Bio and Business Data Science

Nicolas Meseth

Inhalt

Vorwort

Die vorliegende Webseite stellt das Skript für meine Lehrveranstaltungen zum Thema Data Science dar. Es ist momentan im Aufbau und es kommen wöchentlich neue Inhalte hinzu. Zu einem späteren Zeitpunkt soll es in Form eines Buches veröffentlicht werden. Bis dahin bitte ich um Geduld.

Es gibt eine PDF-Version, die hier heruntergeladen werden kann: PDF-Version "Bio and Business Data Science".

Für Hinweise zu Fehlern bin ich dankbar. Am besten per E-Mail unter `n.meseth@hs-osnabrueck.de` oder auf Twitter.

Part I

Fall: Campusbier

Einführung

Die Campusbier-Fallstudie ist in dem Sinne eine besondere Fallstudie im Rahmen dieses Buches, als dass sie als Einführung in das Basishandwerkszeug mit R dient. Wenn die Zeit für nur eine Fallstudie ausreicht und man Anfänger mit der Datenanalyse mit R ist, dann ist diese Fallstudie die richtige. Sie steht deshalb auch am Anfang des Buches und dient in allen meinen Modulen zur Datenanalyse mit R als Gegenstand der Einführung.

Das Campusbier-Projekt beschäftigt sich mit der Vermarktung des hochschuleigenen Bieres, das 2009 von Studierenden auf der Versuchsbrauanlage der Hochschule Osnabrück entwickelt wurde. Im Jahr 2019 wurde das Bier erstmals in größerer Menge gebraut und in Flaschen abgefüllt, so dass es der Osnabrücker Bevölkerung zugänglich gemacht werden konnte. Zuvor war das Bier nur intern in Fässern erhältlich, die etwa für Veranstaltungen am Campus der Hochschule verwendet wurden. Die erste öffentliche Verkaufsrunde im Mai/Juni 2019 war ein voller Erfolg, was zu einer dauerhaften Verfügbarkeit des Bieres über den Onlie-Shop campusbier.de führte. Seitdem sind in vielen Projekten weitere Aktionen und Produkte hinzugekommen, deren Verkäufe sich im vorliegenden Datensatz wiederspiegeln.

Der Datensatz

Der Datensatz besteht aus den Informationen zu den knapp 3000 Bestellungen, die in den Jahren 2019 bis 2022 über den Online-Shop aufgegeben wurden. Neben den Metadaten einer Bestellung, wie der Bestellzeitpunkt, der Kunde oder die Zahlungsart, gibt es auch die Informationen über die gekauften Produkte, also den Warenkorb jeder Bestellung. Diese Daten liegen als CSV-Exporte aus dem E-Commerce-System Shopify vor. Der Datensatz besteht aus den beiden Dateien `orders.csv` und `line_items.csv`. Um dem Datenschutz Rechnung zu tragen, wurden sämtliche persönliche Daten der Kunden entfernt. Jeder Kunde ist nur über eine technische Nummer identifizierbar, die in jeder Bestellung angegeben ist. Haben mehrere Bestellungen die gleiche Kundennummer, so stammen diese alle vom selben Kunden.

Ladet euch am besten jetzt die beiden Dateien in euer Arbeitsverzeichnis herunter. Hier die Links zum Download (rechte Maustaste, “Link speichern unter”):

- orders.csv
- line_items.csv

1 CSV-Dateien einlesen

Im ersten Schritt jeder Datenanalyse müssen wir unserem Computer den Datensatz zur Verfügung stellen. Wir sprechen dabei auch vom *Laden* des Datesatzes. Dabei sagen wir dem Computer, wo die Daten zu finden sind und dass er sie für den schnelleren Zugriff in seinen Arbeitsspeicher holen soll.

Daten liegen in den meisten Fällen in Form von Dateien vor. In manchen Fällen sind sie auch in einer Datenbank gespeichert. Im Fall einer Datei kann ein Datensatz in unterschiedlichen *Formaten* darin gespeichert werden. Ein gängiges Format ist das CSV-Format, das auf einfachen Textdateien basiert.

1.1 Das CSV-Format

1.1.1 CSV als weit verbreitetes Format

Für die Speicherung von Daten bieten sich textuelle Formate an, weil sie auf jedem Betriebsystem mit einem einfachen Texteditor betrachtet und bearbeitet werden können. Das ermöglicht das einfache Teilen von Daten und somit die Zusammenarbeit. Auch für den Datenaustausch zwischen verschiedenen Informationssystemen wird häufig ein textbasiertes Format verwendet, um spezifisches Formate der jeweiligen Hersteller, wie etwa proprieäte Datenbanken, zu überbrücken. Deshalb bieten die meisten Informationssysteme Schnittstellen für den Export und Import von Textdateien an. Speziell das CSV-Format ist hier sehr beliebt, aus guten Gründen:

- Die Verwendung von einfachen Textdateien erlaubt die Speicherung und Verarbeitung auf unterschiedlichen Umgebungen wie Windows, macOS oder Linux.
- Das Format ist einfach zu verstehen und auch für Menschen lesbar.
- CSV ist ein offenes Format, d. h. es gibt keine Organisation, die daran die Rechte besitzt und es kann daher von jeder Software verwendet werden. Es gab lange nicht einmal eine offizielle Spezifikation des Formats. Mittlerweile gibt es eine Spezifikation als offizieller MIME Type.

Auch das E-Commerce-System Shopify, aus dem die vorligenden Verkaufsdaten stammen, bietet eine Möglichkeit zum Exportieren von Textdateien im sogenannten CSV-Format an.

1.1.2 CSV eignet sich für strukturierte Daten

CSV steht für *Comma Separated Values* und beschreibt ein Format, um *strukturierte Daten* in einer Textdatei abzuspeichern. Ihr erkennt eine Textdatei im CSV-Format an der Endung `.csv`.

Das CSV-Format basiert auf einfachen Textkodierungen, häufig im UTF-8 oder ASCII-Kodierungssystem (letzeres immer seltener wegen der geringen Anzahl verfügbarer Zeichen), die mit fast jedem Werkzeug und Editor gelesen und bearbeitet werden können. Zusätzlich bildet das CSV-Format eine tabellarische Struktur ab, bei dem die Daten in Zeilen und Spalten getrennt werden. Alle darauf folgenden Zeilen stellen Beobachtungen oder Datensätze dar, deren Variablenwerte mit dem gleichen Trennzeichen abgegrenzt werden.

Das CSV-Format speichert strukturierte Daten in einer tabellarischen Form, ähnlich wie in Spreadsheets. Die erste Zeile einer CSV-Datei ist üblicherweise der sogenannte Header (Kopfzeile) und beinhaltet die Spaltennamen mit Kommata oder Semikolon (Trennzeichen) voneinander getrennt. Jede weitere Zeile stellt eine Beobachtung (Englisch: *observation* oder *case*) oder auch Datensatz (Englisch: *record*) dar. Jeder Datensatz enthält für die im Header definierten Attribute (oder Spalten) einen Wert, die durch das gleiche Trennzeichen voneinander getrennt sind. Es muss nicht jeder Spaltenwert existieren. Sollte ein Wert für eine Beobachtung nicht vorhanden sein, so wird einfach nach dem Komma nichts eingetragen und es folgen zwei Kommata nacheinander. In R werden fehlende Werte mit `NA` gekennzeichnet.

Die Verwendung des Komma als Trennzeichen in CSV-Dateien ist keineswegs verbindlich, auch wenn es Bestandteil des Namens ist. Generell kann jedes Symbol verwendet werden. Häufige Alternativen sind das Semikolon, Leerzeichen oder ein Tabstop. Letzteres wird oft mit der eigenen Endung `.tsv` für *Tab Separated Values* gespeichert.

1.1.3 Beispiel einer CSV-Datei

Der Auschnitt unten zeigt die ersten vier Zeilen der `orders.csv`. Die erste Zeile enthält die Namen der hier gezeigten vier Spalten (der Datensatz hat mehr Spalten, das ist nur ein Auszug), die mit einem Komma voneinander getrennt sind. Darunter folgen drei beispielhafte Datensätze:

```
id,order_id,name,order_number,app_id,created_at
1130007101519,B1014,1014,580111,2019-05-24T14:59:16+02:00
1130014965839,B1015,1015,580111,2019-05-24T15:09:08+02:00
1130026958927,B1016,1016,580111,2019-05-24T15:22:41+02:00
...
```

1.2 CSV-Daten laden mit `{readr}`

1.2.1 Aus einer lokalen Datei

Für das Laden von Datensätzen aus CSV-Dateien bietet das Tidyverse ein Paket namens `{readr}` an. Dieses wird automatisch mit dem `{tidyverse}`-Paket mitgeladen. Das Paket bietet für CSV-Dateien, bei denen das Komma als Trennzeichen verwendet wird, die Funktion `read_csv` an:

```
orders <- read_csv("./data/orders.csv")
```

> In einigen Ländern Europas wird häufig ein Semikolon als Trennzeichen und ein Komma als Dezimaltrennzeichen verwendet (wie etwa in Deutschland). Für diesen Fall bietet `{readr}` die Funktion `read_csv2`. Um es selbst in der Hand zu haben, kann die Funktion `read_delim` verwendet werden und über den Parameter `delim` das Trennzeichen manuell eingestellt werden.

Auch der R-Basisumfang bietet eine ähnliche Funktion für genau diesen Anwendungsfall an. Diese heisst `read.csv`, man achte hier auf das Detail: Statt eines Unterstrichs `_` wird bei der R-Basisfunktion ein Punkt `.` zwischen den beiden Wörtern `read` und `csv` verwendet. Wenn ihr mit dem Tidyverse und mit Tibbles arbeitet (wie in diesem Buch durchgängig), dann achtet darauf, stets die `{readr}`-Funktion `read_csv` zu verwenden, weil nur diese die Daten als Tibble zurückgibt und zudem einige paar nützliche Zusatzfunktionen bietet.

1.2.2 Nur bestimmte Spalten laden

Die `read_csv`-Funktion erlaubt direkt beim Laden der Daten eine Auswahl der Spalten vorzunehmen. Ich empfehle hier immer die Verwendung der `select`-Funktion, die wir im nächsten Abschnitt kennenlernen werden. Dennoch möchte ich kurz demonstrieren, wie das Filtern der Spalten beim Laden funktioniert.

Der folgende Code lädt die Spalten `order_id`, `name`, sowie alle Spalten, deren Name mit "customer" beginnen:

```
orders <- read_csv("./data/orders.csv", col_select = c(order_id, name, starts_with("custom
```

Im Ergebnis ist der resultierende Tibble dann sehr viel schmaler und beinhaltet nur die gewünschten Spalten:

```
colnames(orders)
```

```
 [1] "order_id"
 [2] "name"
 [3] "customer_id"
 [4] "customer_accepts_marketing"
 [5] "customer_accepts_marketing_updated_at"
 [6] "customer_marketing_opt_in_level"
 [7] "customer_sms_marketing_consent"
 [8] "customer_created_at"
 [9] "customer_updated_at"
[10] "customer_gender"
[11] "customer_is_hsos"
[12] "customer_state"
[13] "customer_orders_count"
[14] "customer_total_spent"
[15] "customer_last_order_id"
[16] "customer_note"
[17] "customer_verified_email"
[18] "customer_tax_exempt"
[19] "customer_tags"
[20] "customer_last_order_name"
```

Hat man nur eine Filterbedingung, so kann man sich die `c()`-Funktion auch sparen:

```
orders <- read_csv("./data/orders.csv", col_select = starts_with("shipping"))
```

Das Ergebnis:

```
colnames(orders)
```

```
[1] "shipping_address_city"      "shipping_address_zip"
[3] "shipping_address_country"   "shipping_address_latitude"
[5] "shipping_address_longitude"
```

Ist man auf möglichst wenige Zeilen Code aus, so kann die Verwendung des `col_types`-Parameters durchaus Sinn ergeben. Man könnte den gleichen Effekt auch mit einem anschließenden `select` erzielen:

```
orders <- read_csv("./data/orders.csv") %>%
  select(starts_with("shipping"))
```

Mehr zur Auswahl von Spalten mit `{dplyr}` erfahrt ihr in Kapitel 3.

1.2.3 Das Dezimaltrennzeichen bestimmen

Wollen wir nur das Dezimaltrennzeichen ändern, weil wir zum Beispiel Daten aus einem deutschen System exportiert haben, das ein Komma verwendet, so können wir das über den `locale`-Parameter erreichen. Im Beispiel unten verwenden wir zusätzlich das Pipe-Symbol | als Trennzeichen. Beides können wir über `read_delim` einstellen:

```
# Daten in sales.csv
# year,month,turnover
# 2022|01|2700,85
# 2022|02|2910,10
# 2022|03|1802,37

turnover <- read_delim("sales.csv",
                       delim = "|",
                       locale = locale(decimal_mark = ",")
                       )
```

Das Ergebnis:

```
turnover
```

```
# A tibble: 3 x 3
   year month turnover
  <dbl> <chr>    <dbl>
1  2022 01       2701.
2  2022 02       2910.
3  2022 03       1802.
```

1.2.4 Datentypen beim Laden setzen

Beim Laden mit `read_csv` wird möglicherweise der Datentyp einiger Spalten falsch erkannt. DIe Funktion betrachtet die ersten Zeilen und ermittelt darauf basierend den Datentyp. Wird dieser falsch erkannt, so können den Datentyp für Spalten explizit angeben. Wie auch das Auswählen von Spalten können wir den Datentyp schon beim Laden mit. Alternativ können wir den Datentypen auch später mit `mutate` noch verändern.

> Manchmal hilft es schon, den Parameter `guess_max` auf einen größeren Wert zu setzen. Dieser bestimmt, wie viele Werte die Funktion betrachtet, um daraus den Datentyp einer Spalte abzuleiten. Standardmäßig ist dieser Wert auf 1000 begrenzt.

In unserem Datensatz `orders.csv` sind tatsächlich einige Datentypen falsch erkannt worden. Etwa die Spalte `order_id`, die als Datentyp `double` erkannt wird. Auch wenn die Werte allesamt als Dezimalzahlen betrachtet werden können, handelt es sich bei einer Bestellnummer nicht um eine Zahl in dem Sinne, dass man damit rechnen möchte. Es ist vielmehr eine Zeichenfolge, die per Konvention nur aus Zahlen besteht (aber nicht müsste). Der Datentyp `character` wäre somit angebrachter. Das können wir wie folgt ändern:

```
orders <- read_csv("./data/orders.csv",
                   col_types = list("order_id" = col_character()))

# Die Spalte order_id ist jetzt vom Typ <chr>
orders %>%
  select(order_id)
```

```
# A tibble: 2,874 x 1
   order_id
   <chr>
 1 1130007101519
 2 1130014965839
 3 1130026958927
 4 1130030563407
 5 1130038853711
 6 1130045964367
 7 1130050519119
 8 1130060283983
 9 1130102194255
10 1130106880079
# ... with 2,864 more rows
```

Das Gleiche können wir mit beliebig vielen Spalten gleichzeitig durchführen:

```
orders <- read_csv("./data/orders.csv",
                   col_types = list(
                     "order_id" = col_character(),
                     "app_id" = col_character()
                     )
                   )

# Die Spalte order_id ist jetzt vom Typ <chr>
orders %>%
  select(order_id, app_id)
```

```
# A tibble: 2,874 x 2
   order_id      app_id
   <chr>         <chr>
 1 1130007101519 580111
 2 1130014965839 580111
 3 1130026958927 580111
 4 1130030563407 580111
 5 1130038853711 580111
 6 1130045964367 580111
 7 1130050519119 580111
 8 1130060283983 580111
 9 1130102194255 580111
10 1130106880079 580111
# ... with 2,864 more rows
```

1.2.5 Daten von einem Webserver laden

Die CSV-Datei muss nicht lokal auf dem eigenen Rechner vorliegen, sondern kann mit `read_csv` über das HTTP-Protkoll direkt von einem Webserver im Internet abgerufen werden. Dabei wird die URL anstelle des lokalen Dateinamens der Funktion übergeben. Der Code unten lädt die tagesaktuelle Version des Covid-19-Datensatzes, der auf den Servern von Our World in Data gehostet wird:

```
covid <- read_csv("https://covid.ourworldindata.org/data/owid-covid-data.csv")
```

```
Rows: 220105 Columns: 67
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (4): iso_code, continent, location, tests_units
dbl  (62): total_cases, new_cases, new_cases_smoothed, total_deaths, new_dea...
date  (1): date

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Übungsaufgaben

1. Ladet den tagesaktuellen Datensatz für die Covid-Daten, lasst aber nur die Spalten `date`, `location` und `new_cases_smoothed_per_million` im Ergebnis!

2 Datensätze erkunden

2.1 Sprechende Tibbles

Alleine beim Aufruf seines Namens gibt ein Tibble in der Konsole viele seiner Informationen preis:

```
orders
```

```
# A tibble: 2,874 x 68
     order_id name  order~1 app_id created_at          updated_at          test
        <dbl> <chr>   <dbl>  <dbl> <dttm>              <dttm>              <lgl>
 1    1.13e12 B1014    1014 580111 2019-05-24 12:59:16 2019-06-19 13:23:26 FALSE
 2    1.13e12 B1015    1015 580111 2019-05-24 13:09:08 2019-06-21 14:40:07 FALSE
 3    1.13e12 B1016    1016 580111 2019-05-24 13:22:41 2019-06-21 12:35:23 FALSE
 4    1.13e12 B1017    1017 580111 2019-05-24 13:27:43 2019-06-21 14:27:18 FALSE
 5    1.13e12 B1018    1018 580111 2019-05-24 13:36:46 2019-06-21 12:11:57 FALSE
 6    1.13e12 B1019    1019 580111 2019-05-24 13:44:41 2019-06-21 14:37:21 FALSE
 7    1.13e12 B1020    1020 580111 2019-05-24 13:49:21 2019-06-21 12:25:16 FALSE
 8    1.13e12 B1021    1021 580111 2019-05-24 13:59:57 2019-06-21 11:49:47 FALSE
 9    1.13e12 B1022    1022 580111 2019-05-24 14:43:53 2019-06-19 14:12:38 FALSE
10    1.13e12 B1023    1023 580111 2019-05-24 14:48:16 2019-06-21 15:54:24 FALSE
# ... with 2,864 more rows, 61 more variables: current_subtotal_price <dbl>,
#   current_total_price <dbl>, current_total_discounts <dbl>,
#   current_total_duties_set <dbl>, total_discounts <dbl>,
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

Neben den ersten paar Zeilen als Vorschau gibt ein Tibble auch die Gesamtzahl an Zeilen und Spalten aus. Hier sind es 2874 Zeilen und 68 Spalten. Darunter folgt eine mit Kommata getrennte Auflistung der Spaltennamen und ihren Datentypen. Diese Liste wird aber nach wenigen Zeilen abgebrochen, um die Konsole nicht mit Text zu überladen.

> Versucht das einmal: Ladet die CSV-Datei statt mit `read_csv` mit der Funktion `read.csv` aus dem Basis-R. Gebt jetzt den Namen des Dataframes in die Konsole ein und drückt Enter. Was ist der Unterschied bei der Ausgabe? Was gefällt euch besser?

2.2 Anzahl Spalten und Zeilen ermitteln

Ein Tibble gibt es uns freiwillig Auskunft über seine Dimensionierung, wenn wir seinen Namen aufrufen. Wir können diese Werte auch dediziert ermitteln, falls wir mit diesen Information etwa weitere Berechnungen in unserem Skript durchführen wollen.

Mit `ncol` erhalten wir die Anzahl Spalten:

```
orders %>%
  ncol()
```

```
[1] 68
```

Mit `nrow` entsprechend die Anzahl Zeilen:

```
orders %>%
  nrow()
```

```
[1] 2874
```

Wir könnten mit beiden Werten die Anzahl Zellen errechnen:

```
cols <- ncol(orders)
rows <- nrow(orders)
cells <- cols * rows
cells
```

```
[1] 195432
```

Mit der Funktion `dim` erhalten wir einen Vektor mit beiden Informationen:

```
orders %>%
  dim()
```

```
[1] 2874   68
```

Der erste Wert steht für die Anzahl Zeilen, der zweite für die Spalten.

Wenn wir als Ergebnis statt eines Vektors lieber ein Tibble hätten, können wir uns für für die Anzahl Zeilen mit `count` helfen:

```
orders %>%
  count()
```

```
# A tibble: 1 x 1
      n
  <int>
1  2874
```

Wir können auch `nrow` und `ncol` als Spalten in einen neuen Dataframe (Tibble) verwenden und darauf basierend eine weitere, berechnete Spalte erstellen:

```
tibble(
  number_cols = ncol(orders),
  number_rows = nrow(orders),
  number_cells = number_cols * number_rows
  )
```

```
# A tibble: 1 x 3
  number_cols number_rows number_cells
        <int>       <int>        <int>
1          68        2874       195432
```

2.3 Datentypen

Die schnellste und, wie ich finde, einfachste Möglichkeit, einen Überblick über die Datentypen in einem Tibble zu erhalten, ist die `glimpse`-Funktion:

```
orders %>%
  glimpse()
```

```
Rows: 2,874
Columns: 68
$ order_id                                <dbl> 1.130007e+12, 1.130015e+12, 1.13~
$ name                                    <chr> "B1014", "B1015", "B1016", "B101~
$ order_number                            <dbl> 1014, 1015, 1016, 1017, 1018, 10~
$ app_id                                  <dbl> 580111, 580111, 580111, 580111, ~
$ created_at                              <dttm> 2019-05-24 12:59:16, 2019-05-24~
$ updated_at                              <dttm> 2019-06-19 13:23:26, 2019-06-21~
$ test                                    <lgl> FALSE, FALSE, FALSE, FALSE, FALS~
$ current_subtotal_price                  <dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
$ current_total_price                     <dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
$ current_total_discounts                 <dbl> 2, 0, 2, 0, 2, 2, 2, 0, 0, 0, 0,~
$ current_total_duties_set                <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ total_discounts                         <dbl> 2, 0, 2, 0, 2, 2, 2, 0, 0, 0, 0,~
$ total_line_items_price                  <dbl> 96.66, 32.22, 32.22, 32.22, 32.2~
$ total_outstanding                       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ total_price                             <dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
$ total_tax                               <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ total_tip_received                      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ taxes_included                          <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TR~
$ discount_codes                          <chr> "DCBPXGJB1JGA", NA, "KY0D5MNEZBM~
$ financial_status                        <chr> "paid", "paid", "paid", "paid", ~
$ fulfillment_status                      <chr> "fulfilled", "fulfilled", "fulfi~
$ source_name                             <chr> "web", "web", "web", "web", "web~
$ landing_site                            <chr> "/password", "/wallets/checkouts~
$ landing_site_ref                        <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ location_id                             <dbl> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ note                                    <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ tags                                    <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ processed_at                            <dttm> 2019-05-24 12:59:15, 2019-05-24~
$ processing_method                       <chr> "direct", "express", "express", ~
$ payment_details_gateway                 <chr> "shopify_payments", "paypal", "p~
$ payment_details_credit_card_company     <chr> "Mastercard", NA, NA, NA, NA, NA~
$ customer_id                             <dbl> 1.861678e+12, 1.856892e+12, 1.87~
$ customer_accepts_marketing              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
$ customer_accepts_marketing_updated_at   <dttm> 2019-05-21 13:38:54, 2019-05-24~
$ customer_marketing_opt_in_level         <chr> "single_opt_in", "single_opt_in"~
$ customer_sms_marketing_consent          <lgl> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ customer_created_at                     <dttm> 2019-05-13 14:41:16, 2019-05-10~
$ customer_updated_at                     <dttm> 2022-02-23 13:50:50, 2022-06-21~
$ customer_gender                         <chr> "f", "m", "m", "m", "f", "m", "m~
$ customer_is_hsos                        <dbl> 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,~
$ customer_state                          <chr> "disabled", "disabled", "enabled~
```

```
$ customer_orders_count                <dbl> 7, 10, 3, 1, 2, 1, 1, 1, 3, 6, 1~
$ customer_total_spent                 <dbl> 345.22, 338.40, 95.17, 32.22, 66~
$ customer_last_order_id               <dbl> 4.640512e+12, 5.053779e+12, 1.95~
$ customer_note                        <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ customer_verified_email              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
$ customer_tax_exempt                  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ customer_tags                        <chr> "password page, prospect, umfrag~
$ customer_last_order_name             <chr> "41005353622", "41005379922", "1~
$ campaign_tag                         <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ shipping_address_city                <chr> NA, "Osnabrück", NA, NA, NA, NA,~
$ shipping_address_zip                 <dbl> NA, 49088, NA, NA, NA, NA, 49090~
$ shipping_address_country             <chr> NA, "Germany", NA, NA, NA, NA, "~
$ shipping_address_latitude            <dbl> NA, 52.29756, NA, NA, NA, NA, 52~
$ shipping_address_longitude           <dbl> NA, 8.058480, NA, NA, NA, NA, 8.~
$ billing_address_city                 <chr> "Quakenbrück", "Osnabrück", "Osn~
$ billing_address_zip                  <dbl> 49610, 49088, 49080, 45888, 4935~
$ billing_address_country              <chr> "Germany", "Germany", "Germany",~
$ billing_address_company              <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ billing_address_latitude             <dbl> 52.67033, 52.29756, 52.26469, 51~
$ billing_address_longitude            <dbl> 7.959575, 8.058480, 8.035603, 7.~
$ client_details_browser_ip            <chr> "131.173.97.189", "131.173.96.36~
$ client_details_browser_height        <dbl> 665, 1010, 657, 600, 560, 553, 5~
$ client_details_browser_width         <dbl> 1263, 2048, 1349, 360, 360, 375,~
$ client_details_user_agent            <chr> "Mozilla/5.0 (Windows NT 10.0; W~
$ cancel_reason                        <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ cancelled_at                         <dttm> NA, NA, NA, NA, NA, NA, NA, NA,~
$ closed_at                            <dttm> 2019-06-19 13:23:26, 2019-06-21~
```

Neben dem Namen der Spalten erhalten wir den Datentyp sowie eine Vorschau der ersten in jeder Spalte enthaltenen Werte. Auch die Anzahl Zeilen und Spalten nennt uns `glimpse`.

Etwas technischer geht es mit der `spec`-Funktion. Diese liefert statt des Datentyp-Kürzels auch gleich die genaue Spezfikation als R-Funktion zurück:

```
spec(orders)
```

```
cols(
  order_id = col_double(),
  name = col_character(),
  order_number = col_double(),
  app_id = col_double(),
  created_at = col_datetime(format = ""),
```

```
updated_at = col_datetime(format = ""),
test = col_logical(),
current_subtotal_price = col_double(),
current_total_price = col_double(),
current_total_discounts = col_double(),
current_total_duties_set = col_double(),
total_discounts = col_double(),
total_line_items_price = col_double(),
total_outstanding = col_double(),
total_price = col_double(),
total_tax = col_double(),
total_tip_received = col_double(),
taxes_included = col_logical(),
discount_codes = col_character(),
financial_status = col_character(),
fulfillment_status = col_character(),
source_name = col_character(),
landing_site = col_character(),
landing_site_ref = col_character(),
location_id = col_double(),
note = col_character(),
tags = col_character(),
processed_at = col_datetime(format = ""),
processing_method = col_character(),
payment_details_gateway = col_character(),
payment_details_credit_card_company = col_character(),
customer_id = col_double(),
customer_accepts_marketing = col_double(),
customer_accepts_marketing_updated_at = col_datetime(format = ""),
customer_marketing_opt_in_level = col_character(),
customer_sms_marketing_consent = col_logical(),
customer_created_at = col_datetime(format = ""),
customer_updated_at = col_datetime(format = ""),
customer_gender = col_character(),
customer_is_hsos = col_double(),
customer_state = col_character(),
customer_orders_count = col_double(),
customer_total_spent = col_double(),
customer_last_order_id = col_double(),
customer_note = col_character(),
customer_verified_email = col_double(),
customer_tax_exempt = col_double(),
customer_tags = col_character(),
```

```
  customer_last_order_name = col_character(),
  campaign_tag = col_character(),
  shipping_address_city = col_character(),
  shipping_address_zip = col_double(),
  shipping_address_country = col_character(),
  shipping_address_latitude = col_double(),
  shipping_address_longitude = col_double(),
  billing_address_city = col_character(),
  billing_address_zip = col_double(),
  billing_address_country = col_character(),
  billing_address_company = col_character(),
  billing_address_latitude = col_double(),
  billing_address_longitude = col_double(),
  client_details_browser_ip = col_character(),
  client_details_browser_height = col_double(),
  client_details_browser_width = col_double(),
  client_details_user_agent = col_character(),
  cancel_reason = col_character(),
  cancelled_at = col_datetime(format = ""),
  closed_at = col_datetime(format = "")
)
```

Fällt uns auf, dass ein Datentyp falsch erkannt wurde, so können wir entweder unseren Ladevorgang anpassen und etwa den Datentyp direkt beim Lesen korrigieren. Alternativ verändern wir die Spalte mit `mutate`, was wir in Kapitel 4 lernen werden.

2.4 Spaltennamen

Ist man noch nicht lange mit einem Datensatz vertraut, ist eine Funktion für die Auflistung der Spaltennamen hilfreich:

```
orders %>%
  colnames()
```

```
[1] "order_id"
[2] "name"
[3] "order_number"
[4] "app_id"
[5] "created_at"
[6] "updated_at"
```

```
 [7] "test"
 [8] "current_subtotal_price"
 [9] "current_total_price"
[10] "current_total_discounts"
[11] "current_total_duties_set"
[12] "total_discounts"
[13] "total_line_items_price"
[14] "total_outstanding"
[15] "total_price"
[16] "total_tax"
[17] "total_tip_received"
[18] "taxes_included"
[19] "discount_codes"
[20] "financial_status"
[21] "fulfillment_status"
[22] "source_name"
[23] "landing_site"
[24] "landing_site_ref"
[25] "location_id"
[26] "note"
[27] "tags"
[28] "processed_at"
[29] "processing_method"
[30] "payment_details_gateway"
[31] "payment_details_credit_card_company"
[32] "customer_id"
[33] "customer_accepts_marketing"
[34] "customer_accepts_marketing_updated_at"
[35] "customer_marketing_opt_in_level"
[36] "customer_sms_marketing_consent"
[37] "customer_created_at"
[38] "customer_updated_at"
[39] "customer_gender"
[40] "customer_is_hsos"
[41] "customer_state"
[42] "customer_orders_count"
[43] "customer_total_spent"
[44] "customer_last_order_id"
[45] "customer_note"
[46] "customer_verified_email"
[47] "customer_tax_exempt"
[48] "customer_tags"
[49] "customer_last_order_name"
```

```
[50] "campaign_tag"
[51] "shipping_address_city"
[52] "shipping_address_zip"
[53] "shipping_address_country"
[54] "shipping_address_latitude"
[55] "shipping_address_longitude"
[56] "billing_address_city"
[57] "billing_address_zip"
[58] "billing_address_country"
[59] "billing_address_company"
[60] "billing_address_latitude"
[61] "billing_address_longitude"
[62] "client_details_browser_ip"
[63] "client_details_browser_height"
[64] "client_details_browser_width"
[65] "client_details_user_agent"
[66] "cancel_reason"
[67] "cancelled_at"
[68] "closed_at"
```

Diese Information liefert einem auch `glimpse`, wie oben gezeigt. Mit `colnames` bekommen wir die Spaltennamen als Vektor, was nützlich sein kann, um damit weiter arbeiten zu können.

2.5 Stichproben betrachten

Für einen ersten Eindruck der Daten reicht es oft schon, sich ein paar Zeilen anzeigen zu lassen. Das geht mit unterschiedlichen Funktionen:

```r
# Zeigt standardmäßig die ersten 6 Zeilen an
orders %>%
  head()

# Zeigt die ersten 20 Zeilen an
orders %>%
  head(20)

# Wenn man mehr anzeigen möchte, kann print verwendet werden
orders %>%
  print(n = 100)

# Das Ganze geht auch von unten
```

```
orders %>%
  tail()
```

Möchte man nicht die oberen oder unteren Zeilen anzeigen, sondern eine zufällige Auswahl, so hilft `sample_n`:

```
orders %>%
  sample_n(size = 5)
```

```
# A tibble: 5 x 68
     order_id name  order~1 app_id created_at          updated_at          test
        <dbl> <chr>   <dbl>  <dbl> <dttm>              <dttm>              <lgl>
1     1.89e12 1681~    1435 580111 2019-11-20 12:51:38 2019-12-13 14:17:37 FALSE
2     3.76e12 4100~    2564 580111 2021-04-20 16:20:54 2021-05-07 15:46:55 FALSE
3     3.14e12 4100~    2397 580111 2020-12-16 19:25:09 2020-12-20 15:16:55 FALSE
4     3.10e12 4100~    2071 580111 2020-11-28 18:19:17 2020-12-07 15:16:30 FALSE
5     2.37e12 4100~    1815 580111 2020-06-16 10:45:30 2020-06-22 09:18:58 FALSE
# ... with 61 more variables: current_subtotal_price <dbl>,
#   current_total_price <dbl>, current_total_discounts <dbl>,
#   current_total_duties_set <dbl>, total_discounts <dbl>,
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

Um ein bestimmten prozentualen Anteil an den gesamten Zeilen zufällig zu ermitteln, gibt es `sample_frac`:

```
# Gibt zufällig gewählte Zeilen zurück, die 1% entsprechen
orders %>%
  sample_frac(size = 0.01)
```

```
# A tibble: 29 x 68
     order_id name  order~1 app_id created_at          updated_at          test
        <dbl> <chr>   <dbl>  <dbl> <dttm>              <dttm>              <lgl>
 1    4.64e12 4100~    3542 580111 2022-01-17 09:17:24 2022-01-27 08:58:28 FALSE
 2    4.60e12 4100~    3263 580111 2021-12-07 09:58:40 2021-12-20 14:28:19 FALSE
 3    4.12e12 4100~    2662 580111 2021-09-09 17:50:33 2021-09-14 11:40:16 FALSE
 4    4.58e12 4100~    3148 580111 2021-11-18 13:24:04 2021-11-22 15:12:50 FALSE
 5    4.11e12 4100~    2656 580111 2021-09-07 11:57:06 2021-09-14 12:01:04 FALSE
```

```
 6    3.26e12 4100~    2455 580111 2021-02-22 15:54:17 2021-03-01 11:07:19 FALSE
 7    4.61e12 4100~    3358 580111 2021-12-11 18:05:25 2021-12-20 13:46:17 FALSE
 8    3.12e12 4100~    2262 580111 2020-12-05 14:09:33 2021-03-01 09:16:02 FALSE
 9    1.14e12 B1123    1123 580111 2019-05-28 13:38:04 2019-06-21 14:42:40 FALSE
10    2.28e12 4100~    1645 580111 2020-05-14 11:22:42 2020-05-16 14:22:56 FALSE
# ... with 19 more rows, 61 more variables: current_subtotal_price <dbl>,
#   current_total_price <dbl>, current_total_discounts <dbl>,
#   current_total_duties_set <dbl>, total_discounts <dbl>,
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

2.6 Häufigkeiten erfassen

Wir haben oben bereits `count` für das Zählen der Zeilen kennengelernt. Die Funktion eignet sich auch für das Zählen von Zeilen gruppiert nach einem Merkmal in den Daten:

```
orders %>%
  count(payment_details_gateway)
```

```
# A tibble: 4 x 2
  payment_details_gateway     n
  <chr>                   <int>
1 manual                    194
2 paypal                   1790
3 shopify_payments          889
4 <NA>                        1
```

Das geht auch sortiert nach Häufigkeit:

```
orders %>%
  count(payment_details_gateway, sort = TRUE)
```

```
# A tibble: 4 x 2
  payment_details_gateway     n
  <chr>                   <int>
1 paypal                   1790
2 shopify_payments          889
```

```
3 manual                     194
4 <NA>                         1
```

Mit dem bereits bekannten `{janitor}`-Paket erhalten wir eine Funktion, um für nominal skalierte Merkmale schnell die Häufigkeiten, sowohl absolut als auch prozentual, zu ermitteln:

```
library(janitor)

orders %>%
  tabyl(payment_details_gateway)
```

```
payment_details_gateway    n      percent valid_percent
                 manual  194 0.0675017397    0.06752523
                 paypal 1790 0.6228253305    0.62304212
       shopify_payments  889 0.3093249826    0.30943265
                   <NA>    1 0.0003479471            NA
```

Hier müssen wir das Sortieren manuelle mit `arrange` vornehmen:

```
orders %>%
  tabyl(payment_details_gateway) %>%
  arrange(-n)
```

```
payment_details_gateway    n      percent valid_percent
                 paypal 1790 0.6228253305    0.62304212
       shopify_payments  889 0.3093249826    0.30943265
                 manual  194 0.0675017397    0.06752523
                   <NA>    1 0.0003479471            NA
```

Wenn wir eine zweite Variable hinzufügen, so erstellt `tabyl` eine Kreuztabelle mit den absoluten Häufigkeiten der jeweiligen Kombinationen:

```
orders %>%
  tabyl(payment_details_gateway, payment_details_credit_card_company)
```

```
payment_details_gateway American Express Mastercard Visa  NA_
                 manual                0          0    0  194
                 paypal                0          1    3 1786
       shopify_payments               14        372  303  200
                   <NA>                0          0    0    1
```

Wir können so erkennen, dass für PayPal-Zahlungen nur sehr selten ein Kreditkartenanbieter hinterlegt ist. Bei den Shopify-Payments hingegen ist das in den meisten Bestellungen der Fall. Dabei liegt Mastercard knapp vor Visa, American Express ist eher die Ausnahme.

> Es wäre doch interessant zu wissen, warum genau eine Bestellung keine Angabe zur Zahlungsart besitzt. Prüft doch mal nach, welche das ist und versucht die Frage zu beantworten.

Übungsaufgaben

Speichert die Lösungen für die folgenden Aufgaben in einem neuen R-Skript. Vergesst nicht, die Datei zwischendurch zu speichern.

1. **Dimensionen ermitteln**

   - Ermittelt die Anzahl Spalten und Zeilen des Datensatzes. Ihr werdet herausfinden, dass es dafür verschiedene Möglichkeiten gibt.
   - Findet auch eine Möglichkeit, mit der ihr beide Größen auf jeweils einer Variable speichern könnt. Das kann zum Beispiel wichtig sein, wenn ihr mit den Größen im weiteren Skriptverlauf weiter arbeiten wollt.

2. **Nützliche Funktionen**

   - Recherchiert und testet die folgenden Funktionen. Notiert euch in eigenen Worten, was jede Funktion zurückgibt:
     - `head()`
     - `tail()`
     - `print()`
     - `sample_n()`
     - `glimpse()`
     - `summary()`
     - `colnames()`
   - Welche Parameter könnt ihr den Funktionen jeweils übergeben und wozu dienen sie?
   - Wozu würdet ihr die Funktionen verwenden?

3. **Datentypen ermitteln**

- Findet einen Weg, um den Datentyp für jede Spalte auf der Konsole auszugeben!
- Welche Datentypen sind vertreten?
- Sind eurer Meinung nach alle Spalten korrekt erkannt worden?

4. **Das Paket `{skimr}`**

   - Installiert das Paket `{skimr}` und betrachtet nun die Funktion `skim()`.
   - Was ermittelt die Funktion, wenn wir ihr einen Datensatz übergeben?
   - Gebt mithilfe der Funktion nur den prozentualen Anteil fehlender Werte (`NA`) für jede Spalte auf der Konsole aus!

3 Spalten auswählen

4 Spalten hinzufügen und verändern

Quellen