

Listing 1.1 Wir können einfach den Namen des Tibbles aufrufen, um an wichtige Informationen zu kommen.

```
survey
```

```
# A tibble: 2,811 x 813
   q001hheinkauf q002geburt q003land q004geschlecht q005os v041nofleisch
           <dbl>      <dbl>    <dbl>           <dbl>  <dbl>         <dbl>
 1             2       1970        1               1      0             0
 2             1       1990        7               1      0             0
 3             2       1963        6               1      0             0
 4             2       1989       13               2      0             1
 5             2       1965        4               1      0             1
 6             2       1957        2               1      0             0
 7             2       1960       14               2      0             1
 8             2       1984       13               1      0             0
 9             2       1974        1               2      0             0
10             2       1954       13               2      0             0
# i 2,801 more rows
# i 807 more variables: v041nofleisch_other <chr>, v041diaet_0nodiaet <dbl>,
#   v041diaet_1lowcarb <dbl>, v041diaet_2laktose <dbl>,
#   v041diaet_3gluten <dbl>, v041diaet_4paleo <dbl>, v041diaet_5ketogen <dbl>,
#   v041diaet_6rohkost <dbl>, v041diaet_7makro <dbl>, v041diaet_8trenn <dbl>,
#   v041diaet_9frutarisch <dbl>, v041diaet_10fleisch <dbl>,
#   v041diaet_11mediterran <dbl>, v041diaet_12histaminarm <dbl>, ...
```

Durch die Verwendung des Tidyverse und den dazugehörigen Tibbles bekommen wir diese kompakte und informative Ausgabe. Die erste Zeile der Ausgabe enthält die Dimensionierung der Daten in der Form `Zeilen x Spalten`. Wir bekommen also nicht nur die Information, wie viele Beobachtungen (2.811), sodnern auch, wie viele Spalten (813). Dazu gibt uns ein Tibble gleich noch eine Vorschau seiner ersten zehn Zeilen aus, die so viele Variablen beinhaltet, wie auf die aktuelle Größe der Konsole passen. Die restlichen Variablennamen werden darunter aufgelistet, zumindest ein Teil davon. Ein Tibble achtet darauf, die Konsole nicht komplett mit Text zu fluten und schneidet daher irgendwann ab.

Es gibt dedizierte Werkzeuge, um mehr über die Variablen herauszufinden. Die schauen wir uns jetzt an.

1.3. Variablen (Spalten)

1.3.1. Was ist eine Variable?

> A variable, in the context of empirical research, is a bunch of observations of the same measurement. (Huntington-Klein 2026)

Neben der Frage, was eine Zeile darstellt, sollten wir auch wissen, was jede Variable genau misst. Während wir Beobachtungen typischerweise horizontal, also in Zeilen, darstellen, sind Variablen das, was wir vertikal, also in Spalten, darstellen.

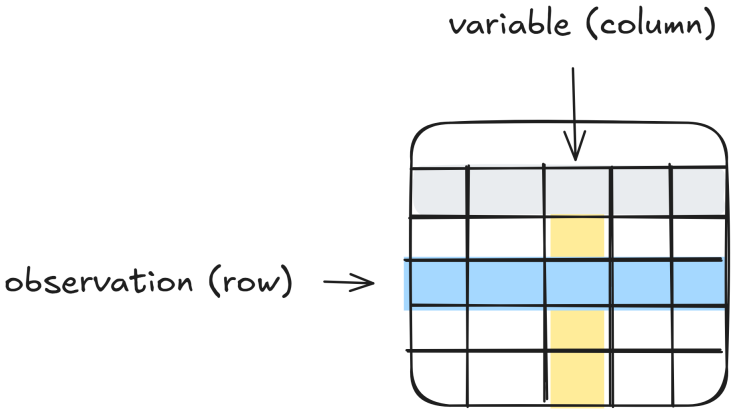


Abbildung 1.1.: Beobachtungen repräsentieren wir als Zeilen, Variablen als Spalten.

Auch die Informationen über die gemessenen Variablen gehören zu den Metainformationen eines Datensatzes. Und im besten Fall gibt es ein so genanntes *Data Dictionary*, eine Art Wörterbuch, in dem jede Variable aufgeführt und inhaltlich erläutert ist. Leider ist das nicht immer der Fall, in diesem Beispiel jedoch schon, wie wir später sehen werden.

Unabhängig davon, ob es ein Data Dictionary gibt, wollen wir uns im Folgenden anschauen, wie wir mit R an wichtige Informationen zu den Variablen kommen. Folgende Fragen solltet ihr euch zu Beginn stellen:

- Wie viele Variablen gibt es?
- Welche Arten von Variablen gibt es?
- Welchen Wertebereiche hat eine Variable?

1.3.2. Wie viele Variablen?

Wie viele Variablen der Datensatz hat wissen wir bereits aus Listing 1.1. Die Anzahl der Variablen ist im vorliegenden Datensatz mit **813** verhältnismäßig hoch. Das ist typisch für Umfragen, insbesondere wenn viele Fragen enthalten sind. Denn jede Antwortoption und deren Ausprägung wird im Ergebnis mit einer eigenen Variable abgebildet, da kommt schnell was zusammen.

Übrigens können wir mit folgendem Befehl die Anzahl Variablen (oder Spalten) ganz konkret als Wert auslesen, und nicht nur auf der Konsole ausgeben:

```r
survey |>
  ncol()
```

```
[1] 813
```

Diesen Wert können wir auf einem neuen Objekt speichern und später in unserem R-Skript wieder verwenden, was häufig nützlich ist:

```r
variable_count <- ncol(survey)
print(str_glue("Es haben {variable_count} Menschen teilgenommen."))          ①
```

① Mit `str_glue` können wir Zeichenketten mit Platzhaltern versehen, die zur Ausführung mit den Werten ersetzt werden.

```
Es haben 813 Menschen teilgenommen.
```

1.3.3. Welche Arten von Variablen gibt es?

Schauen wir uns fürs erste nur die ersten 10 Variablen im Datensatz genauer an:

```r
survey |>
    select(1:10) |>                                                          ①
    glimpse()                                                                ②
```

① Mit `select` können wir Variablen auswählen. Die Notation `1:10` bedeutet so viel wie: *Variablen an Stelle 1 bis 10.*
② Mit `glimpse` bekommen wir schnell eine Übersicht der Namen, Datentypen und ersten Werte der Variablen.

```
Rows: 2,811
Columns: 10
$ q001hheinkauf       <dbl> 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1,~
$ q002geburt          <dbl> 1970, 1990, 1963, 1989, 1965, 1957, 1960, 1984, 19~
$ q003land            <dbl> 1, 7, 6, 13, 4, 2, 14, 13, 1, 13, 5, 14, 9, 4, 3, ~
$ q004geschlecht      <dbl> 1, 1, 1, 2, 1, 1, 2, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1,~
$ q005os              <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ v041nofleisch       <dbl> 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,~
$ v041nofleisch_other <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
$ v041diaet_0nodiaet  <dbl> 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1,~
$ v041diaet_1lowcarb  <dbl> 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ v041diaet_2laktose  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,~
```

Im Codeblock oben lernen wir mit `select` und `glimpse` gleich 2 zwei wichtige Funktionen kennen. Beide werden wir in diesem Buch noch sehr häufig verwenden. Mit `select` können wir aus den vielen Spalten genau die auswählen, an denen wir momentan interessiert sind. Das kann etwa für die Auswertung einer bestimmten Frage sinnvoll sein, bei der wir nur die Variablen benötigen, die sich auf die betroffene Frage beziehen. Die Notation `1:10` in den Klammern der `select`-Funktion bedeutet übersetzt *wähle die Variablen 1 bis 10 aus*. Dazu muss man wissen, dass Variablen in einem Datensatz eine feste Reihenfolge haben. Wir bekommen damit also die ersten 10 Variablen im Datensatz.

Mit der Flexibilität einer Programmiersprache wie R können wir jede beliebige Menge an Spalten auswählen. Zum Beispiel auch die letzten 10:

```
survey |>
  select(tail(everything(), 10))
```

```
# A tibble: 2,811 x 10
   M070handeldiff3_1hochwertig M070handeldiff3_2regio M070handeldiff3_3preis
                         <dbl>                  <dbl>                   <dbl>
 1                          NA                     NA                      NA
 2                          -1                      1                       2
 3                          NA                     NA                      NA
 4                          NA                     NA                      NA
 5                          NA                     NA                      NA
 6                          NA                     NA                      NA
 7                          NA                     NA                      NA
 8                          NA                     NA                      NA
 9                           2                      2                      -2
10                          NA                     NA                      NA
# i 2,801 more rows
# i 7 more variables: M070handeldiff3_4tierwohl <dbl>,
#   M070handeldiff3_5vielfalt <dbl>, M070handeldiff3_6vetrauen <dbl>,
#   M070handeldiff3_7nachhaltig <dbl>, M070handeldiff3_8gesund <dbl>,
#   M070handeldiff3_9kauf <dbl>, M070handeldiff3_10trends <dbl>
```

Mit `tail` ermitteln wir die letzen `n` Elemente einer Liste. `everything` gibt uns die Liste aller Spalten zurück, wenn es innerhalb der `select`-Funktion aufgerufen wird. Somit kann man den Befehl übersetzen mit *Gib mir die letzten 10 Elemente (`tail`) aus der Liste aller Spalten (`everything`)*. Wir werden später noch Werkzeuge im Detail lernen, wie wir geanu die Spalten finden und auswählen, die wir brauchen. Das ist gerade bei Umfragen mit vielen Spalten extrem nützlich.

> **Variablen haben eine feste Position**
>
> Variablen (oder Spalten) haben eine feste Reihenfolge und Position in einem Datensatz. Wir können also jede Spalte statt über ihren Namen auch über ihre Position ansprechen.

Die Funktion `glimpse` wird unmittelbar nach dem `select` aufgerufen und bekommt das Ergebnis übergeben. Dafür sorgt das etwas merkwürdig aussehende Symbol `|>`, das wir den Pipe-Operator oder einfach nur die Pipe nennen. Es sorgt dafür, dass das Ergebnis des ersten Befehls (vor der Pipe) an den nächsten Befehl (nach der Pipe) übergeben wird. Auf diese Weise können wir Ketten von Befehlen aufbauen, in der jeder Befehl sein eigenes Ergebnis an den nächsten übergibt. Eine dedizierte Einfürhung in den Pipe-Operator findet ihr in Kapitel 8.

1.3.4. Welchen Wertebereich hat eine Variable?

Zurück zur Ausgabe vom `glimpse` oben. Was genau bedeutet das, was wir auf der Konsole sehen? Die ersten beiden Zeilen geben Auskunft über die Anzahl an Zeilen und Spalten im Datensatz. Warum sind es hier nur 10 Zeilen, statt der 813 Spalten, wie oben geschrieben? Weil wir `glimpse` nicht mehr für den ursprünglichen Datensatz aufrufen, sondern für den, der als Ergebnis des vorgelagerten `select(1:10)` entsteht. Und der hat nur noch 10 Spalten, was wir aber genau so wollten.

Darunter folgen Zeilenweise die Spalten mit den Informationen Spaltenname, Datentyp und Auflistung der ersten Werte für jede Spalte. Was genau die Datentypen wie `<dbl>` oder `<chr>` bedeuten lernen wir später noch. Nach dem Datentyp sehen wir eine komma-getrennte Liste der ersten Werte, soweit sie auf die Konsole passen. Für die erste Spalte im Datensatz, `q001hheinkauf`, sind die ersten Werte also 2, 1, 2, 2 usw. Was das genau bedeutet können wir nicht wissen, sondern müssen es nachschlagen. Dazu gibt es üblicherweise zu einem Fragebogen ein entsprechendes Codebuch, in dem die Antwortkodierungen in sprechende Werte übersetzt werden. Das ist meistens Teil des Data Dictionary.

Bei der ersten Frage, zu der die Spalte `q001hheinkauf` gehört, geht es um die Frage, ob die teilnehmende Person für den Lebensmitteleinkauf hauptverantwortlich ist oder ob die Aufgabe mit einer anderen Person geteilt wird. Der Wert 1 steht hier kurz gesagt für "allein verantwortlich" und 2 für "geteilte Verantwortung". Aber sind das die einzigen Werte, die die Variable annimmt? Das können wir aus der Ausgabe der `glimpse`-Funktion nicht ableiten.

Wir könnten die Spalte einfach auswählen und uns das Ergebnis ausgeben lassen:

```
survey |>
  select(q001hheinkauf)
```

```
# A tibble: 2,811 x 1
   q001hheinkauf
```

```
           <dbl>
 1             2
 2             1
 3             2
 4             2
 5             2
 6             2
 7             2
 8             2
 9             2
10             2
# i 2,801 more rows
```

Das Ergebnis ist ein Tibble, der auf der Konsole standardmäßig seine ersten 10 Werte ausgibt. Dieses Verhalten können wir ändern und den Tibble um mehr Werte bitten:

```
survey |>
  select(q001hheinkauf) |>
  print(n = 20)
```

```
# A tibble: 2,811 x 1
   q001hheinkauf
           <dbl>
 1             2
 2             1
 3             2
 4             2
 5             2
 6             2
 7             2
 8             2
 9             2
10             2
11             2
12             1
13             2
14             1
15             2
16             2
17             1
18             1
19             2
```

```
20             1
# i 2,791 more rows
```

Damit sind des schon doppelt so viele Beobachtungen für die Variable `q001hheinkauf`, und immernoch sehen wir nur die Ausprägungen 1 und 2. Um wirklich sicher zu wissen, welche Werteausprägungen existieren, geben wir die eindeutigen Werte aus:

```
survey |>
  distinct(q001hheinkauf)
```

```
# A tibble: 2 x 1
  q001hheinkauf
          <dbl>
1             2
2             1
```

Und siehe da: Es ist tatsächlich so, dass nur diese beiden Werte vorkommen. Wie oft? Auch das bekommen wir schnell raus:

```
survey |>
  count(q001hheinkauf)
```

```
# A tibble: 2 x 2
  q001hheinkauf     n
          <dbl> <int>
1             1   919
2             2  1892
```

Die Funktion `count` kennen wir bereits von oben. Wenn wir ihr eine Variable übergeben, dann ermittelt sie, wie häufig jede Ausprägung der Variable in den Daten vorkommt. Technisch gruppiert sie die Daten nach der übergebenen Variable und zählt die Beobachtungen für jede Gruppe.

> **Aggregieren von Daten**
>
> Die Funktion `count` zählt zu den Aggregationsfunktionen. Eine dedizierte Einführung, wie man mit R und dem `dplyr`-Paket Daten zusammenfasst, findet ihr in Kapitel 10

1.4. Beobachtungen (Zeilen)

Was für Variablen gilt, gilt auch für Zeilen. Jede hat eine feste Position. Bei den Zeilen nennen wir das auch die *row number* (Zeilennummer), und es gibt eine Funktion, um diese zu ermitteln:

```
survey |>
  select(1:10) |>
  filter(row_number() <= 10)                                    ①
```

① Mit `row_number` bekommen wir die Position (oder Nummer) für eine Zeile.

```
# A tibble: 10 x 10
   q001hheinkauf q002geburt q003land q004geschlecht q005os v041nofleisch
           <dbl>      <dbl>    <dbl>          <dbl>  <dbl>         <dbl>
 1             2       1970        1              1      0             0
 2             1       1990        7              1      0             0
 3             2       1963        6              1      0             0
 4             2       1989       13              2      0             1
 5             2       1965        4              1      0             1
 6             2       1957        2              1      0             0
 7             2       1960       14              2      0             1
 8             2       1984       13              1      0             0
 9             2       1974        1              2      0             0
10             2       1954       13              2      0             0
# i 4 more variables: v041nofleisch_other <chr>, v041diaet_0nodiaet <dbl>,
#   v041diaet_1lowcarb <dbl>, v041diaet_2laktose <dbl>
```

Die Funktion `filter` verwenden wir, um die Beobachtungen nach bestimmten Kriterien einzugrenzen. Zum Beispiel um nur weibliche Personen zu betrachten. Wir können mit der `row_number`-Funktion die Beobachtungen auch nach ihrer Position im Datensatz eingrenzen. Das obige Beispiel behält nur jene Zeilen im Ergebnis, deren Position kleiner oder gleich (`<=`) 10 ist.

Statt kleiner-gleich können wir auch andere Operatoren verwenden:

```
survey |>
  filter(row_number() == 42) |>
  select(q004geschlecht)
```

```
# A tibble: 1 x 1
  q004geschlecht
           <dbl>
1              2
```

2. Arten von Variablen

Wie haben in Kapitel 1 schon Beispiele für eine Arten von Variablen gesehen. In Umfragen begegnen wir häufig nominalskalierten Variablen, seltener sind numerische oder gar qualitative Variablen. Aber was genau unterscheidet diese Variablentypen?

2.1. Ein Schema zur Orientierung

Um das besser zu verstehen wollen wir katgeoriale Variablen gegenüber anderen Arten von Variablen abgrenzen. Laut Huntington-Klein (2026) unterscheiden wir in der empirischen Forschung zwischen den folgenden Typen von Variablen:

- Kontinuierliche Variablen
- Zählvariablen
- Ordinale Variablen
- Kategoriale Variablen
- Qualitative Variablen

Ich finde diese Einteilung trifft es sehr gut und verwende sie daher in diesem Buch weiter. Man kann noch eine Ebene darüber einziehen und kontinuierliche Variablen und Zählvariablen als numerische Variablen zusammenfassen. Parallel dazu sind ordinale Variablen genau genommen eine Form von kategorialen Variablen und eine weitere wären die nominalskalierten Variablen.

Schauen wir uns für jeden Typ Beispiele an, um ihn besser zu verstehen.

2.2. Numerische Variablen

Bei den numerischen Variablen könnte man auch von quantitativen Variablen sprechen. Wir unterscheiden dabei solche, die prinzipiell jeden beliebigen Wert als reelle Zahl annehmen können und solche, die nur ganze, also natürliche, Zahlen annehmen können. Erstere nennen wir *kontinuierlich*, weil jeder Wert auf einem Kontinuum möglich ist.

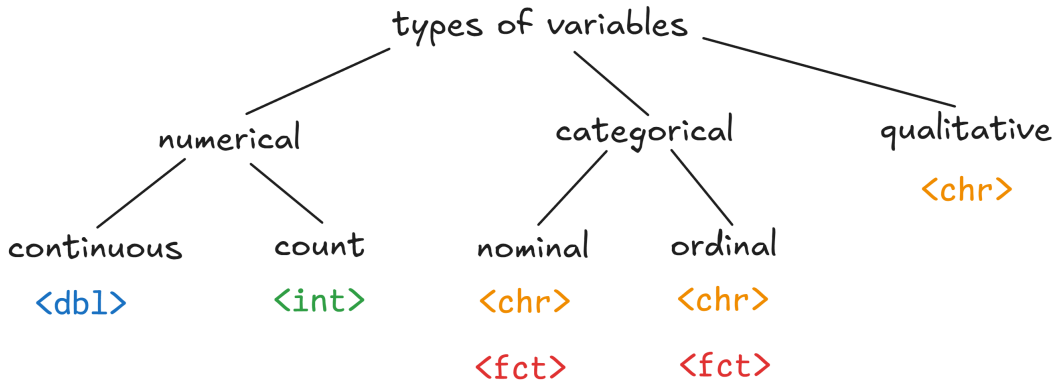


Abbildung 2.1.: Arten von Variablen und typische verwendete Datentypen in R.

2.2.1. Kontinuierliche Variablen

> Continuous variables are variables that could take any value (perhaps within some range). (Huntington-Klein 2026)

Im vorliegenden Umfragedatensatz fragt die Frage 13 nach der Eingabe eines Preises, den die Teilnehmenden bereit wären, für einen Liter Milch vorgegebener Marken auszugeben:

"Geben Sie für die 3 Marken bitte den Preis an, den Sie für angemessen halten und bei dem Sie zugreifen würden! Als Dezimaltrennzeichen verwenden Sie bitte das Komma."

Schauen wir auf ein paar Beobachtungen zu dieser Frage. Wie sehen die Antworten aus?

```
survey |>
    select(u013pzahl_1weihen)
```

```
# A tibble: 2,811 x 1
   u013pzahl_1weihen
               <dbl>
 1              1.49
 2             NA
 3             NA
 4             NA
 5             NA
 6             NA
 7             NA
```

```
 8             NA
 9            150
10             NA
# i 2,801 more rows
```

Anhand der Ausgabe sehen wir 3 Dinge:

1. Der Datentyp der Spalte ist als `<dbl>` angegeben. Das ist typisch für kontinuierliche Varialen, denn das steht für den Datentyp *double*, der in der Programmierung für Dezimalzahlen (mit Nachkommastellen) verwendet wird.
2. Viele Werte sind mit `NA` angegeben, was bedeutet, dass hier kein Wert existiert. In der Auswertung der Daten stellt sich die Frage, was das im Kontext des Datensatzes bedeutet und wie wir damit umgehen.
3. Die beiden einzigen Werte sind 1,49 € und 150 €. Letzteres dürfte ein Tippfehler sein und wirft auch hier die Frage auf, wie wir damit umgehen.

Den zweiten Punkt in der Liste könnten wir einfach abhaken, indem wir die `NA`-Werte herausfiltern (auch wenn das kein Patentrezept ist). Dafür gibt es die `drop_na`-Funktion:

```
survey |>
    select(u013pzahl_1weihen) |>
    drop_na()
```

```
# A tibble: 231 x 1
   u013pzahl_1weihen
               <dbl>
 1              1.49
 2            150
 3              1.19
 4              3
 5              2
 6              1.6
 7              1.09
 8              3
 9              1.8
10              2.2
# i 221 more rows
```

Die Ausgabe enthält jetzt keine fehlenden Werte mehr und wir bekommen ein paar mehr "echte" Werte angezeigt. Wir zu erwarten gibt es fast beliebige Werte mit zwei Nachkommastellen, die als Antwort auf die Frage zulässig sind. Solche Daten sind typisch für kontinuierliche Variablen; es gibt keine feste Liste an Werten, sondern theoretisch existiert jeder Wert auf

einem Kontinuum. Das kann man sich verdeutlichen, wenn man versucht, die eindeutigen Werte zu ermitteln. Dafür haben wir in Kapitel 1 die Funktion `distinct` kennengelernt:

```
survey |>
    select(u013pzahl_1weihen) |>
    drop_na() |>
    distinct()
```

```
# A tibble: 48 x 1
   u013pzahl_1weihen
               <dbl>
 1              1.49
 2            150
 3              1.19
 4              3
 5              2
 6              1.6
 7              1.09
 8              1.8
 9              2.2
10            120
# i 38 more rows
```

Zwar sieht es hier so aus, als wären die eindeutigen Werte mit 48 nicht so viele, als das sie nicht auch eine feste Menge an Kategorien darstellen könnten. Allerdings ist es klar, dass *theoretisch* viel mehr unterschiedliche Preise hätten eingegeben werden können und dass es sich bei der Abfrage eines Preises von Natur her um eine numerische Größe handelt.

2.2.1.1. Ausreißer

Bei kontinuierlichen Variablen spielt häufig das Problem der Ausreißer eine Rolle. Es gibt verschiedene Gründe für Ausreißer, bei Umfragen kommt etwa ein einfacher Tippfehler als Ursache infrage. Je nach Ursprung sind unterschiedliche Maßnahmen möglich. Tippfehler sollten wir entweder korrigieren, oder, wenn wir das nicht können, herausfiltern. Veranschaulichen wir uns die Problematik kurz an einem Beispiel und berechnen den Durchschnittspreis, den die Probanden eingegeben haben:

```
survey |>
    summarize(avg_price = mean(u013pzahl_1weihen))
```

```
# A tibble: 1 x 1
  avg_price
      <dbl>
1        NA
```

Die `mean`-Funktion berechnet das arithemtische Mittel und mit `summarize` erzeugen wir eine neue, aggregierte Spalte. Die neue Spalte `avg_price` sollte also den Durchschnittspreis (als arithmetisches Mittel) über alle Probanden enthalten. Wie wir sehen, ist der Wert aber `NA`.

Der Grund dafür ist einfach: R weiß nicht, wie es `NA`-Werte in einer Berechnung behandeln soll. Daher wird jede Berechnung, in der ein `NA` vorkommt, im Ergebnis sofort auch `NA`. Die Lösung: Die `NA`-Werte müssen weg:

```
survey |>
    drop_na(u013pzahl_1weihen) |>                                    (1)
    summarize(avg_price = mean(u013pzahl_1weihen))
```

(1) Die `drop_na`-Funktion entfernt alle Zeilen mit `NA`-Werten. Wenn eine oder mehrere Spalten übergeben werden, dann bezieht sich die `NA`-Filterung nur auf diese. Sobald eine in der Liste den Wert `NA` aufweist, wird die Zeile entfernt.

```
# A tibble: 1 x 1
  avg_price
      <dbl>
1      4.58
```

Und schon funktioniert es! Was sagt uns der Wert? Im Durchschnitt halten die Probanden also einen Preis von 4,58 € für einen Liter Milch für gerechtfertigt. Kann das sein? Haben die alle zu viel Geld? Berechnen wir den Durchschnitt mit einer anderen Kenngröße, dem Median:

```
survey |>
    drop_na(u013pzahl_1weihen) |>
    summarize(avg_price = median(u013pzahl_1weihen))
```

```
# A tibble: 1 x 1
  avg_price
      <dbl>
1      1.49
```

Oha! Der Median ist mit 1,49 € deutlich kleiner. Wie kommt es dazu? Die Antwort ist einfach: Der Median ist robust gegenüber Ausreißern, das arithmetische Mittel dagegen nicht. Das arithmetische Mittel ist einfach die Summe aller Werte geteilt durch die Anzahl Werte:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Der Median ist dagegen einfach der mittlere Wert der sortierten Preise; bei geradem n ist er der Mittelwert der beiden mittleren Werte. Und ihm ist es egal, ob am Ende der Liste ein paar riesige Werte folgen.

$$\tilde{x} = \begin{cases} x_{\left(\frac{n+1}{2}\right)}, & \text{falls } n \text{ ungerade ist,} \\ \frac{1}{2}\left(x_{\left(\frac{n}{2}\right)} + x_{\left(\frac{n}{2}+1\right)}\right), & \text{falls } n \text{ gerade ist.} \end{cases}$$

Was also tun? Da wir nicht wissen können, was die Probanden eigentlich eingeben wollten, filtern wir die unangemessen hohen Werte heraus. Schauen wir zunächst einmal, welche das überhaupt sind:

```
survey |>
    select(u013pzahl_1weihen) |>
    arrange(desc(u013pzahl_1weihen))
```

```
# A tibble: 2,811 x 1
   u013pzahl_1weihen
               <dbl>
 1               150
 2               150
 3               129
 4               120
 5               109
 6                50
 7                 7
 8                 4
 9                 4
10                 3
# i 2,801 more rows
```

Was meint ihr? Gar nicht so einfach, zu entscheiden, welche Werte ernst gemeint sind und welche auf Tippfehler schließen lassen. Pragmatisch entscheiden wir uns für die 10 € als Grenze:

```
survey |>
    filter(u013pzahl_1weihen < 10.0) |>
    summarize(avg_price = mean(u013pzahl_1weihen))
```

```
# A tibble: 1 x 1
  avg_price
      <dbl>
1      1.56
```

Durch das Herausfiltern der Ausreißer nähert sich das arithmetische Mittel dem Median deutlich an. Jetzt haben wir die Ausreißer nach oben, also die extrem hohen Werte, betrachtet und behandelt. Gibt es auch Ausreißer nach unten?

```
survey |>
    select(u013pzahl_1weihen) |>
    arrange(u013pzahl_1weihen)
```

```
# A tibble: 2,811 x 1
   u013pzahl_1weihen
               <dbl>
 1              0
 2              0
 3              0
 4              0.09
 5              0.89
 6              0.95
 7              0.99
 8              0.99
 9              0.99
10              0.99
# i 2,801 more rows
```

Die `desc`-Funktion hat zuvor dafür gesorgt, dass die Werte absteigend sortiert werden (*descending* = absteigend). Lassen wir sie weg, dann wird standardmäßig aufsteigend sortiert. Die kleinsten Werte sind jetzt oben in der Liste und wir können erkennen, dass auch hier offensichtlich unplausible Werte existieren. Ein Preis von Null Euro oder nur 9 Cent? Auch die sollten wir vielleicht herausfiltern. Biedes zusammen sieht dann so aus:

```
survey |>
    filter(u013pzahl_1weihen < 10.0) |>
    filter(u013pzahl_1weihen > 0.1) |>
    summarize(avg_price = mean(u013pzahl_1weihen))
```

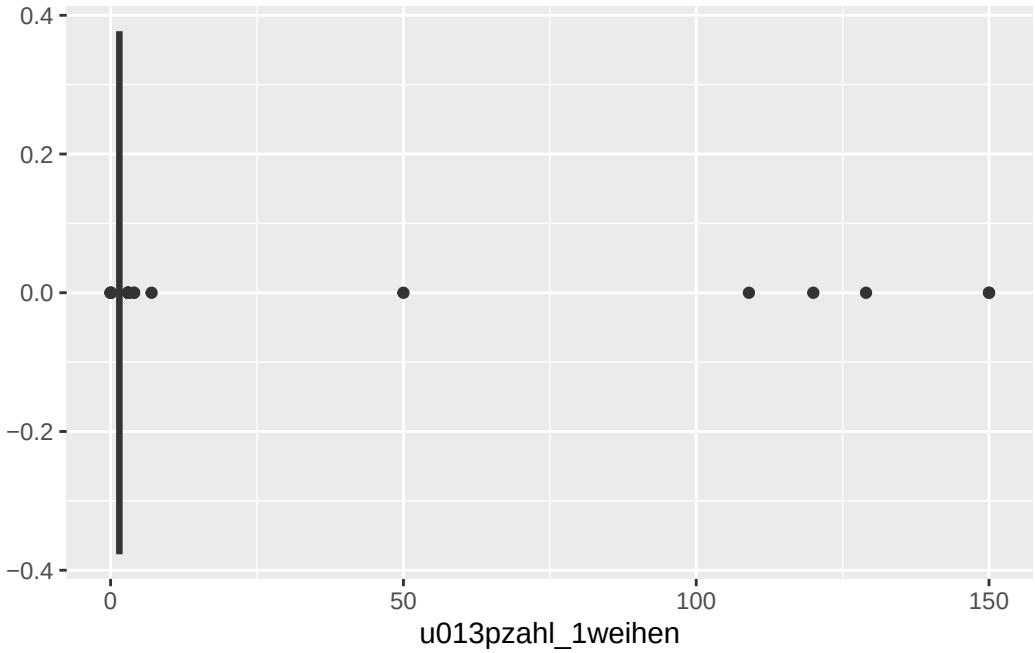
```
# A tibble: 1 x 1
  avg_price
      <dbl>
1      1.59
```

Durch den Aufbau einer Pipeline von `filter`-Anweisungen werden diese nacheinander angewendet. Zuerst fliegen alle Preise raus, die über 10 € liegen, danach alle, die kleiner als 10 Cent sind. Am Ende berechnen wir auf Basis der gefilterten Werte erneut das arithmetische Mittel.

> **Filtern von Daten**
>
> Die Funktion `filter` ist zentral für die Auswahl der richtigen Beobachtungen. Eine Einführung, wie man mit R und dem `dplyr`-Paket Daten filtert, findet ihr in Kapitel 9

Übrigens stellt die Visualisierung eine geeignete Methode dar, um Ausreißer schnell zu erkennen. Ein Boxplot etwa stellt Ausreißer als Punkte außerhalb der Box dar:

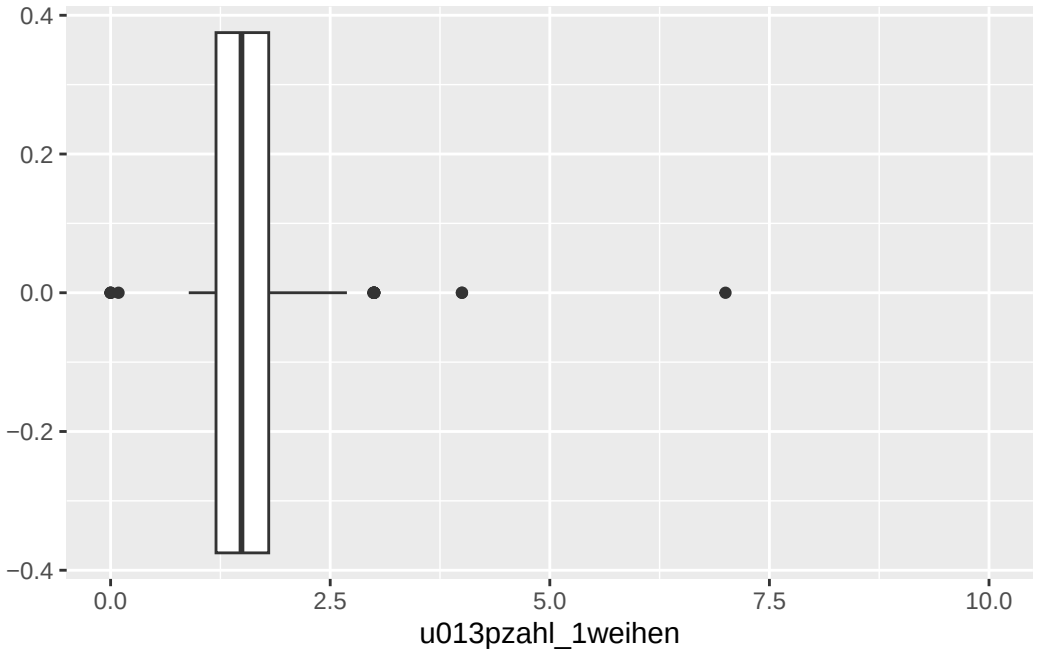
```
survey |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_boxplot()
```



Man erkennt hier sehr schön die handvoll Ausreißer ab der 50€-Marke. Die Box des Plots ist durch die Ausreißer nur als dicker Strich zu erkennen. Was tun? Ein einfaches Filtern ist im Fall eines Boxplots nicht zulässig, nur um den Plot besser zu sehen. Das liegt daran, dass der Boxplot verschiedene Werte berechnet und als Größen der Box darstellt. Darunter auch der Median. Filtern wir aber Werte heraus und berechnen dann den Median, dann ändert sich dadurch die Grundlage für die Berechnung, und damit auch der Plot selbst. Was also tun, damit wir dennoch eine sinnvolle Visualisierung bekommen?

Die Anwort ist: Reinzoomem statt filtern! Reinzoomen bedeutet, wir betrachten nur den Teil der x-Achse, in dem der wichtigste Teil des Plots liegt. Versuchen wir es und legen einen Bereich zwischen 0 und 10 € fest:

```
survey |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_boxplot() +
    coord_cartesian(xlim = c(0, 10))
```



Schon besser, die Box wird sichtbar. Allerdings ist rechts daneben noch immer viel Platz, den wir in der Visualisierung für die Darstellung der Box verwenden könnten. Wenn man sich vergegegwärtigt, dass die Begrenzungen der Box den 25% bzw. 75%-Punkt der Werte bedeuten, und er Strich in der Mitte der 50%-Punkt (Median), dann liegt die Hälfte der Werte in der Box selbst. Ein Viertel liegt jeweils links und rechts von der Box. Die sogenannten Whisker, also die Linien die aus der Box herauslaufen, stellen Werte dar, die noch 1,5-fachen Interquartilsabstand zu den Punkten Q_1 und Q_3 liegen:

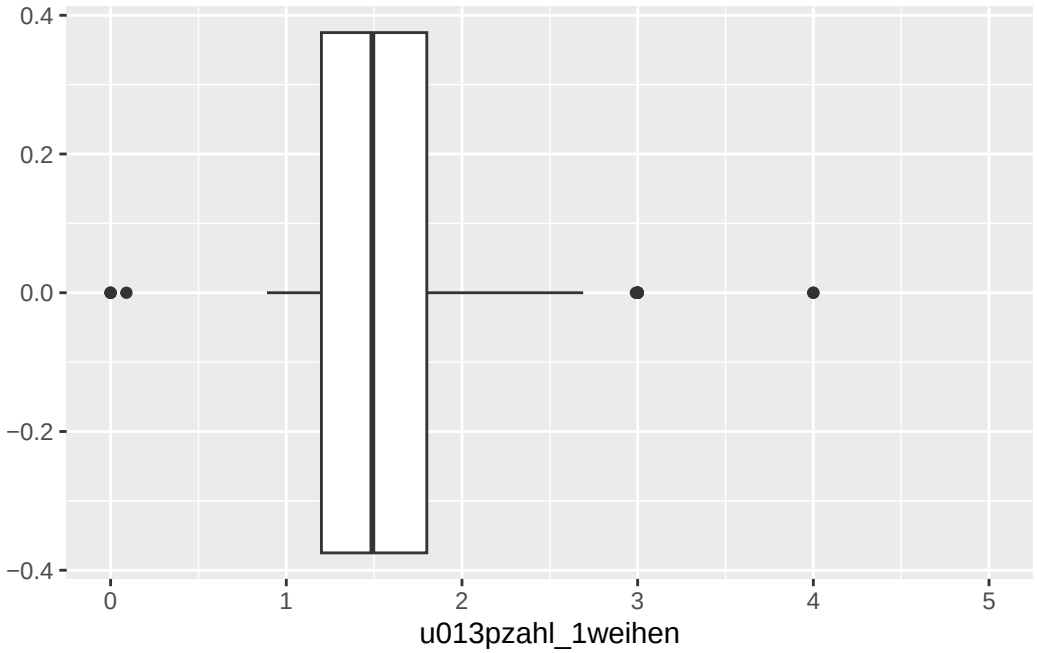
$$[\,Q_1 - 1.5 \cdot \text{IQR},\ Q_3 + 1.5 \cdot \text{IQR}\,]$$

Wobei gilt:

$$\text{IQR} = Q_3 - Q_1$$

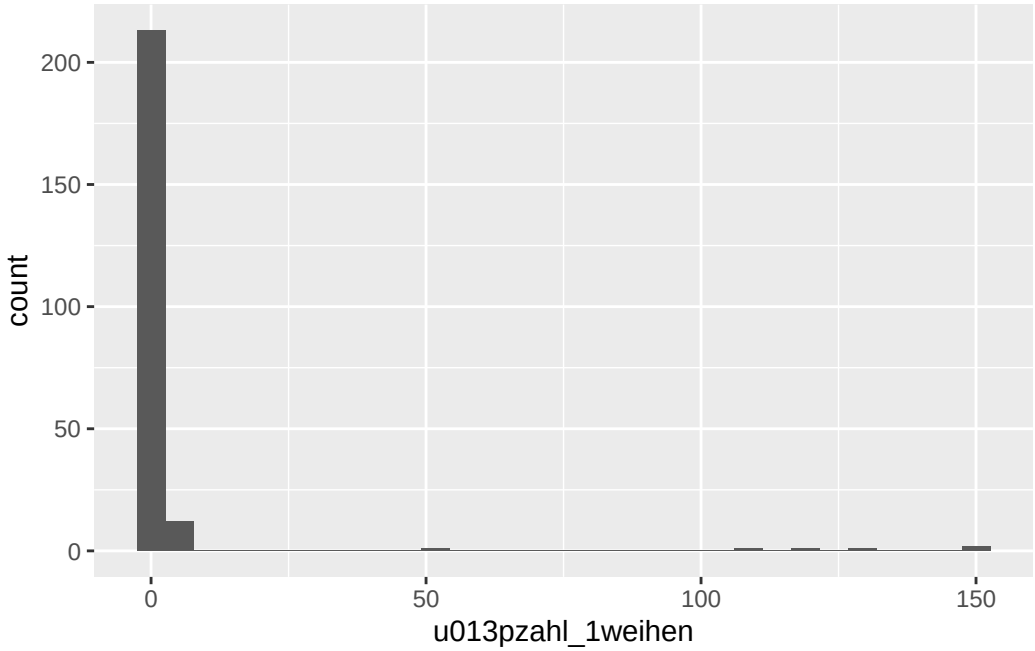
Zoomen wir also noch ein Stück weiter hinein:

```
survey |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_boxplot() +
    coord_cartesian(xlim = c(0, 5))
```



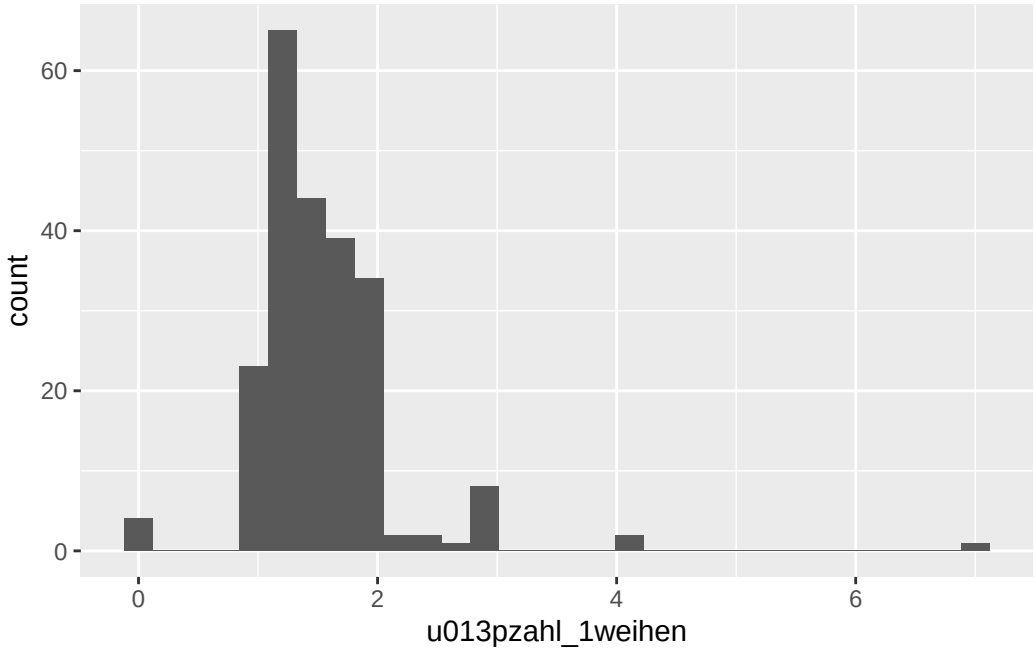
Eine Alternative für die Darstellung der Verteilung einer kontinuierlichen Variable ist das Histogram:

```
survey |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_histogram()
```



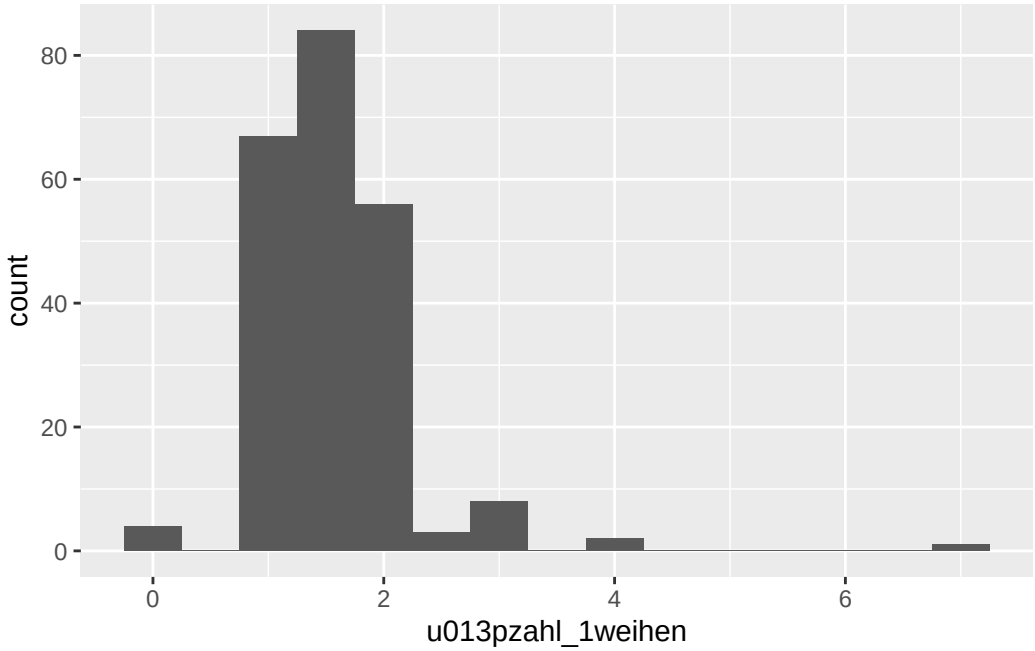
Auch hier leiden wir erneut unter den Ausreißern. Anders als beim Boxplot können wir hier aber einfach filtern, schließlich berechnet das Histogramm lediglich die Häufigkeiten pro Klasse, die wir dadurch nicht verändern:

```
survey |>
    filter(u013pzahl_1weihen < 10.0) |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_histogram()
```



Wenn wir nichts angeben, dann wählt `ggplot2` für uns eine passende Klassengröße basierend auf den Daten. Diese können wir beeinflussen, vielleicht wollen wir Klassengrößen von 50 Cent betrachten:

```
survey |>
    filter(u013pzahl_1weihen < 10.0) |>
    ggplot() +
    aes(x = u013pzahl_1weihen) +
    geom_histogram(binwidth = 0.5)
```



Wir kommen im Verlauf des Projekts auf unterschiedliche kontinuierliche Variablen zurück. Für jetzt merken wir uns, dass bei ihen Vorsicht mit Ausreißern geboten ist und dass wir mit Boxplots und Histogrammen ihre Verteilung schnell überblicken können.

2.2.2. Zählvariablen

> Count variables are those that, well, count something. Perhaps how many times something happened or how many of something there are. (Huntington-Klein 2026)

Kurz gesagt sind der Unterschied zwischen kontinuierlichen Variablen und Zählvariablen die Stellen nach dem Komma. Letztere haben keine. Und das ist typisch für Dinge, die einfach abzählbar sind:

- Die Jahre, die ein Mensch schon lebt (Alter)
- Die Häufigkeit, wie oft man im Jahr im Fitnessstudio war
- Die Anzahl Produkte, die verkauft wurden

Zählvariablen können sehr viele unterschiedliche Ausprägungen haben und unterscheiden sich dann oft nicht großartig von kontinuierlichen Variablen. Zumindest nicht in der Auswertung. Dennoch gibt es wichtig Unterschiede. Betrachten wir ein Beispiel aus dem Umfragedaten: Das Alter der Probanden.

Das Alter ist keine Variable, die direkt in der Umfrage gemessen wurde. Es steckt aber indirekt in der Antwort auf die Frage 2: *"In welchem Jahr sind Sie geboren?"* Wenn wir das Geburtsjahr kennen, dann können wir das Alter mit einer kleinen Restunsicherheit schätzen.

```
survey |>
  select(q002geburt)
```

```
# A tibble: 2,811 x 1
   q002geburt
        <dbl>
 1       1970
 2       1990
 3       1963
 4       1989
 5       1965
 6       1957
 7       1960
 8       1984
 9       1974
10       1954
# i 2,801 more rows
```

Das Geburtsjahr ist numerisch (`<dbl>`), wir müssen es nur noch von der Jahreszahl abziehen, in der die Umfrage stattgefunden hat:

```
survey |>
  transmute(age = 2025 - q002geburt)
```

```
# A tibble: 2,811 x 1
     age
   <dbl>
 1    55
 2    35
 3    62
 4    36
 5    60
 6    68
 7    65
 8    41
 9    51
10    71
# i 2,801 more rows
```

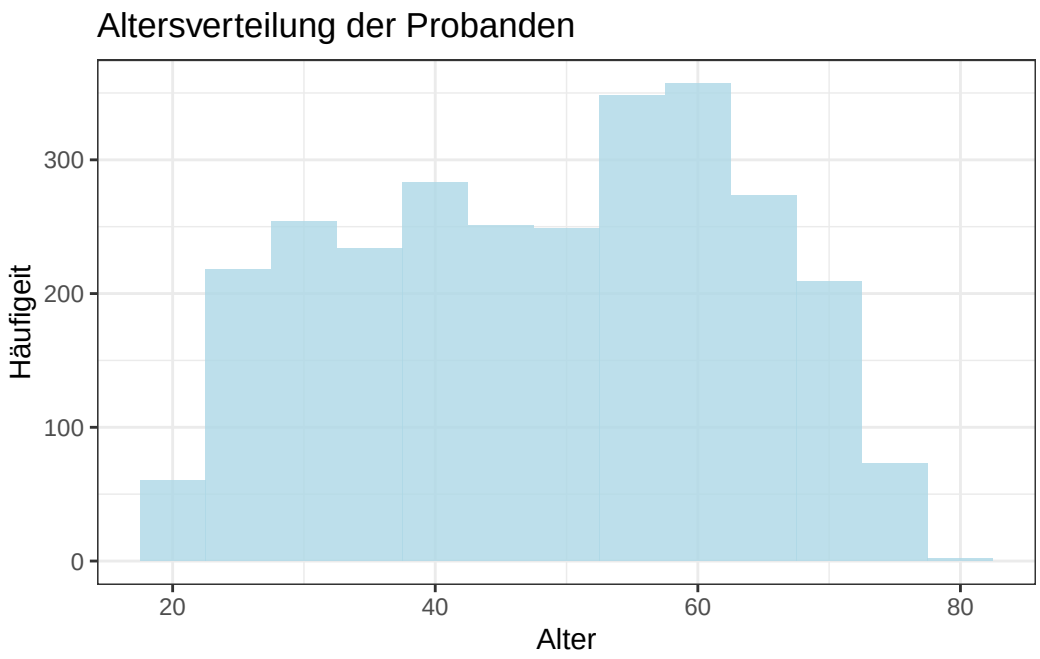
Sehr schön, wir haben unsere Zählvariable `age`! Wie verschaffen wir uns einen Überblick? Wir könnten für die Variable `age` die selben Analysen durchführen, wie wir sie oben für die kontinuierliche Variable eines Preises angewendet haben. Schließlich gibt es ähnlich viele Ausprägungen für das Alter einer Person:

```
survey |>
  transmute(age = 2025 - q002geburt) |>
  distinct()
```

```
# A tibble: 61 x 1
     age
   <dbl>
 1    55
 2    35
 3    62
 4    36
 5    60
 6    68
 7    65
 8    41
 9    51
10    71
# i 51 more rows
```

Es sind mit 61 sogar noch mehr Ausprägungen als für den Preis. Betrachten wir also auch hier ein Histogramm, um einen Eindruck über die Alterverteilung der Probanden im Datensatz zu bekommen. Eine Klassenbreite von 5 Jahren erscheint mir sinnvoll:

```
survey |>
  transmute(age = 2025 - q002geburt) |>
  ggplot() +
  aes(x = age) +
  geom_histogram(binwidth = 5, fill = "lightblue", alpha = 0.8) +
  labs(y = "Häufigeit", x = "Alter", title = "Altersverteilung der Probanden") +
  theme_bw()
```



Ich habe mir in diesem Plot etwas mehr Mühe gemacht und ganz nebenbei ein paar Extras eingebaut, um das Ergebnis ansprechender zu gestalten:

- Füllfarbe Hellblau (`fill`) mit 80%-iger Deckkraft (`alpha`) für die Balken des Histogramms.
- Beschriftungen der Achsen und ein sprechender Titel mittels der `labs`-Funktion.
- Anwendung eines Themes mithilfe von `theme_bw`, was einen helleren Hintergrund bewirkt.

> **Visualisierungen polieren**
>
> Während wir in der explorativen Datenanalyse Wert auf schnelle Ergebnisse legen, wollen wir für Publikationen ansprechende Visualisierungen im Hochglanzformat erstellen. In Kapitel 11 schauen wir explizit auf die Möglichkeiten, Visualisierungen aufzupolieren.

2.2.2.1. Ganze Zahlen (`int`)

Zählvariablen besitzen verglichen mit kontinuierlichen Variablen einen wesentlichen Unterschied. Es handelt sich um so genannte ganze Zahlen, die wir in der Mathematik auch *natürliche Zahlen* nennen. Normalerweise würden wir eine Variable, die nur ganze Zahlen annehmen kann, als diskret und nicht als kontinuierlich bezeichnen. Mit diskret meinen wir gang einfach, dass es eine abzählbare Menge an möglichen Werten gibt und nicht unendlich viele denkbare. Beim Alter sind das die ganzen Zahlen zwischen 0 und dem maximalen Alter, das Menschen erreichen

können. Pragmatisch nehmen wir hier mal 140 an, meines Wissens gab es bis jetzt keinen Menschen, der so alt wurde.

Für diskrete Variablen würden wir in der Regel keinen Boxplot und auch kein Histogramm verwenden, weil diese für kontinuierliche Variablen entwickelt wurden. Wie oben schon beschrieben gibt es bei Zählvariablen Fälle, bei denen wir sie ganz ähnlich wie kontinuierliche Variablen behandeln. Die Variable Alter ist so ein Fall, weil es hier sehr viele Ausprägungen gibt, die zwar diskret sind, aber dennoch eher einer kontinuierlichen Größe ähneln.

Für ganze Zahlen gibt es in R den Datentyp `int`, der in der Abbildung 2.1 auch für Zählvariablen angegeben ist. In der Praxis macht es fast keinen Unterschied, ob wir eine Zählvariable als `double` oder als `int` speichern. Die einzeigen beiden, die mir einfallen, sind:

1. Der Datentyp `int` benötigt weniger Speicherplatz, was bei sehr großen Datensätzen interessant werden könnte.
2. Durch die Verwendung des Datentyps `int` stellen wir auf der Ebene von R sicher, dass die Variable nur ganze Zahlen annehmen kann. Und es macht es für Personen, die mit dem Datensatz nicht vertraut sind, ein Stück transparenter und einfacher zu verstehen.

Machen wir uns die Mühe und wandeln das Alter in den Datentyp `int` um. In dem Zuge speichern wir ihn auch gleich als neue Variable im existierenden Tibble:

```
survey <-
    survey |>
        mutate(Q002age = as.integer(2025 - q002geburt), .after = "q002geburt")
```

Eine neue Spalte können wir mit `mutate` erzeugen. Eine Umwandlung in eine ganze Zahl klappt mit der `as.integer`-Funktion, allerdings nur, wenn der übergebene Wert als ganze Zahl darstellbar ist. Ansonsten gibt es einen Fehler. Mit dem Parameter `.after` können wir die Position der neuen Variable bestimmen, nämlich direkt hinter der Urpsrungsvariable des Geburtsjahres.

Überprüfen wir das Ganze einmal und geben die neue Spalte aus:

```
survey |>
    select(q002geburt, Q002age)
```

```
# A tibble: 2,811 x 2
   q002geburt Q002age
        <dbl>   <int>
 1       1970      55
 2       1990      35
 3       1963      62
 4       1989      36
```

```
 5       1965      60
 6       1957      68
 7       1960      65
 8       1984      41
 9       1974      51
10       1954      71
# i 2,801 more rows
```

Wie alt ist der älteste Teilnehmende?

```
survey |>
    select(Q002age) |>
    arrange(desc(Q002age)) |>
    head(1)
```

```
# A tibble: 1 x 1
  Q002age
    <int>
1      80
```

Und der jüngste?

```
survey |>
    select(Q002age) |>
    arrange(Q002age) |>
    head(1)
```

```
# A tibble: 1 x 1
  Q002age
    <int>
1      18
```

2.2.2.2. Schneller Überblick über eine Spalte mit `skim`

Es gibt ein nützliches Paket mit dem Nmen `skimr`, das unter anderem die gleichnamige Funktion `skim` bereitstellt. Sie gibt uns einen schnellen Überblick über einzelne Variablen oder sogar den gesamten Datensatz.

```
library(skimr)
survey |>
  select(Q002age) |>
  skim() |>
  as_tibble()
```

```
# A tibble: 1 x 12
  skim_type skim_variable n_missing complete_rate numeric.mean numeric.sd
  <chr>     <chr>              <int>         <dbl>        <dbl>      <dbl>
1 numeric   Q002age                0             1         48.4       14.7
# i 6 more variables: numeric.p0 <dbl>, numeric.p25 <dbl>, numeric.p50 <dbl>,
#   numeric.p75 <dbl>, numeric.p100 <dbl>, numeric.hist <chr>
```

Den letzten Schritt `as_tibble()` benötigt ihr nicht, er ist nur notwendig, um die Ausgabe in diesem Buch als Konsolenausgabe zu erzeugen. Wenn ihr `skim` in eurem RStudio verwendet, dann wir automatisch eine Konsolenausgabe erzeugt.

Was sehen wir? Die Funktion `skim` erzeugt einen Tibble, der eine Reihe von berechneten Variablen in Bezug auf die von uns genannte Spalte `Q002age` hat:

- Den Datentyp (`skim_type`), der als `numeric` erkannt wurde
- Anzahl fehlender Werte (`n_missing`) - hier fehlt kein Wert
- Füllgrad (`complete_rate`), hier dementsprechend 1 oder 100%
- Arithmetisches Mittel: `numeric.mean`
- Standardabweichung: `numeric.sd`
- Kleinster Wert: `numeric.p0`
- Unteres Quartil (Q1 oder 25%-Wert): `numeric.p25`
- Median (Q2 oder 50%-Wert): `numeric.p50`
- Oberes Quartil (Q3 oder 75%-Wert): `numeric.p75`
- Größter Wert: `numeric.p100`
- ASCII-Histogram: `numeric.hist`

Das sind eine ganze Menge interessanter Werte, die wir mit einem Befehl errechnen können. Und es wird noch besser: Dadurch, dass es sich im Ergebnis um einen Tibble handelt, also einen weiteren Datensatz, können wir mit ihm wie mit jedem anderen Tibble arbeiten. Zum Beispiel nur die beiden Varianten des Durchschnitts auswwählen:

```
survey |>
  select(Q002age) |>
  skim() |>
  yank("numeric") |>
  select(mean_age = mean, median_age = p50)
```

Variable type: numeric

mean_age	median_age
48.39	50

2.3. Kategoriale Variablen

2.3.1. Nominale Variablen

> Categorical variables are variables recording which category an observation is in - simple enough! The color of a flower is an example of a categorical variable. Is the flower white, orange, or red? None of those options is "more" than the others; they're just different. (Huntington-Klein 2026)

Wie der Name schon sagt verwenden wir kategoriale Variablen dann, wenn es definierte Kategorien gibt, in die eine Messung eingruppiert werden kann. Gleichzeitig ist keine Gruppe besser oder schlechter als die andere, sie sind nur unterschiedlich.

2.3.2. Ordinale Variablen

> Ordinal variables are variables where some values are "more" and others are "less," but there's not necessarily a rule as to how much more "more" is. (Huntington-Klein 2026)

2.4. Qualitative Variablen

> Qualitative variables are a sort of catch-all category for everything else. They aren't numeric in nature, but also they're not categorical. The text of a Washington Post headline is an example of a qualitative variable. (Huntington-Klein 2026)

Teil II.

Projekt 2: News

3. Laden der Daten

In diesem ersten Kapitel des Projekts erkunden wir wie immer den neuen Datensatz. Dazu laden wir ihn mit R, und zwar als Tibble, damit wir auf alle Funktionen des Tidyverse zurükgreifen können. Anschließend lernen wir etwas über die enthaltenen Informationen.

3.1. Tagesschau-Beiträge

Die Daten stammen von der Webseite Tagesschau.de, die täglich aktuelle Nachrichten veröffentlicht. Für dieses Projekt wurden für den Zeitraum **05.01.2006** bis **31.12.2025** insgesamt **59.500** Nachrichtenartikel gesammelt. Wer gerne mehr über den Prozess der Datensammlung erfahren möchte findet im Anhang eine detaillierte Beschreibung der Python-Skripte, die für die Erstellung dieses Datensatzes verwendet wurden.

Für unsere Analysen gehen wir wie immer davon aus, dass die Daten bereits vorliegen. In diesem Fall als CSV-Datei, die wir sofort als Tibble in R laden können.

```
library(tidyverse)
ts <- read_csv("data/tagesschau.csv")
```

Mit der Funktion `glimpse()` bekommen wir einen schnellen Überblick über die Struktur des Datensatzes.

```
ts |>
  glimpse()
```

```
Rows: 59,500
Columns: 21
$ supertitle     <chr> "ARD-DeutschlandTrend Januar 2006", "Treffen der EU-
Inn~
$ title          <chr> "ARD-DeutschlandTrend Januar 2006", "Grundzüge für geme~
$ date_time      <dttm> 2006-01-05 10:50:33, 2006-01-13 13:47:00, 2006-01-
13 1~
$ author         <chr> "Jörg Schönenborn", "tagesschau.de", "tagesschau.de", "~
$ ressort        <chr> "inland", "ausland", "inland", "ausland", "wirtschaft",~
```

```
$ url            <chr> "https://www.tagesschau.de/inland/deutschlandtrend/meld~
$ thumbnail      <chr> "\"https://images.tagesschau.de/image/47dedcab-ee73-
4e8~
$ tag            <chr> NA, NA, "INTERVIEW", NA, "HINTERGRUND", NA, NA, NA, "IN~
$ shorttext      <chr> "Angela Merkel hat in den ersten sechs Wochen ihrer Amt~
$ description    <chr> "Angela Merkel hat in den ersten sechs Wochen ihrer Amt~
$ keywords       <chr> "[\"DeutschlandTrend\"]", "[\"Meldung\"]", "[\"Intervie~
$ date_modified  <dttm> 2021-01-28 10:32:31, 2023-03-01 23:51:29, 2023-03-
01 1~
$ canonical_url  <chr> "https://www.tagesschau.de/inland/deutschlandtrend/meld~
$ language       <chr> "de", "de", "de", "de", "de", "de", "de", "de", "de", "~
$ paragraphs     <chr> "[\"Bundeskanzlerin Angela Merkel hat in den ersten sec~
$ text           <chr> "Bundeskanzlerin Angela Merkel hat in den ersten sechs ~
$ word_count     <dbl> 569, 406, 658, 264, 601, 399, 262, 516, 801, 433, 567, ~
$ image_urls     <chr> "[\"https://images.tagesschau.de/image/47dedcab-ee73-
4e~
$ image_captions <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ related_links  <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ source_file    <chr> "2006-01-05-articles.jsonl", "2006-01-13-articles.jsonl~
```

Wir sehen, dass der Datensatz **59.500** Zeilen und **21** Spalten enthält. Jede Zeile entspricht einem Nachrichtenbeitrag, und jede Spalte enthält Informationen über diesen Beitrag, wie zum Beispiel den Titel, den Text, das Veröffentlichungsdatum und einige interessante Informationen mehr. Lasst uns einen Blick auf ein paar ausgewählte Spalten werfen.

4. Faktoren

Viele Variablen im Tagesschau-Datensatz sind *kategorial*: `ressort`, `language`, `author`, manchmal auch `tag` oder `supertitle`. In R werden solche Kategorien häufig als **Faktoren** (*factors*) dargestellt. Ein Faktor ist im Kern ein Vektor mit festen Ausprägungen (*levels*). Das ist in der Datenanalyse nützlich, weil

- Grafiken und Tabellen automatisch sinnvoll gruppieren,
- eine feste Reihenfolge der Kategorien reproduzierbar ist,
- Modelle kategoriale Merkmale korrekt behandeln können.

Im Tidyverse ist `forcats` (Teil von `tidyverse`) das Werkzeug für Faktor-Operationen.

4.1. Überblick: Welche Variablen sind gute Faktoren?

Ein schneller Check ist: Welche Spalten sind Text (`<chr>`) und haben eher wenige unterschiedliche Werte?

```
# Anzahl unterschiedlicher Werte je Zeichen-Spalte
char_uniques <- ts |>
  summarise(
    across(where(is.character), ~ n_distinct(.x, na.rm = TRUE))
  ) |>
  pivot_longer(everything(), names_to = "variable", values_to = "n_distinct") |>
  arrange(n_distinct)

char_uniques
```

```
# A tibble: 18 x 2
   variable       n_distinct
   <chr>               <int>
 1 language                4
 2 ressort                38
 3 tag                   375
 4 author               5216
```

```
 5 source_file          6351
 6 image_captions       9434
 7 related_links       17122
 8 thumbnail           41449
 9 keywords            41962
10 supertitle          46474
11 image_urls          48806
12 title               58902
13 paragraphs          58986
14 text                58986
15 shorttext           59056
16 description         59059
17 canonical_url       59231
18 url                 59433
```

Interpretation:

- Kleine `n_distinct` (z.B. `language`) → sehr gute Kandidaten für Faktoren.
- Große `n_distinct` (z.B. `title`, `url`) → eher Identifikatoren/Free-Text, meistens *kein* Faktor.

4.2. Häufigkeiten: Ressorts zählen und visualisieren

Bei News-Daten ist `ressort` meist eine der wichtigsten Kategorien.

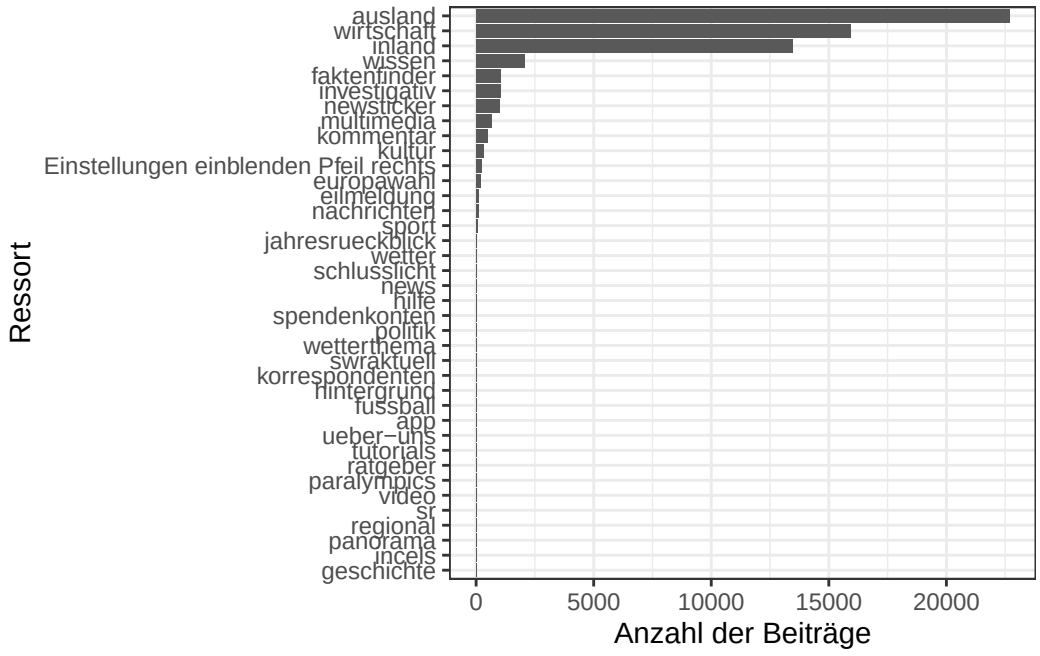
```r
# Häufigkeitstabelle
ressort_counts <- ts |>
  count(ressort, sort = TRUE)

ressort_counts
```

```
# A tibble: 39 x 2
   ressort          n
   <chr>        <int>
 1 ausland      22675
 2 wirtschaft   15935
 3 inland       13463
 4 wissen        2045
 5 faktenfinder  1059
 6 investigativ  1042
 7 newsticker    1021
```

```
 8 multimedia     650
 9 kommentar      468
10 kultur         340
# i 29 more rows
```

Für eine Visualisierung ist es hilfreich, `ressort` als Faktor zu nutzen und nach Häufigkeit zu ordnen.

```
ressort_counts |>
  filter(!is.na(ressort)) |>
  mutate(ressort = fct_reorder(ressort, n)) |>
  ggplot(aes(x = ressort, y = n)) +
  geom_col() +
  coord_flip() +
  theme_bw() +
  labs(x = "Ressort", y = "Anzahl der Beiträge")
```



Warum ist das nützlich?

- Du erkennst sofort, welche Ressorts dominieren (wichtig für Stichproben, Bias, Gewichtung).
- Viele weitere Analysen (Zeitreihen, Textfeatures) lassen sich sinnvoll nach Ressort splitten.

4.3. Zu viele Kategorien: Lumpen (seltene Werte zusammenfassen)

Bei Kategorien wie `author` gibt es oft sehr viele Ausprägungen. Für Auswertungen und Plots ist es dann sinnvoll, seltene Werte in `Other` zusammenzufassen.

```
# Top-Autor:innen + Other
# (Falls author sehr oft NA ist: NAs explizit als Kategorie behandeln)

ts |>
  mutate(
    author = fct_explicit_na(author, na_level = "(fehlend)"),
    author = fct_lump_n(author, n = 15)
  ) |>
  count(author, sort = TRUE)
```

```
# A tibble: 16 x 2
   author                    n
   <fct>                 <int>
 1 Other                 20487
 2 tagesschau.de         19926
 3 (fehlend)             16625
 4 Kai Küstner             387
 5 Martin Bohne            207
 6 Silvia Stöber           201
 7 Stephan Ueberbach       185
 8 Helga Schmidt           184
 9 Ralph Sina              182
10 Jakob Mayr              177
11 Karin Bensch            177
12 Patrick Gensing         176
13 Frank Bräutigam         157
14 Angela Göpfert          149
15 Christoph Prössl        148
16 Jan-Christoph Kitzler   132
```

Das ist ein typischer Schritt in der Datenanalyse:

- reduziert visuelle Unordnung,
- verhindert, dass „Einzelfälle“ die Story dominieren,
- stabilisiert Modelle (zu viele Kategorien führen sonst schnell zu Overfitting).

4.4. Sinnvolle Reihenfolgen: Faktoren nach Kennzahlen ordnen

Ein Faktor muss nicht alphabetisch sortiert sein. Häufig willst du Kategorien nach einer analytischen Kennzahl ordnen.

Beispiel: Welche Ressorts haben im Median die längsten Texte (über `word_count`)?

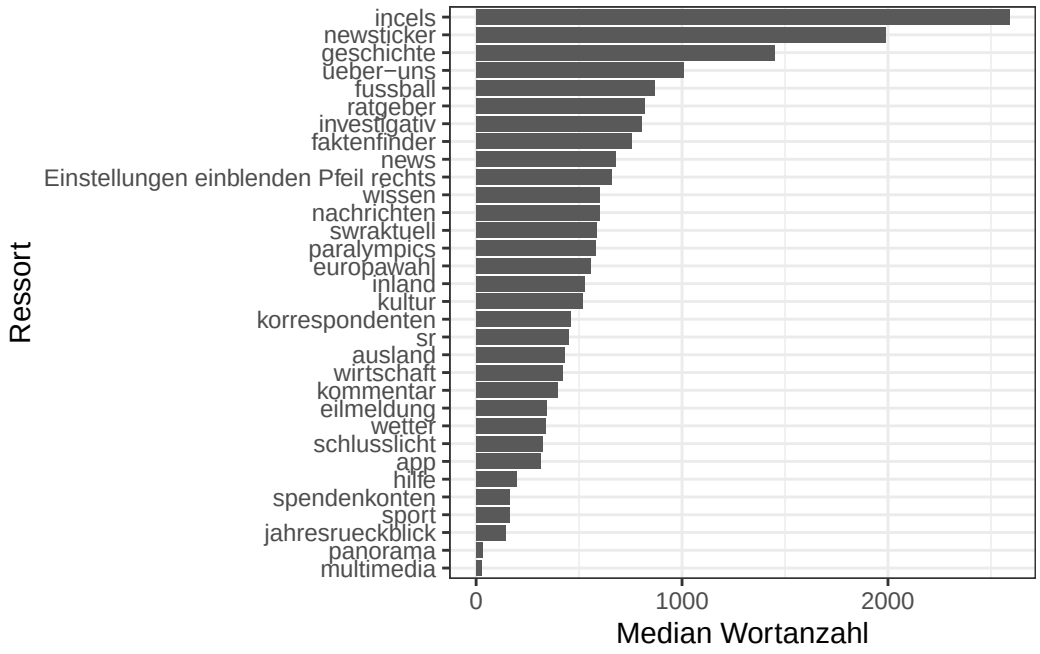
```
ressort_wordcount <- ts |>
  filter(!is.na(ressort), !is.na(word_count)) |>
  group_by(ressort) |>
  summarise(
    n = n(),
    median_word_count = median(word_count),
    .groups = "drop"
  ) |>
  arrange(desc(median_word_count))

ressort_wordcount
```

```
# A tibble: 32 x 3
   ressort                                n median_word_count
   <chr>                              <int>             <dbl>
 1 incels                                 1              2589
 2 newsticker                          1021              1987
 3 geschichte                             1              1451
 4 ueber-uns                              2              1010.
 5 fussball                               4               864.
 6 ratgeber                               2               820.
 7 investigativ                        1042               804.
 8 faktenfinder                        1059               754
 9 news                                   8               678.
10 Einstellungen einblenden Pfeil rechts  210               657
# i 22 more rows
```

Und als Plot, nach Median-Wortanzahl geordnet:

```
ressort_wordcount |>
  mutate(ressort = fct_reorder(ressort, median_word_count)) |>
  ggplot(aes(x = ressort, y = median_word_count)) +
  geom_col() +
  coord_flip() +
```

```
  theme_bw() +
  labs(x = "Ressort", y = "Median Wortanzahl")
```



So ein Plot ist ein guter Einstieg, um Hypothesen zu formulieren (z.B. „Politikartikel sind länger als Sportmeldungen“).

4.5. Kategorien bereinigen: Recoding und Missingness sichtbar machen

Oft sind Kategorien „nicht sauber“: unterschiedliche Schreibweisen, leere Strings, oder NAs. Für Analytics ist es wichtig, diese Fälle bewusst zu behandeln.

Beispiel: `language` sollte im Datensatz meistens `de` sein.

```
# Sprache: NA explizit machen + Häufigkeiten

ts |>
  mutate(language = fct_explicit_na(language, na_level = "(fehlend)")) |>
  count(language, sort = TRUE)
```

```
# A tibble: 5 x 2
  language      n
  <fct>     <int>
1 de        59495
2 en            2
3 es            1
4 it            1
5 (fehlend)     1
```

Wenn du Kategorien zusammenführen möchtest (z.B. Synonyme), hilft `fct_recode()`:

```
# Beispiel-Recode (nur als Muster; passe Mapping bei Bedarf an deine Daten an)

ts |>
  mutate(
    ressort = fct_recode(
      ressort,
      inland = "inland",
      ausland = "ausland"
    )
  ) |>
  count(ressort, sort = TRUE)
```

```
# A tibble: 39 x 2
   ressort          n
   <fct>        <int>
 1 ausland      22675
 2 wirtschaft   15935
 3 inland       13463
 4 wissen        2045
 5 faktenfinder  1059
 6 investigativ  1042
 7 newsticker    1021
 8 multimedia     650
 9 kommentar      468
10 kultur         340
# i 29 more rows
```

4.6. Abgeleitete Faktoren: Wochentag und Tageszeit

Faktoren entstehen nicht nur aus Textspalten. Gerade Zeitstempel werden in Analytics oft in kategoriale Einheiten transformiert, um Muster sichtbar zu machen.

```
# Wochentag (deutsche Labels) und Stunde
# Hinweis: base::weekdays() hängt von der Locale ab; daher nutzen wir lubridate::wday()

ts |>
  mutate(
    weekday = wday(date_time, label = TRUE, abbr = FALSE, week_start = 1),
    hour = hour(date_time)
  ) |>
  count(weekday, sort = TRUE)
```

```
# A tibble: 8 x 2
  weekday        n
  <ord>      <int>
1 Donnerstag 10649
2 Mittwoch   10503
3 Dienstag   10158
4 Freitag     9345
5 Montag      8979
6 Samstag     5107
7 Sonntag     4730
8 <NA>          29
```

Wochentage als Faktor sind besonders nützlich für:

- Publikationsmuster (z.B. weniger Beiträge am Wochenende?),
- Kontrollvariablen in Modellen,
- segmentierte Reports (“Mo–Fr” vs. Wochenende).

5. Variablen

Nachdem wir in Kapitel 3 die Tagesschau-Daten geladen und einen ersten Überblick über die enthaltenen Informationen gewonnen haben, wollen wir uns nun genauer mit den einzelnen Variablen beschäftigen.

5.1. Übersicht der Variablen

Um eine Übersicht über die in einem Datensatz enthaltenen Variablen zu bekommen, können wir die Funktion `glimpse()` verwenden. Sie gibt uns einen schnellen Überblick über die Struktur des Datensatzes, einschließlich der Namen der Variablen, ihrer Datentypen und einiger Beispielwerte.

```
ts |>
  glimpse()
```

```
Rows: 59,500
Columns: 21
$ supertitle     <chr> "ARD-DeutschlandTrend Januar 2006", "Treffen der EU-
Inn~
$ title          <chr> "ARD-DeutschlandTrend Januar 2006", "Grundzüge für geme~
$ date_time      <dttm> 2006-01-05 10:50:33, 2006-01-13 13:47:00, 2006-01-
13 1~
$ author         <chr> "Jörg Schönenborn", "tagesschau.de", "tagesschau.de", "~
$ ressort        <chr> "inland", "ausland", "inland", "ausland", "wirtschaft",~
$ url            <chr> "https://www.tagesschau.de/inland/deutschlandtrend/meld~
$ thumbnail      <chr> "\"https://images.tagesschau.de/image/47dedcab-ee73-
4e8~
$ tag            <chr> NA, NA, "INTERVIEW", NA, "HINTERGRUND", NA, NA, NA, "IN~
$ shorttext      <chr> "Angela Merkel hat in den ersten sechs Wochen ihrer Amt~
$ description    <chr> "Angela Merkel hat in den ersten sechs Wochen ihrer Amt~
$ keywords       <chr> "[\"DeutschlandTrend\"]", "[\"Meldung\"]", "[\"Intervie~
$ date_modified  <dttm> 2021-01-28 10:32:31, 2023-03-01 23:51:29, 2023-03-
01 1~
$ canonical_url  <chr> "https://www.tagesschau.de/inland/deutschlandtrend/meld~
```

```
$ language       <chr> "de", "de", "de", "de", "de", "de", "de", "de", "de", "~
$ paragraphs     <chr> "[\"Bundeskanzlerin Angela Merkel hat in den ersten sec~
$ text           <chr> "Bundeskanzlerin Angela Merkel hat in den ersten sechs ~
$ word_count     <dbl> 569, 406, 658, 264, 601, 399, 262, 516, 801, 433, 567, ~
$ image_urls     <chr> "[\"https://images.tagesschau.de/image/47dedcab-ee73-
4e~
$ image_captions <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ related_links  <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ source_file    <chr> "2006-01-05-articles.jsonl", "2006-01-13-articles.jsonl~
```

5.2. Fehlende Werte

Fehlende Werte (*missing values*) sind in realen Datensätzen völlig normal: Manche Informationen sind für einen Beitrag nicht vorhanden (z.B. kein:e Autor:in), wurden beim Scraping nicht gefunden oder sind nur für bestimmte Ressorts sinnvoll.

Ein erster, sehr nützlicher Schritt ist ein „Missingness-Profil“: Welche Spalten haben überhaupt fehlende Werte – und wie viele?

```
ts |>
  summarise(
    across(
      everything(),
      list(n_missing = ~ sum(is.na(.x)), pct_missing = ~ mean(is.na(.x))),
      .names = "{.col}__{.fn}"
    )
  ) |>
  pivot_longer(everything(), names_to = "metric", values_to = "value") |>
  separate(metric, into = c("variable", "metric"), sep = "__") |>
  pivot_wider(names_from = metric, values_from = value) |>
  arrange(desc(pct_missing))
```

```
# A tibble: 21 x 3
   variable       n_missing pct_missing
   <chr>              <dbl>       <dbl>
 1 image_captions     49446     0.831
 2 related_links      42196     0.709
 3 tag                32797     0.551
 4 author             16625     0.279
 5 image_urls          6140     0.103
 6 keywords             638     0.0107
```

```
 7 date_modified        627     0.0105
 8 paragraphs           323     0.00543
 9 text                 323     0.00543
10 word_count           323     0.00543
# i 11 more rows
```

Das Ergebnis hilft dir analytisch sofort weiter:

- Spalten mit *sehr vielen* fehlenden Werten eignen sich oft eher als optionale Zusatzinformationen.
- Spalten mit *wenigen* fehlenden Werten sind meist robuste „Kernvariablen“.
- Wenn wichtige Variablen viele NAs enthalten, lohnt sich Ursachenforschung (Erhebung, Scraper, Parsing, Definition der Variable).

Wenn du eine kompakte, gut lesbare Gesamtsicht möchtest, ist `skimr` sehr praktisch (inkl. Missingness, Verteilungen, Beispiele). Damit die Kapitel auch ohne das Paket rendern, ist es hier optional:

```
if (requireNamespace("skimr", quietly = TRUE)) {
  skimr::skim(ts)
} else {
  cat("Optional: install.packages('skimr') für eine kompakte Variable-Übersicht.\n")
}
```

Tabelle 5.1.: Data summary

Name	ts
Number of rows	59500
Number of columns	21
_____	
Column type frequency:	
character	18
numeric	1
POSIXct	2
_____	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
supertitle	0	1.00	3	67	0	46474	0
title	0	1.00	3	146	0	58902	0
author	16625	0.72	3	476	0	5216	0
ressort	5	1.00	2	37	0	38	0
url	0	1.00	38	174	0	59433	0
thumbnail	22	1.00	67	320	0	41449	0
tag	32797	0.45	2	76	0	375	0
shorttext	7	1.00	10	717	0	59056	0
description	7	1.00	10	807	0	59059	0
keywords	638	0.99	5	763	0	41962	0
canonical_url	6	1.00	31	174	0	59231	0
language	1	1.00	2	2	0	4	0
paragraphs	323	0.99	22	84250	0	58986	0
text	323	0.99	18	84140	0	58986	0
image_urls	6140	0.90	128	166482	0	48806	0
image_captions	49446	0.17	10	4379	0	9434	0
related_links	42196	0.29	42	27104	0	17122	0
source_file	0	1.00	25	25	0	6351	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
word_count	323	0.99	540.12	402.18	3	320	461	657	11165	

Variable type: POSIXct

skim_variable	n_missing	complete_rate	min	max	median	n_unique
date_time	29	1.00	2006-01-05 10:50:33	2025-12-31 20:29:48	2023-06-05 17:39:28	59372
date_modified	627	0.99	2007-05-10 14:15:00	2026-02-09 17:57:43	2024-11-07 18:20:14	58867

5.3. Duplikate

Duplikate können in News-Daten aus verschiedenen Gründen entstehen: ein Artikel wurde mehrfach gespeichert, die gleiche URL taucht in mehreren Quellfiles auf, oder Inhalte sind sehr ähnlich.

In der Praxis definieren wir Duplikate über eine eindeutige ID. Bei Webdaten ist das häufig die `url` (oder `canonical_url`). Schauen wir zuerst, ob es URLs gibt, die mehrfach vorkommen:

```
ts |>
  count(url, sort = TRUE) |>
  filter(!is.na(url), n > 1)
```

```
# A tibble: 11 x 2
   url                                                                    n
   <chr>                                                              <int>
 1 https://www.tagesschau.de/multimedia/podcast/11km/11km-feed-100.html      53
 2 https://www.phoenix.de/livestream.html                                    5
 3 https://www.tagesschau.de/multimedia/podcasts/podcast-11km-101.html       4
 4 https://www.tagesschau.de/multimedia/podcast/11km/podcast-11km-2788.ht~    2
 5 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-corona-
s~     2
 6 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-kontroll~    2
 7 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-rauchen-
~     2
 8 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-streit-
u~     2
 9 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-tabubruc~    2
10 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-weltglue~    2
11 https://www.tagesschau.de/multimedia/podcast/15-minuten/audio-wer-mach~    2
```

Wenn du wissen willst, *welche* Datensätze dahinterstehen, kannst du dir einzelne Fälle anzeigen lassen. Das ist ein typischer Debugging-Schritt in der Datenbereinigung:

```
ts |>
  add_count(url, name = "n_url") |>
  filter(!is.na(url), n_url > 1) |>
  select(url, date_time, ressort, title, n_url) |>
  arrange(desc(n_url), url, date_time)
```

```
# A tibble: 78 x 5
```

```
   url                                   date_time           ressort title n_url
   <chr>                                 <dttm>              <chr>   <chr> <int>
 1 https://www.tagesschau.de/multimedia~ 2023-04-25 11:15:00 multim~ 11KM~    53
 2 https://www.tagesschau.de/multimedia~ 2023-05-30 05:46:00 multim~ 11KM~    53
 3 https://www.tagesschau.de/multimedia~ 2023-06-28 05:14:00 multim~ 11KM~    53
 4 https://www.tagesschau.de/multimedia~ 2023-07-07 07:07:00 multim~ 11KM~    53
 5 https://www.tagesschau.de/multimedia~ 2023-09-26 09:14:00 multim~ 11KM~    53
 6 https://www.tagesschau.de/multimedia~ 2023-10-19 09:02:00 multim~ 11KM~    53
 7 https://www.tagesschau.de/multimedia~ 2023-11-30 08:58:00 multim~ 11KM~    53
 8 https://www.tagesschau.de/multimedia~ 2023-12-19 08:22:00 multim~ 11KM~    53
 9 https://www.tagesschau.de/multimedia~ 2024-01-31 07:04:00 multim~ 11KM~    53
10 https://www.tagesschau.de/multimedia~ 2024-02-28 06:29:00 multim~ 11KM~    53
# i 68 more rows
```

Für viele Analysen (z.B. Zählen, Zeitreihen) willst du Duplikate entfernen, damit Ergebnisse nicht „aufgeblasen“ werden. Wenn `url` eindeutig sein soll, kannst du eine deduplizierte Version erzeugen:

```
ts_dedup <- ts |>
  arrange(date_time) |>
  distinct(url, .keep_all = TRUE)

ts |>
  summarise(n_rows = n()) |>
  bind_cols(ts_dedup |> summarise(n_rows_dedup = n()))
```

```
# A tibble: 1 x 2
  n_rows n_rows_dedup
   <int>        <int>
1  59500        59433
```

Wichtig: Welche Zeile du bei Duplikaten behältst (erste/letzte, nach `date_modified`, nach Datenqualität) ist eine fachliche Entscheidung. `arrange()` vor `distinct()` macht diese Entscheidung explizit und reproduzierbar.

5.4. Wertebereiche

Wertebereiche (*ranges*) sind ein schneller Plausibilitätscheck. Gerade numerische Variablen enthalten manchmal Ausreißer oder „kaputte“ Werte (z.B. negative Längen, extrem große Zählwerte), die aus Parsing- oder Scraping-Problemen stammen.

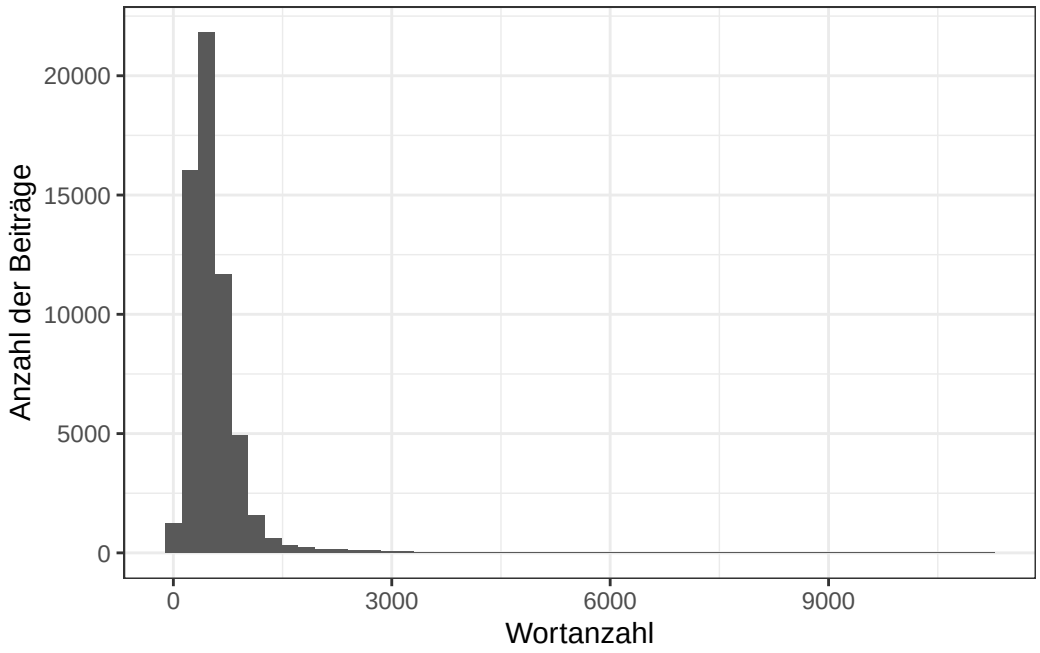
Im Datensatz gibt es z.B. `word_count` (Wortanzahl) und oft auch `paragraphs` (Absatzanzahl). Wir schauen uns typische Kennzahlen und Ausreißer an:

```
ts |>
  summarise(
    n = n(),
    word_count_min = min(word_count, na.rm = TRUE),
    word_count_p25 = quantile(word_count, 0.25, na.rm = TRUE),
    word_count_median = median(word_count, na.rm = TRUE),
    word_count_p75 = quantile(word_count, 0.75, na.rm = TRUE),
    word_count_max = max(word_count, na.rm = TRUE),
    paragraphs_min = min(paragraphs, na.rm = TRUE),
    paragraphs_median = median(paragraphs, na.rm = TRUE),
    paragraphs_max = max(paragraphs, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 9
      n word_count_min word_count_p25 word_count_median word_count_p75
  <int>          <dbl>          <dbl>             <dbl>          <dbl>
1 59500              3            320               461            657
# i 4 more variables: word_count_max <dbl>, paragraphs_min <chr>,
#   paragraphs_median <chr>, paragraphs_max <chr>
```

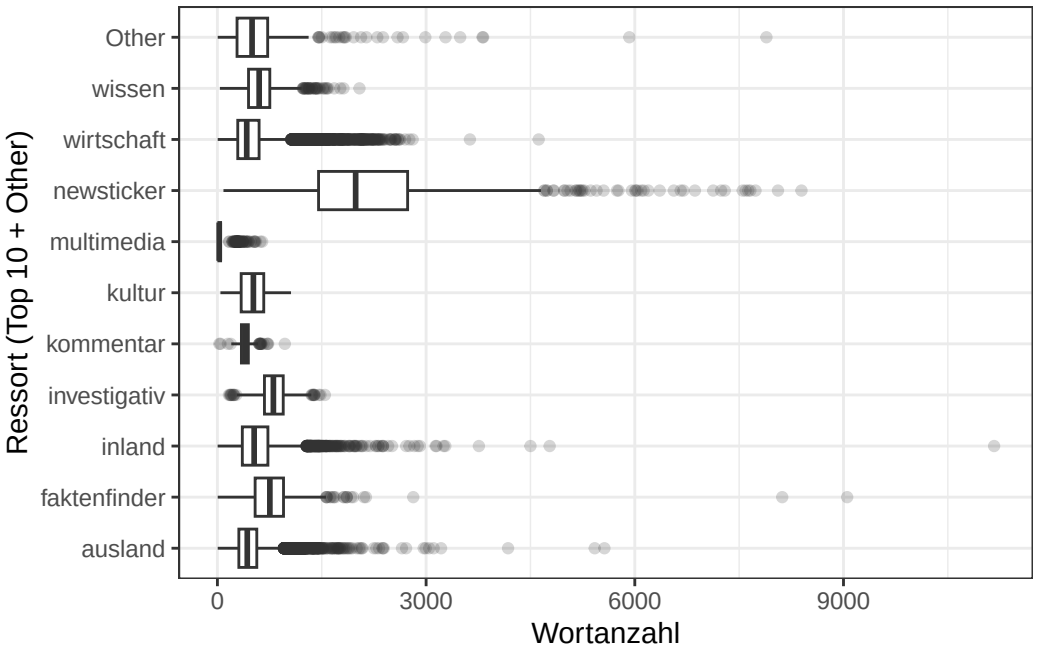
Eine Visualisierung macht Verteilungen und Ausreißer noch schneller greifbar. Ein Histogramm zeigt dir, wie „lang“ Tagesschau-Beiträge typischerweise sind:

```
ts |>
  ggplot(aes(x = word_count)) +
  geom_histogram(bins = 50, na.rm = TRUE) +
  theme_bw() +
  labs(x = "Wortanzahl", y = "Anzahl der Beiträge")
```



Und ein Boxplot nach Ressort ist nützlich, um Unterschiede zwischen Kategorien sichtbar zu machen (z.B. sind Wirtschaftsartikel im Schnitt länger?):

```
ts |>
  filter(!is.na(ressort), !is.na(word_count)) |>
  mutate(ressort = fct_lump_n(ressort, n = 10)) |>
  ggplot(aes(x = ressort, y = word_count)) +
  geom_boxplot(outlier.alpha = 0.2) +
  coord_flip() +
  theme_bw() +
  labs(x = "Ressort (Top 10 + Other)", y = "Wortanzahl")
```



Warum ist das für Data Analytics hilfreich?

- Du bekommst ein Gefühl für „typische“ Inhalte (Baseline), bevor du Modelle baust.
- Ausreißer-Fälle sind oft inhaltlich spannend (Breaking News) *oder* Datenprobleme.
- Kategorienvergleiche liefern schnell Hypothesen für tiefergehende Analysen (z.B. Trends je Ressort).

6. Zeit

6.1. Verteilung der Beiträge über die Zeit

Nachrichten haben einen inhärenten zeitlichen Charakter, da sie sich auf aktuelle Ereignisse beziehen. Es ist daher interessant zu sehen, wie die Anzahl der veröffentlichten Nachrichtenbeiträge über die Zeit verteilt ist. Die entsprechende Spalte für den Zeitbezug erkennen wir in der Ausgabe der `glimpse()`-Funktion oben am Datentyp `<dttm>`, was für *datetime* steht. Schauen wir uns ein paar Beispiele an:

```
ts |>
  select(date_time)
```

```
# A tibble: 59,500 x 1
   date_time
   <dttm>
 1 2006-01-05 10:50:33
 2 2006-01-13 13:47:00
 3 2006-01-13 10:18:00
 4 2006-01-14 14:24:00
 5 2006-01-15 14:44:00
 6 2006-01-18 19:30:00
 7 2006-01-25 17:37:00
 8 2006-02-02 10:52:26
 9 2006-02-02 08:39:00
10 2006-02-14 14:20:00
# i 59,490 more rows
```

Wir erkennen an der Ausgabe der ersten Zeilen wie genau ein Wert vom *datetime*-Datentyp aufgebaut ist. Es handelt sich um ein standardisiertes Format mit dem Namen ISO-8601, das einer festen Syntax folgt. Der erste Teil stellt das Datum mit seinen Bestandteilen Jahr, Monat und Tag dar, jeweils als vierstellige (Jahr) und zweistellige (Monat, Tag) Zahlen, getrennt mit einem Bindestrich. Danach folgt, getrennt durch ein Leerzeichen, die Uhrzeitangabe. Hier ist das Format wie gewohnt: `HH:MM:ss`, also jeweils zwei Ziffern für die Stunde, Minute und Sekunde, jeweils durch einen Doppelpunkt getrennt.

Weil das Format standardisiert ist, gibt es entsprechende Funktionen, mit denen wir jeden Bestandteil extrahieren wollen. Mit der Funktion `year()` bekommen wir etwa nur das Jahr als Zahl. Ensprechende Funktionen gibt es auch für andere Datums- und Zeitbestandteile. Details zur Arbeit mit Datum und Zeit findet ihr im Kapitel Dates and times aus Wickham, Çetinkaya-Rundel, und Grolemund (2023).

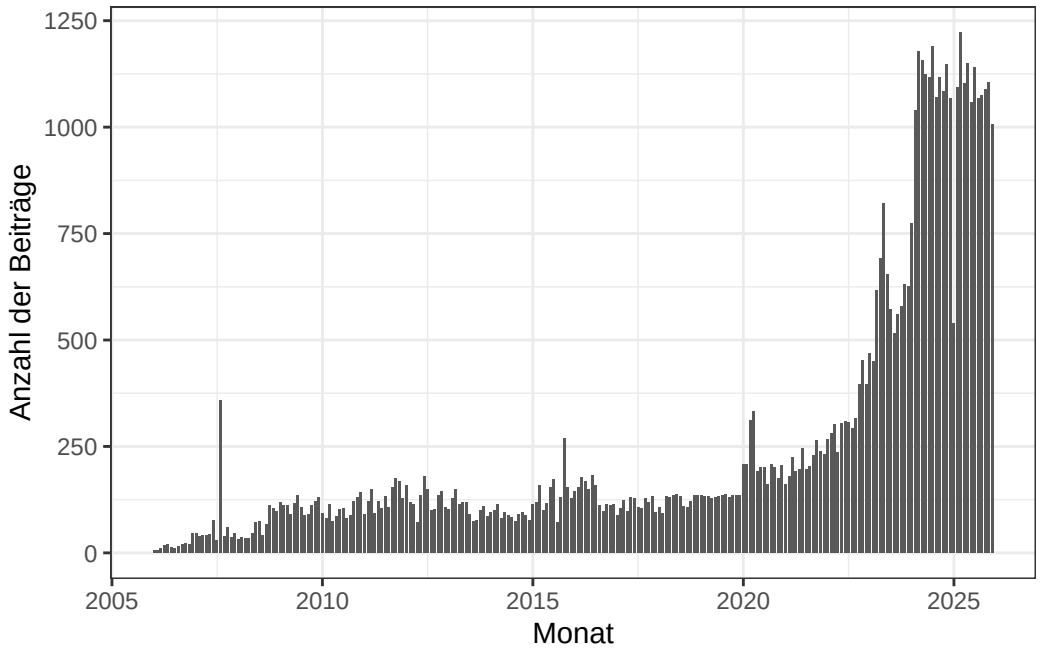
Gleichzeitig können wir ein Datum abrunden - klingt komisch? Dabei schneiden wir einfach den Teil des Zeitstempels ab, der unseren gewünschten Detailgrad überschreitet. Wenn wir etwa eine Analyse auf Monatsbasis erstellen wollen, dann können wir jedes Datum auf den jeweils ersten des Monats runden, im dem sich der Zeitstempel befindet:

```
ts |>
  mutate(date_month = floor_date(date_time, unit = "month"), .keep = "used")
```

```
# A tibble: 59,500 x 2
   date_time           date_month
   <dttm>              <dttm>
 1 2006-01-05 10:50:33 2006-01-01 00:00:00
 2 2006-01-13 13:47:00 2006-01-01 00:00:00
 3 2006-01-13 10:18:00 2006-01-01 00:00:00
 4 2006-01-14 14:24:00 2006-01-01 00:00:00
 5 2006-01-15 14:44:00 2006-01-01 00:00:00
 6 2006-01-18 19:30:00 2006-01-01 00:00:00
 7 2006-01-25 17:37:00 2006-01-01 00:00:00
 8 2006-02-02 10:52:26 2006-02-01 00:00:00
 9 2006-02-02 08:39:00 2006-02-01 00:00:00
10 2006-02-14 14:20:00 2006-02-01 00:00:00
# i 59,490 more rows
```

In der Ausgabe seht ihr links das Originaldatum mit allen Details und rechts das abgeschnittene und auf den ersten des jeweiligen Monats gerundete Datum. Das Schöne an `floor_date()` ist, dass es als Ergebnis einen Wert vom gleichen Datentyp erzeugt, also auch ein *datetime*. Warum ist das gut? Weil wir damit in der Visualisieurung gut klarkommen, wie ihr im nächsten Schritt sehen werdet:

```
ts |>
  mutate(date_month = floor_date(date_time, "month")) |>
  ggplot() +
  aes(x = date_month) +
  geom_bar() +
  theme_bw() +
  labs(x = "Monat", y = "Anzahl der Beiträge")
```



Schaut mal auf die x-Achse. Fällt euch was auf?

Teil III.

Grundlagen

7. Daten laden

8. Pipe-Operator

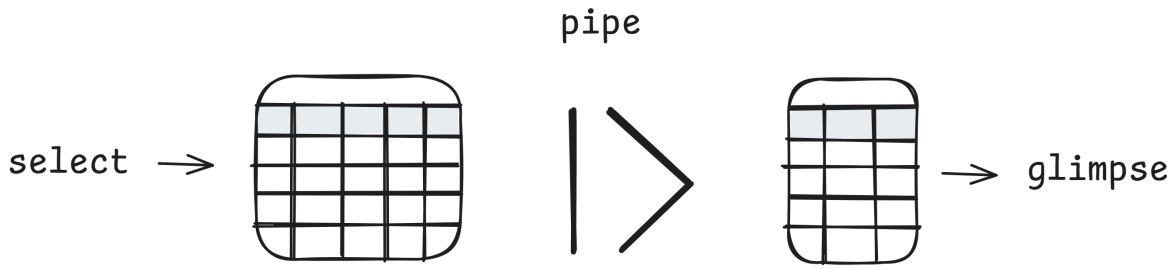


Abbildung 8.1.: Die Pipe leitet das Ergebnis des ersten Befehls an den zweiten Befehl weiter.

9. Filtern

10. Aggregieren

11. Visualisuerungen polieren

Anhang A: Scraping

In diesem Anhang findest du eine detaillierte Beschreibung der Python-Skripte, die für die Erstellung des Datensatzes aus Kapitel 3.1 verwendet wurden. Diese Skripte wurden entwickelt, um die Nachrichtenbeiträge von Tagesschau.de zu sammeln und in einem strukturierten Format zu speichern.

Quellen

Huntington-Klein, Nick. 2026. *The effect: an introduction to research design and causality*. Second edition. A Chapman & Hall Book. Boca Raton London New York: CRC Press.

Wickham, Hadley, Mine Çetinkaya-Rundel, und Garrett Grolemund. 2023. *R for data science: import, tidy, transform, visualize, and model data*. 2nd edition. Sebastopol, CA: O'Reilly Media, Inc.