# LV: Analysing Quality of Care data - resources

Catherine Blatter

2023-04-11

# Table of contents

# Preface

This web-resource contains code that was talked about upon during class.

## Necessary packages

If you do not have these installed ("Could not find package xyz") you need to install the package.

```r
# load necessary packages
# data preparation
library(readr)
library(tibble)
library(dplyr)

# visualization
library(ggplot2)

# missings
library(naniar)

# tables
library(kableExtra)
```

## Import dataset

- By storing it into an object with the `zz_`-prefix, you can easier modify it afterwards without re-importing

```r
# import the dataset using the {readr}-package
zz_dataset <- read_csv("data/2023-02-20_QoC_sample_dataset.csv",
                       show_col_types = FALSE)
```

## Check the import

By looking at the top five rows (the “head” of the dataset):

```
head(zz_dataset, n = 5)
```

```
# A tibble: 5 x 6
  patient_ids sex   unit   adm          los disch     
        <dbl> <chr> <chr>  <date>     <dbl> <date>    
1       13679 f     ward_B 2022-01-27    10 2022-02-06
2       15231 f     ward_A 2022-01-16     3 2022-01-19
3       10758 f     ward_A 2022-01-10    18 2022-01-28
4       10319 m     ward_B 2022-01-20     7 2022-01-27
5       17826 m     ward_A 2022-01-26     5 2022-01-31
```

- Seems the input worked as expected :relieved:
- the columns ‘sex’ and ‘unit’ should be a factor, not a character

```
# change 'sex' into factor
dataset <-
  zz_dataset %>%
  mutate(sex = as.factor(sex),
         unit = as.factor(unit))
```

… then again check the top five rows:

```
head(dataset, n = 5)
```

```
# A tibble: 5 x 6
  patient_ids sex   unit   adm          los disch     
        <dbl> <fct> <fct>  <date>     <dbl> <date>    
1       13679 f     ward_B 2022-01-27    10 2022-02-06
2       15231 f     ward_A 2022-01-16     3 2022-01-19
3       10758 f     ward_A 2022-01-10    18 2022-01-28
4       10319 m     ward_B 2022-01-20     7 2022-01-27
5       17826 m     ward_A 2022-01-26     5 2022-01-31
```

# 1 Duplicates

## 1.1 Different ‘types’ of duplicates

- fully duplicated entries are ‘easy’ to catch but rare
- partially duplicated or mismatched entries are frequent but require more sophisticated checking

### 1.1.1 Fully duplicated entries

- `base::duplicated()` tests for fully duplicated entries (same row twice)
- several options to use:

```r
# check numbers of duplicated rows
table(duplicated(dataset))
```

```

FALSE  TRUE 
   99     1 
```

- add a column `is_dupe` with information:

```r
# adding column
dataset$is_dupe <- duplicated(dataset)

# check entry
head(dataset)
```

```
# A tibble: 6 x 7
  patient_ids sex   unit   adm          los disch      is_dupe
        <dbl> <fct> <chr>  <date>     <dbl> <date>     <lgl>  
1       13679 f     ward_B 2022-01-27    10 2022-02-06 FALSE  
2       15231 f     ward_A 2022-01-16     3 2022-01-19 FALSE  
```

```
3       10758 f     ward_A 2022-01-10    18 2022-01-28 FALSE
4       10319 m     ward_B 2022-01-20     7 2022-01-27 FALSE
5       17826 m     ward_A 2022-01-26     5 2022-01-31 FALSE
6       17463 f     ward_A 2022-01-30    33 2022-03-04 FALSE
```

### 1.1.2 Fully distinct entries

- `dplyr::distinct()` tests for full distinct entries (same row twice)
- basically the opposite of `base::duplicated()`

### 1.1.3 Identify duplicates by counting the key (or key-index-pair)

```
# counting the number of rows per patient_ids
# using sort = T moves the highest number on top
count(dataset, patient_ids, sort = T)
```

```
# A tibble: 96 x 2
   patient_ids     n
         <dbl> <int>
 1       10119     2
 2       12335     2
 3       16839     2
 4       18098     2
 5       10236     1
 6       10319     1
 7       10522     1
 8       10647     1
 9       10758     1
10       10875     1
# ... with 86 more rows
```

- there are several duplicated rows - although `duplicated()` only identified one fully duplicated entry
- this requires a further examination of these cases by identfiying the “weird” ids and in a second create a subset (filtered) dataset, to check these cases:

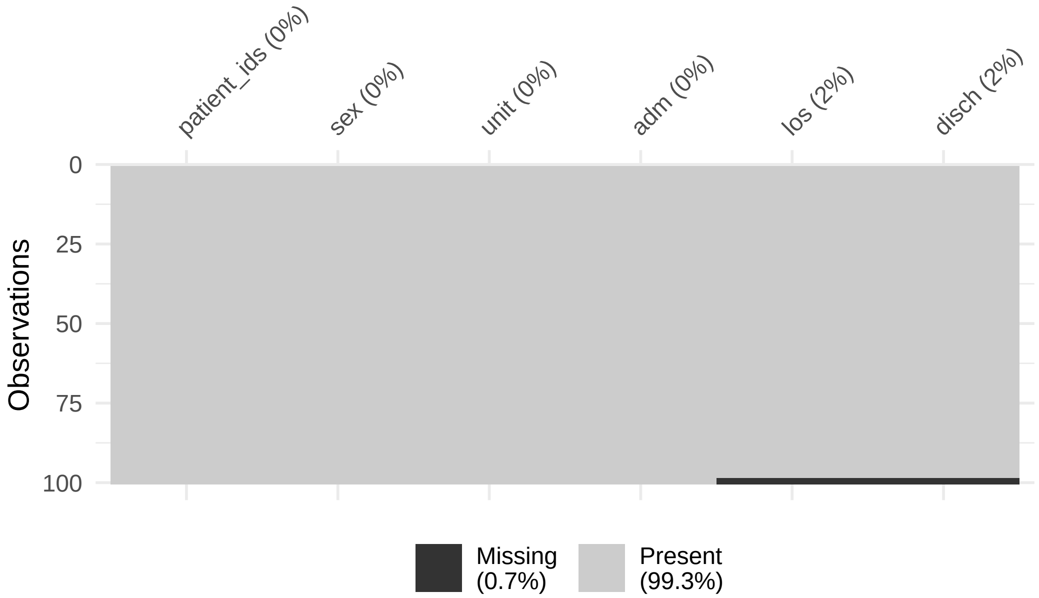
```
# keep the weird ids
weird_ids <-
  count(dataset, patient_ids, sort = T) %>%
  filter(n > 1 ) %>%
  pull(patient_ids)

# filter the dataset by these values
dataset %>%
  filter(patient_ids %in% weird_ids) %>%
  arrange(patient_ids)
```

```
# A tibble: 8 x 7
  patient_ids sex   unit   adm          los disch      is_dupe
        <dbl> <fct> <chr>  <date>     <dbl> <date>     <lgl>
1       10119 f     ward_A 2022-01-08    27 2022-02-04 FALSE
2       10119 f     ward_A 2022-01-08    27 2022-02-04 TRUE
3       12335 m     ward_B 2022-01-15    21 2022-02-05 FALSE
4       12335 m     ward_B 2022-01-25    25 2022-02-19 FALSE
5       16839 m     ward_A 2022-01-08    16 2022-01-24 FALSE
6       16839 f     ward_B 2022-01-08    NA NA         FALSE
7       18098 f     ward_B 2022-01-13    16 2022-01-29 FALSE
8       18098 f     ward_B 2022-01-13    NA NA         FALSE
```

# 2 Missings

## 2.1 Explore missings in sample data

```
naniar::vis_miss(dataset)
```



- There are only few missings
- NA's in `disch` are linked to NA's in `los`

## 2.2 Dataframe from slides

- by using the command `set.seed(1234)` you should be able to reproduce the same dataframe as in the slides

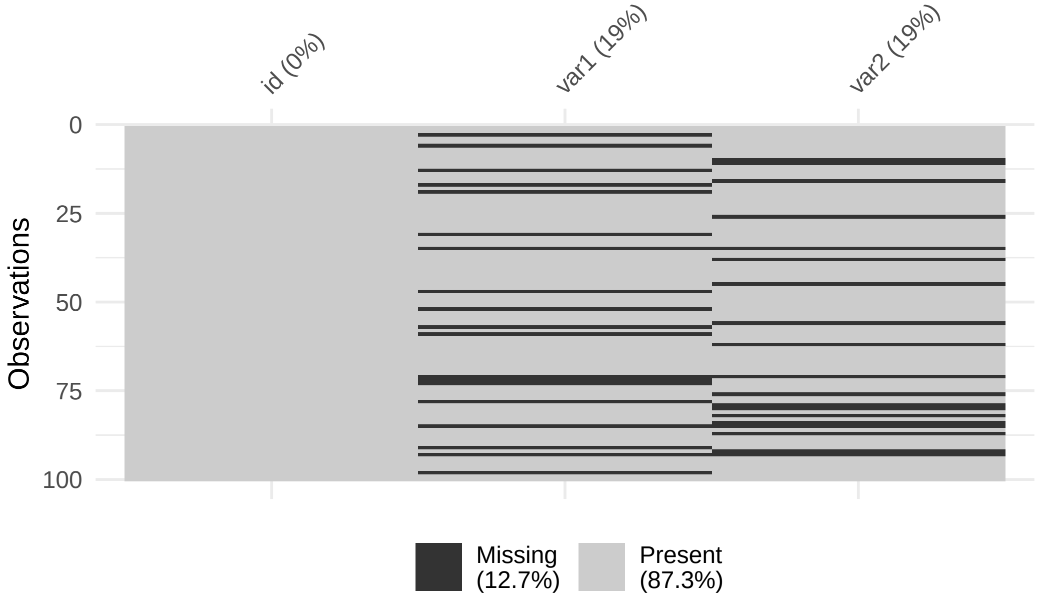
```
# set seed for reproducibility
set.seed(1234)

# create a sample-df with 100 rows and
# 3 variables
df_miss <-
  tibble(id = 1:100,
         var1 = sample(
           c(1:4, NA_real_), 100, T),
         var2 = sample(
           c(1:4, NA_real_), 100, T))

# first 4 entries
head(df_miss, 4)
```

```
# A tibble: 4 x 3
     id  var1  var2
  <int> <dbl> <dbl>
1     1     4     3
2     2     2     1
3     3    NA     3
4     4     4     4
```

```
# quick summary of proportional missings
naniar::miss_prop_summary(df_miss)
```

```
# A tibble: 1 x 3
     df   var  case
  <dbl> <dbl> <dbl>
1 0.127 0.667  0.34
```

```
naniar::vis_miss(df_miss)
```


id (0%)
var1 (19%)
var2 (19%)
0
25
50
75
100
Observations
Missing (12.7%)
Present (87.3%)

# 3 Visualising data

## 3.1 Prepare data for visualisation

Consider the example from class where four people (A, B, C, D) were asked about their tool-preferences for accessing their e-banking account:

- Formally, this dataframe is in 'tidy' format and the unit of analysis is the `person_id`

Now we represent the same information in a barchart, to compare the frequencies:

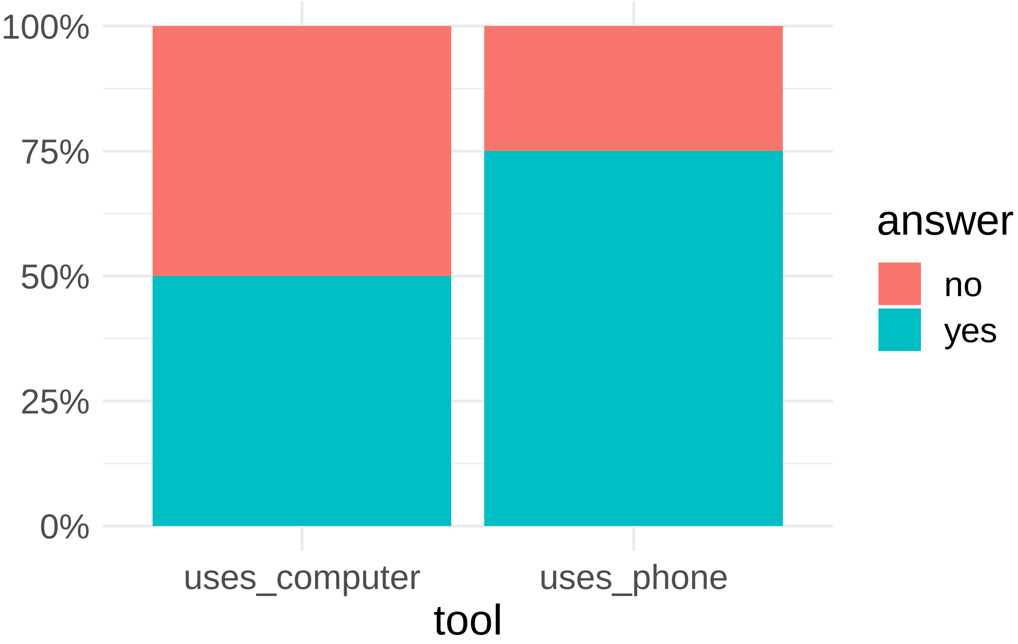
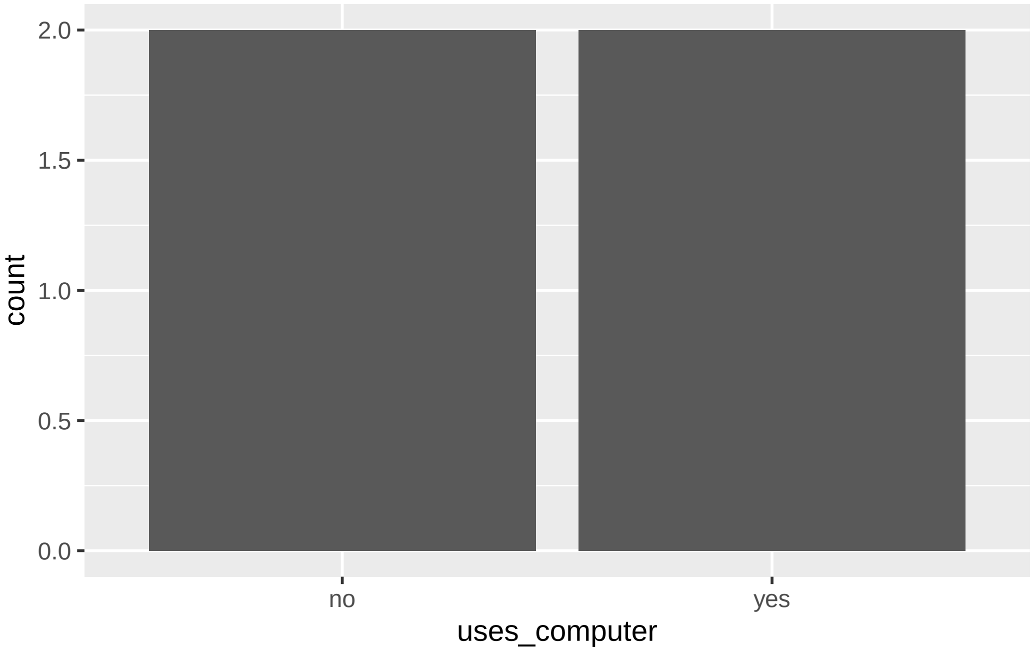


Table 3.1: Tidy dataframe - wide format

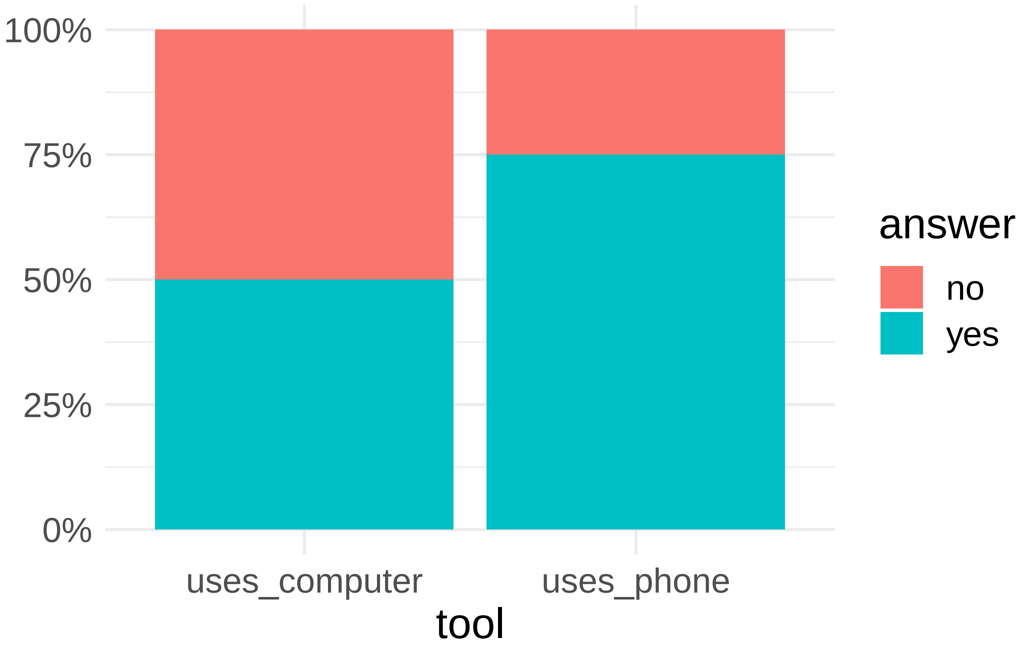
| person_id | uses_phone | uses_computer |
|---|---|---|
| A | yes | no |
| B | yes | yes |
| C | yes | no |
| D | no | yes |

But how did we actually input the data? If we try this we only get the variable `uses_computer` on the x-axis (with the values no and yes):

```
ggplot(data = wldf, aes(x = uses_computer)) +
  geom_bar()
```



When we go back to the original plot:



- we see that we want a variable `tool` on the x-axis, that has the values ‘uses_computer’ and ‘uses_phone’
- we want a variable `answer` that has the values ‘no’ and ‘yes’

(- we possibly want a variable `percentage` on the y-axis, that has numeric values according to the proportions)

- In summary, the format of the data needs to change in order to plot

## 3.2 Option 1: wide-to-long a.k.a stack

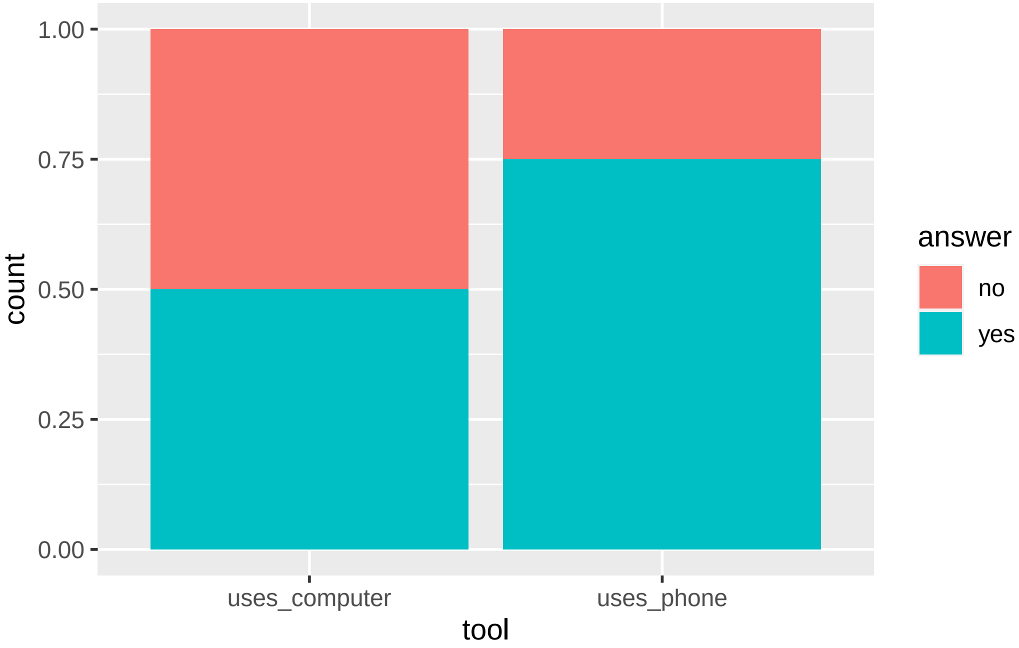
In this option we use `tidyr::pivot_longer()` to create a dataframe in long format:

```
# we want to stack the columns uses_computer and uses_phone
long_df <-
  wldf %>%
  pivot_longer(cols = c(uses_computer, uses_phone),
               names_to = "tool",
               values_to = "answer")

# print the df
long_df
```

```
# A tibble: 8 x 3
  person_id tool         answer
  <chr>     <chr>        <chr>
1 A         uses_computer no
2 A         uses_phone    yes
3 B         uses_computer yes
4 B         uses_phone    yes
5 C         uses_computer no
6 C         uses_phone    yes
7 D         uses_computer yes
8 D         uses_phone    no
```

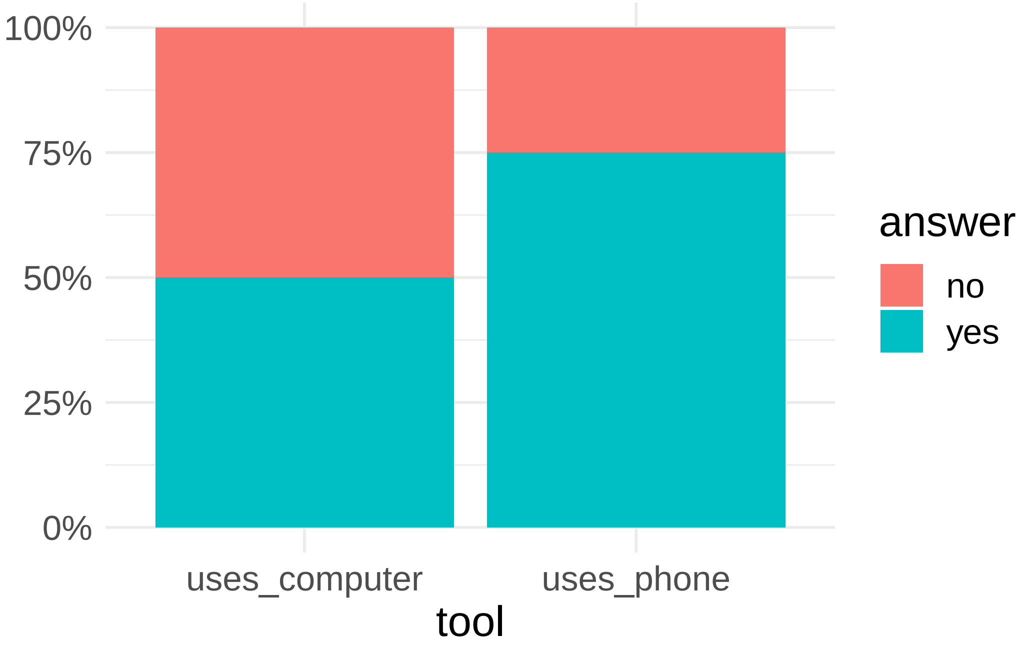
- this dataframe now has the variable `tool` and `answer` for the ggplot2-code using `geom_bar(position = "fill")`:

```
ggplot(long_df, aes(x = tool, fill = answer)) +
  geom_bar(position = "fill")
```



- of course, this needed some further 'layout-work' to improve the labeling of scales and the removal of the grey-background

```
ggplot(long_df, aes(x = tool, fill = answer)) +
  geom_bar(position = "fill") +
  scale_y_continuous(name = NULL,
                     labels = scales::percent) +
  theme_minimal() +
  theme(text = element_text(size = 16))
```



## 3.3 Option 2: Aggregation of values before plotting

Another option is to aggregate (or summarise) values manually, before we pass them to a plotting function (the example above is not really a good example for this), but the code might be handy if you have more variables than 2 for this situation.

In the example, we'd need to have three variables as indicated above: `tool` (for the x-axis), `answer` for the fill and `percent` for the percentage. We can create this table with:

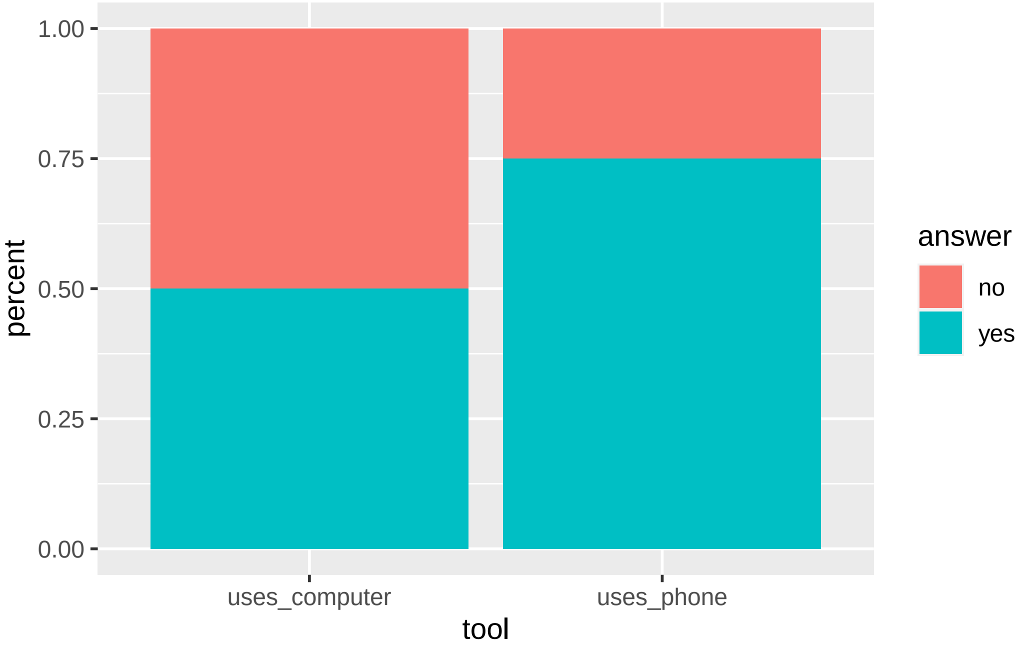
```
agg_data <-
  wldf |>
  select(uses_computer, uses_phone) |>
  as.list() |>
  map_dfr(~janitor::tabyl(.x), .id = "tool") |>
  rename(answer = .x)
```

```
# print agg_data
agg_data
```

```
         tool answer n percent
uses_computer     no 2    0.50
uses_computer    yes 2    0.50
   uses_phone     no 1    0.25
   uses_phone    yes 3    0.75
```

And then we can plot it:

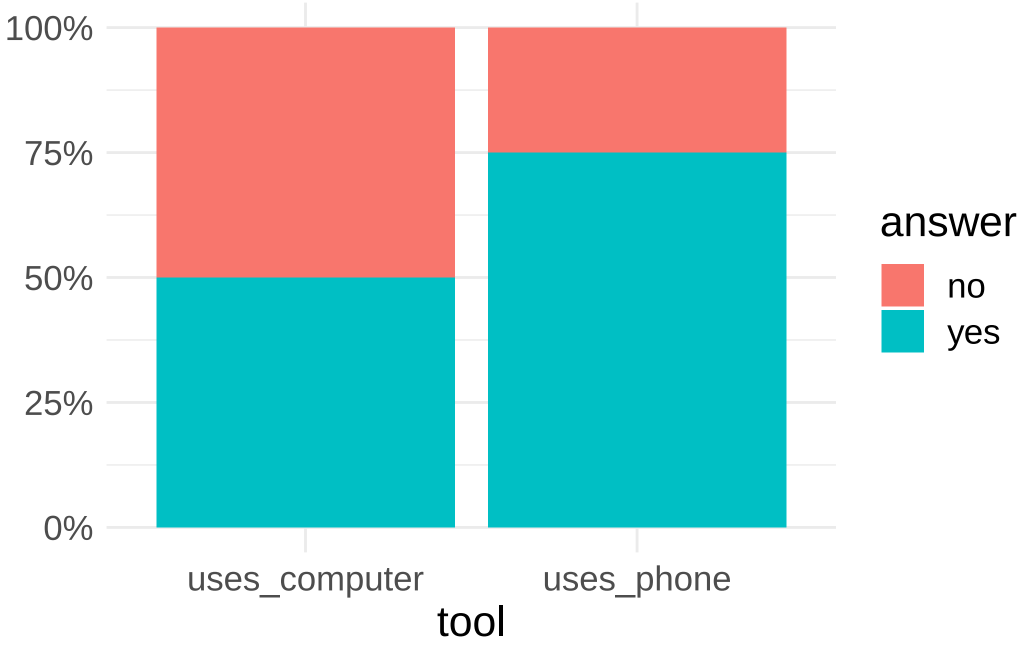
```
ggplot(data = agg_data, aes(x = tool, y = percent, fill = answer)) +
  geom_col(position = "stack")
```



- Careful: instead of using `geom_bar(position="fill")` without a y-variable, here we pass the `percent`-value to the y-axis and use `geom_col(position="stack")` for the `answer`-variable

An finally, improve the layout:

```
ggplot(data = agg_data, aes(x = tool, y = percent, fill = answer)) +
  geom_col(position = "stack") +
  scale_y_continuous(name = NULL,
                     labels = scales::percent) +
  theme_minimal() +
  theme(text = element_text(size = 16))
```


uses_computer
uses_phone
tool
answer
🟥 no
🟦 yes