

Lab 1: Intro to R and data analysis

Practice session covering topics
discussed in Lecture 1

M. Chiara Mimmi, Ph.D. | Università degli Studi di Pavia

July 25, 2024

Lecture 1: topics

- Introduction to R and R-studio
 - Why R?
 - Principles of reproducible analysis with R + RStudio
- R objects, functions, packages
- Understanding different types of variables
 - Principles of “tidy data”
- Descriptive statistics
 - Measures of central tendency, measures of variability (or spread), and frequency distribution
- Visual data exploration
 - `{ggplot2}`

INTRO TO R AND RSTUDIO

R version

If you have previously installed R on your machine, you can check which version you are running by executing this command in R:

```
1 # check your R version  
2 R.Version()
```

```
$platform  
[1] "x86_64-apple-darwin17.0"  
  
$arch  
[1] "x86_64"  
  
$os  
[1] "darwin17.0"  
  
$system  
[1] "x86_64, darwin17.0"  
  
$status  
[1] ""  
  
$major  
[1] "4"  
  
$minor  
[1] "2.2"  
  
1 # or just  
2 #R.version.string
```

Install R

R is available for free for Windows 📺, GNU/Linux 🐧, and macOS 🍎.

- To install R, go to this link <https://cloud.r-project.org/>. The latest available release is R 4.3.3 “Angel Food Cake” released on 2024-02/29, but any (fairly recent) version will do.

Install RStudio IDE

RStudio Desktop is an Integrated Development Editor (IDE), basically a graphical interface wrapping and interfacing R (which needs to be installed first).

Besides RStudio, R (which is a command line driven program) can be executed:

- via its native interface (**R GUI**)
- from many other code editors, like **VS Code, Sublime Text, Jupyter Notebook**

To install **RStudio**, go to this link <https://posit.co/download/rstudio-desktop/>. The free-version contains everything you need.

Use RStudio IDE

The screenshot illustrates the RStudio IDE interface with several labeled components:

- Source**: The leftmost pane, highlighted with a red border, contains the R script code for generating a plot.
- Console**: The bottom-left pane, also highlighted with a red border, shows the R command-line interface output.
- Output**: The rightmost pane, highlighted with a red border, displays the resulting ggplot2 scatter plot.
- Environment**: A top-level tab in the header bar.
- History**: A top-level tab in the header bar.
- Connections**: A top-level tab in the header bar.
- Build**: A top-level tab in the header bar.
- Git**: A top-level tab in the header bar.
- Global Environment**: A dropdown menu under the Environment tab.
- Imports**: A dropdown menu under the Environment tab.
- Grid**: A dropdown menu under the Environment tab.
- rstudio-user-guide**: The title bar of the RStudio window.

Source Pane Content:

```
1 library(ggplot2)
2 mpg_plot <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(colour = class))
3
4
5 mpg_plot
6
```

Console Pane Content:

```
R 4.2.0 · ~/rstudio-user-guide/
> library(ggplot2)
> mpg_plot <- ggplot(mpg, aes(x = displ, y = hwy)) +
+   geom_point(aes(colour = class))
>
> mpg_plot
>
```

Output Pane Content:

A scatter plot showing the relationship between engine displacement (displ) on the x-axis and highway fuel economy (hwy) on the y-axis. The plot uses color to represent vehicle classes. The legend (class) includes:

- 2seater (red)
- compact (yellow)
- midsize (green)
- minivan (teal)
- pickup (blue)
- subcompact (purple)
- suv (pink)

The x-axis (displ) ranges from approximately 1.6 to 7.0 liters. The y-axis (hwy) ranges from approximately 12 to 44 mpg.

RStudio Pane Layout Source: Posit's RStudio User Guide

Creating an R Project [in Rstudio]

An **R Project** will keep all the files associated with a project (including invisible ones!) organized together – input data, R scripts, analytical results, figures. Besides being common practice, this has the advantage of implicitly setting the “working directory”, which is incredibly important when you need to load or output files, specifying their file path.

In [Figure 1](#) you can see how easy it is just following RStudio prompts:

- Create a new directory for each project
- Select parent folder

Creating an R Project [in Rstudio] (co

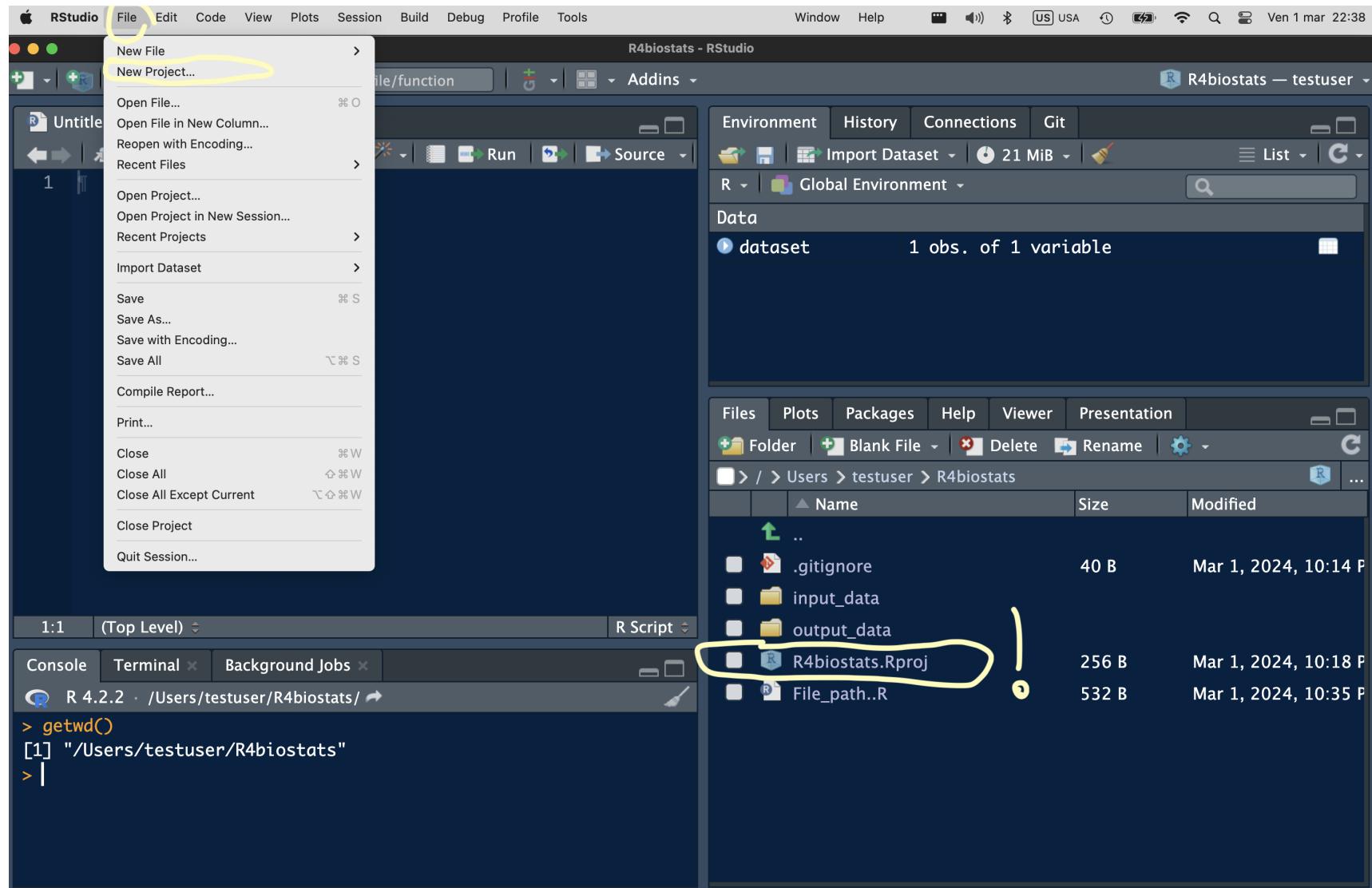


Figure 1: Creating an R project

Install R packages from CRAN (stable version)

An **R package*** is a shareable bundle of functions. Besides the basic built-in functions already contained in the program (i.e. the **base** package), many useful R functions come in free libraries of code (or *packages*) written by R's users. You can find them in different repositories:

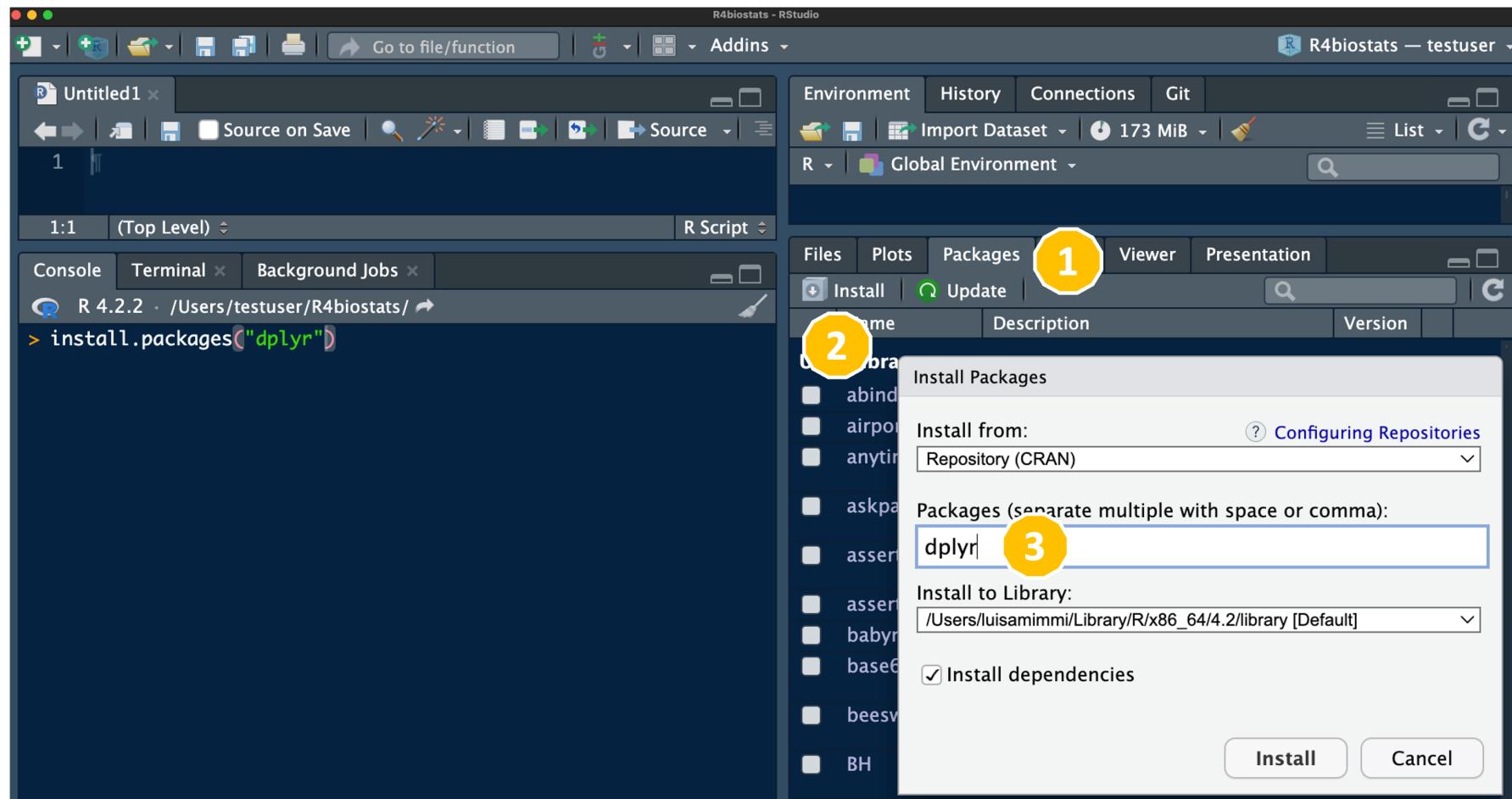
To install a package use **utils** function `install.packages("package_name")`

```
1 # Installing (ONLY the 1st time)
2 utils::install.packages('here')
3
4 # OR (same)
5 install.packages('here')
```

Here you are actually using a *function* (`install.packages`) of a pre-installed *package* (**utils**) using the syntax `packagename::function_name`. This prevents any ambiguity in case of duplicate function name... also helps you see

Install R packages RStudio pane

In alternative, you can install/update packages using the **Packages** tab on the lower right pane of RStudio.



Screenshot Install/Update pckgs from RStudio

Install R packages from GitHub (testing version)

Use `install_github` from the package `devtools`.

EXAMPLE: let's install a little package `paint` (which colors the structure of dataset when printing).

Code

Output `{str}` function

Output `{paint}` function

```
1 # Installing devtools (ONLY the 1st time)
2 utils::install.packages('devtools')
3
4 # Installing paint from GitHub
5 library(devtools)
6 devtools::install_github("MilesMcBain/paint")
7
8 # test paint out
9 library(paint)
```

Use R Packages

- We will be using {base} & {utils} (pre-installed and pre-loaded)
- We will also use the packages below (specifying **package::function** for clarity).

```
1 # Load them for this R session
2 library(here)      # tools find your project's files, based on working directory
3 library(janitor)    # tools for examining and cleaning data
4 library(skimr)      # tools for summary statistics
5 library(dplyr)       # {tidyverse} tools for manipulating and summarising tidy data
6 library(ggplot2)     # {tidyverse} tools for plotting
7 library(forcats)    # {tidyverse} tool for handling factors
8 library(ggridges)   # alternative to plot density functions
9 library(fs)          # file/directory interactions
```

Help on R package/function

To inquire about a package and/or its functions, you can again write in your console `?package_name` or `??package_name` and RStudio will open up the **Help** tab in the lower right pane.

```
1 # Opening Help page on package/function  
2 ?here  
3  
4 ??here
```

File paths logistics

It is never good practice to “hard code” the file’s *absolute path*: most likely it will break your code as soon as you (or someone else) need to run it on a different computer, let alone within a different OS.

Let’s look at this example code using function `readr::read_csv()` (which reads a `*.CSV` data file into the R workspace)

```
1 # [NOT REPRODUCIBLE] hard coding your file path -----
2
3 # File path on Mac:
4 dataset <- readr::read_csv("/Users/testuser/R4biostats/input_data/dataset.csv")
5 # Same file path on Windows:
6 dataset <- readr::read_csv("C:\Users\testuser\R4biostats\input_data\dataset.csv")
```

 ...it won’t work on any other computer since it won’t have that same file structure!

(Reproducible) file paths with `here` (in Rstudio)

The `here` package lets you reference file paths in a **reproducible** manner (anchored on the R Project's folder as the **root**).

Where is my Working Directory?

```
1 here::here()
```

You should get: “/Users/YourName/RProj_Dir”

Now, you can embed `here(dir, subdir)` specifications in other functions.

For example, create sub-directories (for saving input data and output data) with the `fs` package

```
1 ## --- [check the function documentation]
2 ?fs::dir_create
3 # with `here` I simply add subfolder names relative to my wd
4 fs::dir_create(here("practice", "data", "data_input"))
5 # ...and a subfolder to put output files at the end
6 fs::dir_create(here("practice", "data", "data_output"))
7
8 ## --- [if I need to remove it (I have them already)]
9 fs::dir_delete(here("practice", "data", "MINEGEST::training(2024))
```

R OBJECTS, FUNCTIONS, PACKAGES

Importing data into R workspace

We are using real data provided by Thabtah,Fadi. (2017). Autism Screening Adult. UCI Machine Learning Repository. <https://doi.org/10.24432/C5F019>

We use `utils::read.csv` to load a csv file

```
1 ?read.csv # to learn about function and arguments
```

Option 1: Importing from a url

```
1 autism_data_url <- read.csv(  
2   file = "https://raw.githubusercontent.com/Sydney-Informatics-Hub/lessonbmc/gh-pages/_epis  
3   header = TRUE, # 1st line is the name of the variables  
4   sep = ",", # which is the field separator character.  
5   na.strings = c("?") # specific values R should interpret as NA  
6 )
```

Option 2: Importing from my folder (if you previously downloaded the file)

- `here` lets me specify the complete path of the destination folder
- (make sure to replace the file path `here(...)` according to *your own* folder structure)

```
1 # Check my working directory location
2 # here::here()
3
4 # Use `here` in specifying all the subfolders AFTER the working directory
5 autism_data_file <- read.csv(
6   file = here("practice", "data_input", "01_datasets", "autism_data.csv"),
7   header = TRUE, # 1st line is the name of the variables
8   sep = ",", # which is the field separator character.
9   na.strings = c(?),# specific values R should interpret as NA
10  row.names = NULL)
```

DATA OBSERVATION & MANIPULATION

Viewing the dataset and variables

```
1 View(autism_data_file)
```

- Or click on the object in Environment tab (upper right pane of RStudio)

```
1 # What data type is this data?
2 class(autism_data_file)

[1] "data.frame"

1 # What variables are included in this dataset?
2 base::colnames(autism_data_file)

[1] "id"                  "A1_Score"            "A2_Score"            "A3_Score"
[5] "A4_Score"             "A5_Score"             "A6_Score"             "A7_Score"
[9] "A8_Score"             "A9_Score"             "A10_Score"            "age"
[13] "gender"               "ethnicity"            "jaundice"             "autism"
[17] "contry_of_res"        "used_app_before"      "result"                "age_desc"
[21] "relation"              "Class.ASD"
```

- Notice the var name formatting inconsistency: **Class.ASD**

Manipulate / clean the dataframe

I want consistent name formatting for variables: no “`.`”, only “`_`” separator. So, I use a very handy function `clean_names` from the `janitor` package

```
1 autism_data <- janitor::clean_names(autism_data_file,
2                                         case = "none")
3 # check change
4 colnames(autism_data)

[1] "id"                  "A1_Score"            "A2_Score"            "A3_Score"
[5] "A4_Score"             "A5_Score"             "A6_Score"             "A7_Score"
[9] "A8_Score"             "A9_Score"             "A10_Score"            "age"
[13] "gender"               "ethnicity"            "jaundice"             "autism"
[17] "contry_of_res"        "used_app_before"      "result"                "age_desc"
[21] "relation"              "Class_ASD"

1 dim(autism_data)

[1] 704 22
```

- By default `clean_names` renames cols into “**snake**” format (i.e. “`abc_xyz`”)
- The option `case` is for capitalization preferences
 - `case = "none"` leaves the case as is, but only uses “`_`” separator

Isolate a variable (column)

You can use the **\$** sign to extract a variable (column name)

```
1 autism_data$id  
2 autism_data$A1_Score  
3 autism_data$gender  
4 autism_data$autism
```

Add a new column

(I prefer to rename the dataframe when I make changes)

```
1 # rename dataframe  
2 autism_pids <- autism_data
```

Create a **new column**, using **paste** (function to concatenate strings)

```
1 # create a new column  
2 autism_pids$pids <- paste("PatientID_" , autism_data$id, sep = "")
```

Check results:

```
1 # check change in df structure  
2 base:::colnames(autism_pids)  
  
[1] "id"                 "A1_Score"           "A2_Score"           "A3_Score"  
[5] "A4_Score"           "A5_Score"           "A6_Score"           "A7_Score"  
[9] "A8_Score"           "A9_Score"           "A10_Score"          "age"  
[13] "gender"             "ethnicity"          "jaundice"          "autism"  
[17] "contry_of_res"      "used_app_before"    "result"             "age_desc"  
[21] "relation"           "Class_ASD"          "pids"
```

```
1 dim(autism_data)
```

```
[1] 704 22
```

```
1 dim(autism_pids)
```

```
[1] 704 23
```

(optional) Clean up my workspace

```
1 # what do I have in the environment?  
2 ls()  
  
[1] "autism_data"        "autism_data_file" "autism_pids"  
  
1 # remove all EXCEPT for "autism_pids"  
2 rm("autism_data", "autism_data_file", "autism_data_url" )
```

Different ways to select rows &/or columns (from base)

Option 1 Extract cols with

- (**head** only specifies to take the first 6 observations of the dataset)

```
1 # With the `\$` sign I extract a variable (column name)
2 head(autism_pids$id)

[1] 1 2 3 4 5 6

1 head(autism_pids$pids)

[1] "PatientID_1" "PatientID_2" "PatientID_3" "PatientID_4" "PatientID_5"
[6] "PatientID_6"

1 head(autism_pids$A1_Score)

[1] 1 1 1 1 1 1

1 head(autism_pids$ethnicity)

[1] "White-European" "Latino"           "Latino"           "White-European"
[5] NA                 "Others"
```

Option 2a Extract cols with [, #cols]

- This is called “indexing”
- (**head** only specifies to take the first 6 observations of the dataset)

```
1 # Indexing to pick `[, #col]`  
2 head(autism_pids[ ,1]) # empty rows means all  
[1] 1 2 3 4 5 6  
  
1 head(autism_pids[ ,23])  
[1] "PatientID_1" "PatientID_2" "PatientID_3" "PatientID_4" "PatientID_5"  
[6] "PatientID_6"  
  
1 head(autism_pids[ ,2])  
[1] 1 1 1 1 1 1  
  
1 head(autism_pids[ ,14])  
[1] "White-European" "Latino"           "Latino"           "White-European"  
[5] NA                 "Others"
```

Option 2b Extract rows with [#row

```
1 # Indexing to pick `[#row, ]`  
2 head(autism_pids[1 , ] ) # empty cols means all
```

```
id A1_Score A2_Score A3_Score A4_Score A5_Score A6_Score A7_Score A8_Score  
1 1 1 1 1 1 0 0 1 1  
A9_Score A10_Score age gender ethnicity jaundice autism contry_of_res  
1 0 0 26 f White-European no no United States  
used_app_before result age_desc relation Class_ASD pids  
1 no 6 18 and more Self NO PatientID_1
```

```
1 head(autism_pids[50 , ])
```

```
id A1_Score A2_Score A3_Score A4_Score A5_Score A6_Score A7_Score A8_Score  
50 50 1 1 0 0 0 1 1  
A9_Score A10_Score age gender ethnicity jaundice autism contry_of_res  
50 0 1 30 f Asian no no Bangladesh  
used_app_before result age_desc relation Class_ASD pids  
50 no 6 18 and more Self NO PatientID_50
```

```
1 head(autism_pids[25:26 , ])
```

```
id A1_Score A2_Score A3_Score A4_Score A5_Score A6_Score A7_Score A8_Score  
25 25 1 1 1 1 0 0 0 1  
26 26 0 1 1 0 0 0 0 1  
A9_Score A10_Score age gender ethnicity jaundice autism contry_of_res  
25 0 0 43 m <NA> no no Lebanon  
26 0 0 24 f <NA> yes no Afghanistan  
used_app_before result age_desc relation Class_ASD pids  
25 no 5 18 and more <NA> NO PatientID_25  
26 no 3 18 and more <NA> NO PatientID_26
```

Option 3 Extract rows & cols with [#row, #cols]

```
1 # Indexing to pick `[#row, #col]`  
2 autism_pids[1:3,1]
```

```
[1] 1 2 3
```

```
1 autism_pids[1:3,23]
```

```
[1] "PatientID_1" "PatientID_2" "PatientID_3"
```

```
1 autism_pids[1:3,2]
```

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

```
1 autism_pids[1:3,14]
```

```
[1] "White-European" "Latino"           "Latino"
```

What are the data types of the variables?

Option 1 using base functions

- on the whole dataset

```
1 # What are the data types of the variables? -----
2 str(autism_pids) # integer and character

'data.frame': 704 obs. of 23 variables:
 $ id          : int 1 2 3 4 5 6 7 8 9 10 ...
 $ A1_Score    : int 1 1 1 1 1 1 0 1 1 1 ...
 $ A2_Score    : int 1 1 1 1 0 1 1 1 1 1 ...
 $ A3_Score    : int 1 0 0 0 0 1 0 1 0 1 ...
 $ A4_Score    : int 1 1 1 1 0 1 0 1 0 1 ...
 $ A5_Score    : int 0 0 1 0 0 1 0 0 1 0 ...
 $ A6_Score    : int 0 0 0 0 0 0 0 0 0 1 ...
 $ A7_Score    : int 1 0 1 1 0 1 0 0 0 1 ...
 $ A8_Score    : int 1 1 1 1 1 1 0 1 1 1 ...
 $ A9_Score    : int 0 0 1 0 0 1 0 1 1 1 ...
 $ A10_Score   : int 0 1 1 1 0 1 0 0 1 0 ...
 $ age         : int 26 24 27 35 40 36 17 64 29 17 ...
 $ gender      : chr "f" "m" "m" "f" ...
 $ ethnicity   : chr "White-European" "Latino" "Latino" "White-European" ...
 $ jaundice    : chr "no" "no" "yes" "no" ...
 $ autism       : chr "no" "yes" "yes" "yes" ...
 $ contry_of_res: chr "United States" "Brazil" "Spain" "United States" ...
 $ used_app_before: chr "no" "no" "no" "no" ...
 $ result       : int 6 5 8 6 2 9 2 5 6 8 ...
```

Option 1 using **base** functions (cont.)

- on specific columns

```
1 # What values can the variables take? -----
2 summary(autism_pids$pids)
```

```
Length     Class      Mode
704 character character
```

```
1 length(unique(autism_pids$pids)) # N unique values
```

```
[1] 704
```

```
1 sum(is.na(autism_pids$pids)) # N missing values
```

```
[1] 0
```

```
1 summary(autism_pids$ethnicity)
```

```
Length     Class      Mode
704 character character
```

```
1 length(unique(autism_pids$ethnicity)) # N unique values
```

```
[1] 12
```

```
1 sum(is.na(autism_pids$ethnicity)) # N missing values
```

```
[1] 95
```

Option 2 using **skimr** function **skim**

- on specific columns

```
1 autism_pids %>%
2   skimr::skim(pids, ethnicity) %>%
3   dplyr::select(#skim_variable,
4                 skim_type,
5                 complete_rate,
6                 n_missing,
7                 character.n_unique)

# A tibble: 2 × 4
  skim_type complete_rate n_missing character.n_unique
  <chr>           <dbl>      <int>                <int>
1 character          1            0                  704
2 character        0.865        95                  11
```

Option 2 using `skimr` function `skim` (cont.)

- on the whole dataset

```
1 # Barbatrucco !!!!!  
2 autism_pids %>%  
3   skimr::skim()
```

— Variable type: character —								
	skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
1	gender	0	1	1	1	0	2	0
2	ethnicity	95	0.865	5	15	0	11	0
3	jaundice	0	1	2	3	0	2	0
4	autism	0	1	2	3	0	2	0
5	contry_of_res	0	1	4	20	0	67	0
6	used_app_before	0	1	2	3	0	2	0
7	age_desc	0	1	11	11	0	1	0
8	relation	95	0.865	4	24	0	5	0
9	Class_ASD	0	1	2	3	0	2	0
10	pids	0	1	11	13	0	704	0

— Variable type: numeric —										
	skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
1	id	0	1	352.	203.	1	177.	352.	528.	704
2	A1_Score	0	1	0.722	0.449	0	0	1	1	1
3	A2_Score	0	1	0.453	0.498	0	0	0	1	1
4	A3_Score	0	1	0.457	0.499	0	0	0	1	1

Recoding variables

From character to factor

char to factor using **base**

char to factor (n cols) using **base**

```
1 ##### char 2 factor -----
2 # Say I want to treat some variables as factors
3 autism_pids$gender <- as.factor(autism_pids$gender)
4 autism_pids$ethnicity <- as.factor(autism_pids$ethnicity)
5 autism_pids$contry_of_res <- as.factor(autism_pids$contry_of_res)
6 autism_pids$relation <- as.factor(autism_pids$relation)
7
8 # check
9 class(autism_pids$gender)

[1] "factor"

1 class(autism_pids$ethnicity)

[1] "factor"

1 class(autism_pids$contry_of_res)

[1] "factor"

1 class(autism_pids$relation)

[1] "factor"
```

Inspect factors levels (3 different ways)

- using `base::levels` function

```
1 levels(autism_pids$ethnicity)

[1] "Asian"           "Black"          "Hispanic"        "Latino"
[5] "Middle Eastern" "others"         "Others"          "Pasifika"
[9] "South Asian"    "Turkish"        "White-European"
```

- using `base::table` function

```
1 table(autism_pids$ethnicity,useNA = "ifany")
```

	Asian	Black	Hispanic	Latino	Middle Eastern
	123	43	13	20	92
others		Others	Pasifika	South Asian	Turkish
	1	30	12	36	6
White-European		<NA>			
	233	95			

Inspect factors levels – 3 different ways (cont.)

- using **janitor** function **tabyl**, which uses the “pipe” operator `%>%` which takes the output of a function as input of the next one

```
1 janitor::tabyl(autism_pids$ethnicity) %>%
  2   adorn_totals() %>%
  3   adorn_pct_formatting()

autism_pids$ethnicity    n percent valid_percent
          Asian    123    17.5%      20.2%
          Black     43    6.1%       7.1%
          Hispanic   13    1.8%       2.1%
          Latino    20    2.8%       3.3%
Middle Eastern    92    13.1%      15.1%
          others     1    0.1%       0.2%
          Others    30    4.3%       4.9%
          Pasifika   12    1.7%       2.0%
South Asian     36    5.1%       5.9%
          Turkish     6    0.9%       1.0%
White-European  233   33.1%      38.3%
          <NA>      95   13.5%      -
          Total     704   100.0%     100.0%
```

Identify missing values

Use `is.na` to check if the 95 missing obs are the same missing for `ethnicity` and `relation`

```
1 which(is.na(autism_pids$ethnicity)) # indices of TRUE elements in vector
```

```
[1] 5 13 14 15 20 21 25 26 63 80 81 82 92 217 222 239 258 271 277  
[20] 278 286 307 316 325 338 339 340 341 342 343 344 345 346 347 348 349 350 351  
[39] 352 353 354 355 356 362 366 370 371 373 379 380 381 382 383 384 385 386 387  
[58] 388 389 391 396 400 401 402 404 424 428 429 430 433 439 454 486 506 519 528  
[77] 535 536 537 538 557 565 572 573 589 594 637 643 646 652 653 659 660 667 702
```

```
1 which(is.na(autism_pids$relation)) # indices of TRUE elements in vector
```

```
[1] 5 13 14 15 20 21 25 26 63 80 81 82 92 217 222 239 258 271 277  
[20] 278 286 307 316 325 338 339 340 341 342 343 344 345 346 347 348 349 350 351  
[39] 352 353 354 355 356 362 366 370 371 373 379 380 381 382 383 384 385 386 387  
[58] 388 389 391 396 400 401 402 404 424 428 429 430 433 439 454 486 506 519 528  
[77] 535 536 537 538 557 565 572 573 589 594 637 643 646 652 653 659 660 667 702
```

...indeed they are the same IDs!

From character to logical

I may prefer to code a variable as logical. For example, `age_desc` may be more explicit if coded as logical.

- I create a new column `age_desc_log`

```
1 # observe a subset of some columns
2 autism_subset <- autism_pids[1:5, c("gender", "jaundice", "autism", "age_desc", "Class_ASD")]
3 # View(autism_subset)
4
5 # recode "age_desc" as LOGICAL new var "age_desc_log"
6 autism_pids$age_desc_log <- ifelse(autism_pids$age_desc == "18 and more", TRUE, FALSE)
7 class(autism_pids$age_desc)

[1] "character"

[1] class(autism_pids$age_desc_log)

[1] "logical"
```

From character to dummy [0,1]

I also may need my binary variables expressed as [0,1] (e.g. to incorporate nominal variables into regression analysis). Let's recode **autism**.

```
1 autism_pids$autism_dummy <- ifelse(autism_pids$autism == 'yes', 1, 0)
2 class(autism_pids$autism)

[1] "character"

1 class(autism_pids$autism_dummy)

[1] "numeric"
```

Subsetting the data for further investigation

Recall how to view the names of columns / variables

```
1 colnames(autism_pids)
```

```
[1] "id"                 "A1_Score"           "A2_Score"           "A3_Score"
[5] "A4_Score"           "A5_Score"           "A6_Score"           "A7_Score"
[9] "A8_Score"           "A9_Score"           "A10_Score"          "age"
[13] "gender"             "ethnicity"          "jaundice"          "autism"
[17] "contry_of_res"      "used_app_before"    "result"            "age_desc"
[21] "relation"           "Class_ASD"          "pids"              "age_desc_log"
[25] "autism_dummy"
```

using head or tail from utils

- `head` or `tail` return the first or last parts of an object

```
1 head(autism_pids)    #return fist 6 obs  
2 tail(autism_pids)   #return last 6 obs
```

using head or tail from utils (cont.)

```
1 head(autism_pids, n = 2) #return first 2 obs
```

```
  id A1_Score A2_Score A3_Score A4_Score A5_Score A6_Score A7_Score A8_Score
1  1          1          1          1          1          0          0          1          1
2  2          1          1          0          1          0          0          0          1
  A9_Score A10_Score age gender      ethnicity jaundice autism contry_of_res
1      0          0    26      f  White-European        no       no United States
2      0          1    24      m      Latino        no      yes     Brazil
  used_app_before result      age_desc relation Class_ASD           pids
1      no      6 18 and more      Self      NO PatientID_1
2      no      5 18 and more      Self      NO PatientID_2
  age_desc_log autism_dummy
1      TRUE          0
2      TRUE          1
```

```
1 tail(autism_pids, n = 2) #return last 2 obs
```

```
  id A1_Score A2_Score A3_Score A4_Score A5_Score A6_Score A7_Score A8_Score
703 703          1          0          0          1          1          0          1          0
704 704          1          0          1          1          1          0          1          1
  A9_Score A10_Score age gender      ethnicity jaundice autism contry_of_res
703      1          1    35      m  South Asian        no       no    Pakistan
704      1          1    26      f  White-European        no       no     Cyprus
  used_app_before result      age_desc relation Class_ASD           pids
703      no      6 18 and more      Self      NO PatientID_703
704      no      8 18 and more      Self      YES PatientID_704
  age_desc_log autism_dummy
703      TRUE          0
704      TRUE          0
```

Investigating a subset of observations

E.g. I learned that some patients have missing `age`... how many are they?

```
1 # run...
2 sum(is.na(autism_pids$age))

[1] 2

1 # or
2 skimr::n_missing(autism_pids$age)

[1] 2
```

So, next, I want to ID those patients with missing `age`.

New df (only the patients with missing age) as SUBSET of the given df

I want to extract only the obs (*rows*) of interest with a few useful vars (*cols*)

using `[]` from `base`

using `which` from `base`

using `subset` from `base`

```
1 missing_age_subset <- autism_pids[is.na(autism_pids$age), c("pids", "age", "autism_dummy")]
2 missing_age_subset
```

	pids	age	autism_dummy
63	PatientID_63	NA	0
92	PatientID_92	NA	0

New df (filtering on 2 conditions) as SUBSET of the given df

using `base::subset`

using `dplyr(filter + select)`

```
1 # Creates a SUBSET based on MORE conditions (`age` and `ethnicity`)
2 twocond_base_subset <- subset(x = autism_pids,
3                               # 2 logical conditions
4                               subset = age < 25 & contry_of_res == "Brazil",
5                               # pick a few cols
6                               select = c("pids", "age", "contry_of_res",
7                                       "autism_dummy"))
8
9 twocond_base_subset
```

	pids	age	contry_of_res	autism_dummy
2	PatientID_2	24	Brazil	1
54	PatientID_54	21	Brazil	1
94	PatientID_94	19	Brazil	1
429	PatientID_429	20	Brazil	0
587	PatientID_587	21	Brazil	0
588	PatientID_588	21	Brazil	0

Dealing with missing data

Input values where missing

⚠ WARNING: This is a very delicate step ⚠

- any modified/imputed data (beyond the *original collection*) can affect subsequent analysis and statistical modeling
- it will be necessary to document and justify whichever approach is used to deal with missing data.

Let's assume we can get the missing data by cross-checking related clinical information

```
1 # 1/2 create a new variable
2 autism_pids$age_inputed <- autism_pids$age
3 # 2/2 replace value (presumably taken from other source) of `aged_inputed`
4   # CONDITIONAL on `pids`
5 autism_pids$age_inputed[autism_pids$pids == "PatientID_63"] <- 65
6 autism_pids$age_inputed[autism_pids$pids == "PatientID_92"] <- 45
7
8 # check
9 skimr::n_missing(autism_pids$age)
```

```
[1] 2
```

```
1 skimr::n_missing(autism_pids$age_inputed)
```

```
[1] 0
```

DESCRIPTIVE STATISTICS

Summarizing all variables

Try these 2 options:

base::c

```
1 summary(autism_pids)
```

skimr::skim

```
1 skimr::skim(autism_pids)
```

Notice **summary** different behavior according to the variable's type

The function's results depend on the class of the object

- integer (A1_Score)

```
1 summary(autism_pids$A1_Score)      # min, max quartiles, mean, median
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0000	0.0000	1.0000	0.7216	1.0000	1.0000

- factor (ethnicity)

```
1 summary(autism_pids$ethnicity)      # counts of levels' frequency (included NA!)
```

	Asian	Black	Hispanic	Latino	Middle Eastern
123	123	43	13	20	92
others	others	Others	Pasifika	South Asian	Turkish
1	1	30	12	36	6
White-European	White-European	NA's	95		
233	233				

Notice **summary** different behavior according to the variable's type (cont.)

- logical (age_desc_log)

```
1 summary(autism_pids$age_desc_log) # counts of TRUE
```

Mode	TRUE
logical	704

Frequency distributions with `table`

- Frequency distributions can be used for nominal, ordinal, or interval/ratio variables

```
1 table(autism_pids$gender)
```

```
f     m  
337 367
```

```
1 table(autism_pids$age) # automatically drops missing...
```

```
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42  
18 31 35 46 49 37 37 34 27 28 31 24 27 30 21 18 16 12 17 13 17 13 7 16 3 15  
43 44 45 46 47 48 49 50 51 52 53 54 55 56 58 59 60 61 64  
11 10  4   6   8   4   3   5   1   5   6   2   6   2   2   1   1   2   1
```

```
1 table(autism_pids$age, useNA = "ifany") #...unless specified
```

```
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32  
18 31 35 46 49 37 37 34 27 28 31 24 27 30 21 18  
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48  
16 12 17 13 17 13  7 16  3 15 11 10  4  6  8  4  
49 50 51 52 53 54 55 56 58 59 60 61 64 <NA>  
 3   5   1   5   6   2   6   2   2   1   1   2   1   2
```

Cross tabulation with **table** (2 vars)

- Cross tabulation

```
1 table(autism_pids$gender, autism_pids$age_inputed)
```

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
f	7	11	22	22	18	14	17	10	11	14	18	15	16	13	8	14	6	7	12	7	11	6	5	9	0
m	11	20	13	24	31	23	20	24	16	14	13	9	11	17	13	4	10	5	5	6	6	7	2	7	3
	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	58	59	60	61	64	65				
f	6	5	4	5	4	3	2	2	2	0	2	4	1	1	0	1	0	1	1	0	0				
m	9	6	6	0	2	5	2	1	3	1	3	2	1	5	2	1	1	0	1	1	1				

```
1 table(autism_pids$ethnicity, autism_pids$autism_dummy)
```

	0	1
Asian	118	5
Black	38	5
Hispanic	12	1
Latino	12	8
Middle Eastern	83	9
others	1	0
Others	28	2
Pasifika	10	2
South Asian	34	2
Turkish	5	1
White-European	183	50

Grouping and summarizing with base

E.g. I want to know the average age of men and women sub-groups.

Here are 3 ways using `base R

using by

using tapply

using split + sapply

```
1 # by(data$column, data$grouping_column, mean)
2 by(data = autism_pids$age_inputed, INDICES = autism_pids$gender, FUN = mean)

autism_pids$gender: f
[1] 29.60237
-----
autism_pids$gender: m
[1] 28.98365
```

Grouping and summarizing with dplyr

```
1 autism_pids %>%
2   dplyr::group_by(gender) %>%
3   dplyr::summarise(mean(age_inputted)) # returns a dataframe!

# A tibble: 2 × 2
  gender `mean(age_inputted)`
  <fct>      <dbl>
1 f          29.6
2 m          29.0
```

I could add more statistics to the grouped summary...

```
1 autism_pids %>%
2   dplyr::group_by(gender) %>%
3   dplyr::summarise(mean_age = mean(age_inputted),
4                     N_obs = n(),
5                     N_with_autism = sum(autism_dummy == 1)
6                   )

# A tibble: 2 × 4
  gender mean_age N_obs N_with_autism
  <fct>     <dbl>  <int>        <int>
1 f          29.6    337         54
2 m          29.0    367         37
```

Measures of central tendency

Mean and median

Recall that:

$$\text{Population MEAN } \mu = \frac{\sum_{i=1}^n x_i}{n}$$

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

Sample MEDIAN

$$\text{For uneven n: } Mdn = \frac{x_{(n+1)}}{2} \quad \text{For even n: } Mdn = \frac{x_{(n/2)} + x_{(n/2+1)}}{2}$$

Mean/Median using **base R**

- Important to specify the argument `na.rm = TRUE` or the functions won't work

```
1 ## Let's use `age` and `age_inputed` to see what inputed missing values did
2 mean(autism_pids$age)

[1] NA

1 median(autism_pids$age)

[1] NA

1 mean(autism_pids$age, na.rm = TRUE)

[1] 29.20655

1 median(autism_pids$age, na.rm = TRUE)

[1] 27

1 mean(autism_pids$age_inputed)

[1] 29.27983

1 median(autism_pids$age_inputed)

[1] 27
```

Create custom function to calculate statistical mode

R doesn't have a built-in function for the statistical mode, so we can create a custom one: `f_calc_mode`

Create custom function

Call custom function

```
1 f_calc_mode <- function(x) {  
2   # `unique` returns a vector of unique values  
3   uni_x <- unique(x)  
4   # `match` returns the index positions of 1st vector against 2nd vector  
5   match_x <- match(x, uni_x)  
6   # `tabulate` count the occurrences of integer values in a vector.  
7   tab_x <- tabulate(match_x)  
8   # returns element of uni_x that corresponds to max occurrences  
9   uni_x[tab_x == max(tab_x)]  
10 }
```

Measures of variability (or spread)

Variance and Standard deviation

Recall that:

$$\text{Population Variance } \sigma^2 = \frac{\sum_{i=1}^n (x_i - \mu)^2}{n}$$

$$\text{Sample Variance } s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1}$$

$$\text{Population Standard deviation } \sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n}}$$

$$\text{Sample Standard deviation } s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

Variance and Standard deviation using base R

- Important to specify the argument `na.rm = TRUE` or the functions won't work (or use the `age_inputed` variable)

```
1 var(autism_pids$age, na.rm = TRUE)
[1] 94.28966
1 var(autism_pids$age_inputed)
[1] 96.19328
1 sd(autism_pids$age, na.rm = TRUE)
[1] 9.710286
1 sd(autism_pids$age_inputed)
[1] 9.807817
```

VISUAL DATA EXPLORATION

Plotting with ggplot2

`ggplot2` provides a set of tools to map data to visual elements on a plot, to specify the kind of plot you want, and then subsequently to control the fine details of how it will be displayed. It basically allows to build a plot layer by layer ([Figure 2](#)).

- **data** -> specify what our dataset is
- **aesthetic mappings** (or just *aesthetics*) -> specify which dataset's variables will turn into the plot elements (e.g. x and y values, or categorical variable into colors, points, and shapes).
- **geom** -> the overall type of plot, e.g. `geom_point()` makes scatterplots, `geom_bar()` makes barplots, `geom_boxplot()` makes boxplots.

Additional (optional) pieces:

- information about the **scales**,
- the labels of **legends** and axes
- other **guides** that help people to read the plot,

Plotting with ggplot2 (cont.)

a layered approach!

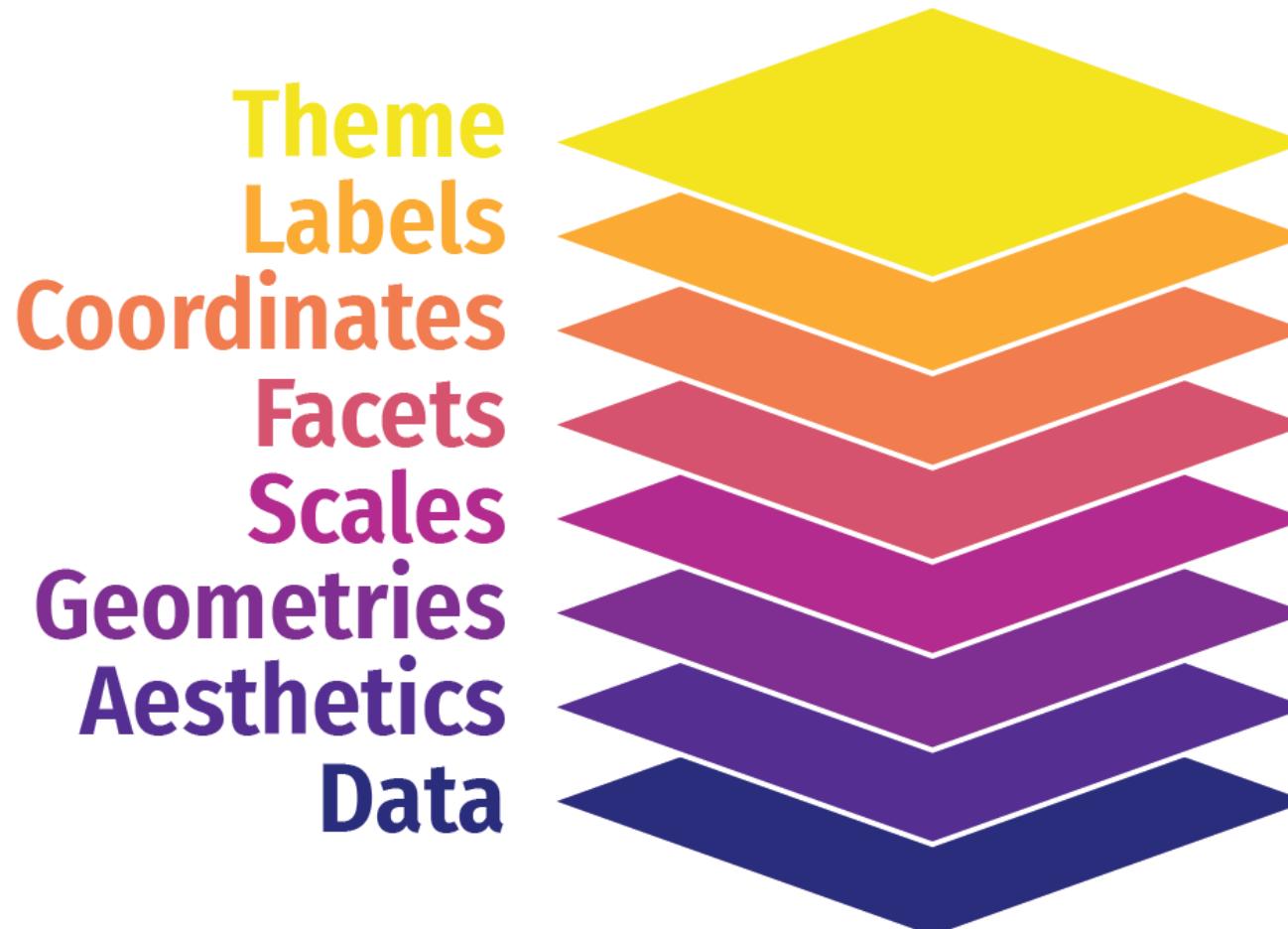


Figure 2: ggplot2 layers [Source: Andrew Heiss' talk](#)

Save some colors (for customizing plots)

- Colors are defined in the form of **Hexadecimal color values**

```
1 two_col_palette <- c("#9b2339", "#005ca1")
2
3 contrast_cols_palette <- c("#E7B800", "#239b85", "#85239b", "#9b8523", "#23399b",
4                           "#d8e600", "#0084e6", "#399B23", "#e60066",
5                           "#00d8e6", "#e68000")
```

Distribution of continuous var

Histograms

Histograms (and density plots) are often used to show the distribution of a continuous variable.

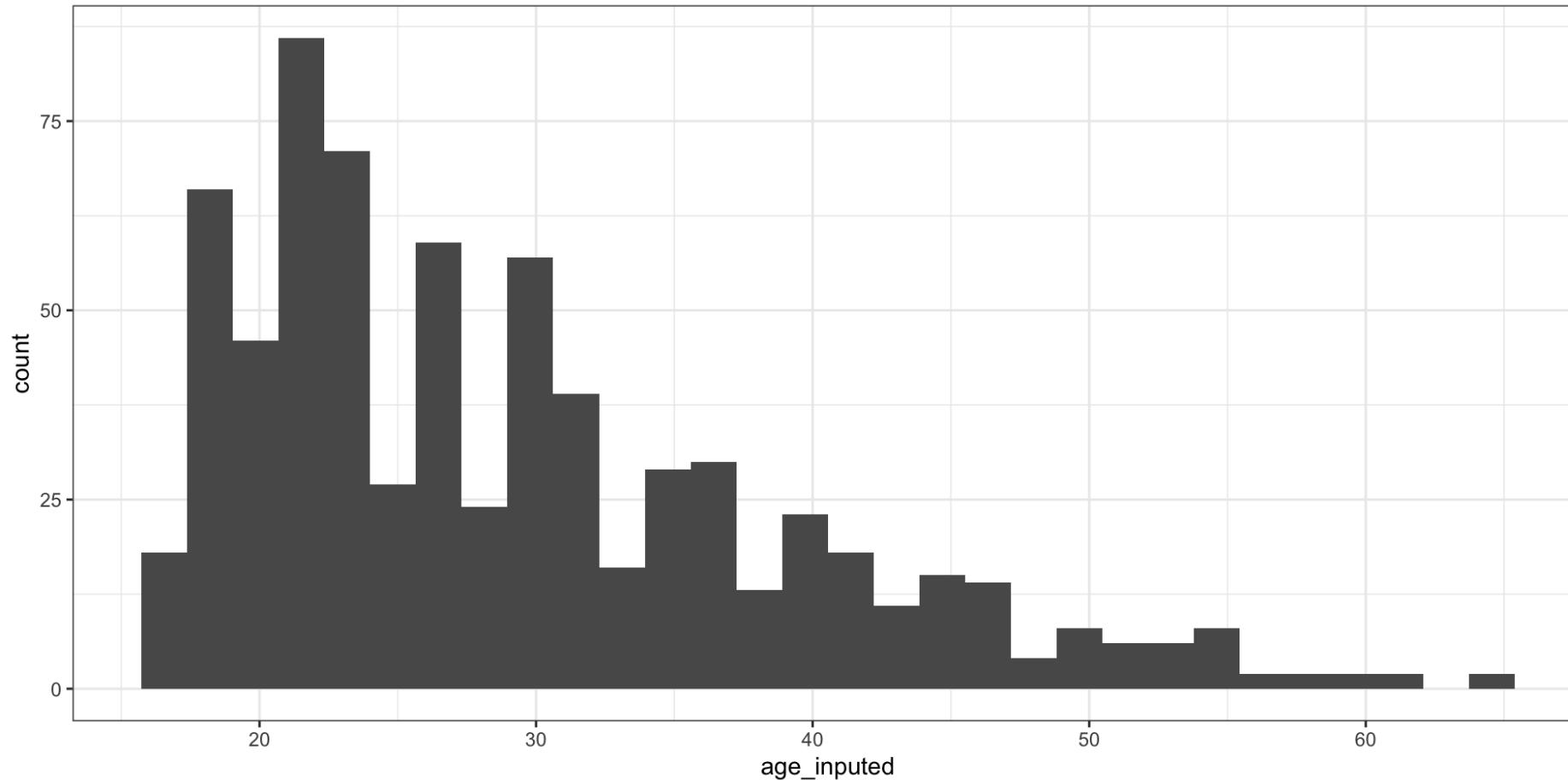
- Option 1) **data** inside the **ggplot()** function

```
1 ggplot(data = autism_pids, mapping = aes(x=age_inputed)) +  
2   geom_histogram() +  
3   theme_bw()
```

- Option 2) **data** before the pipe `%>%`

```
1 autism_pids %>%  
2   ggplot(aes(x = age_inputed )) +  
3   geom_histogram() +  
4   theme_bw()
```

Histograms



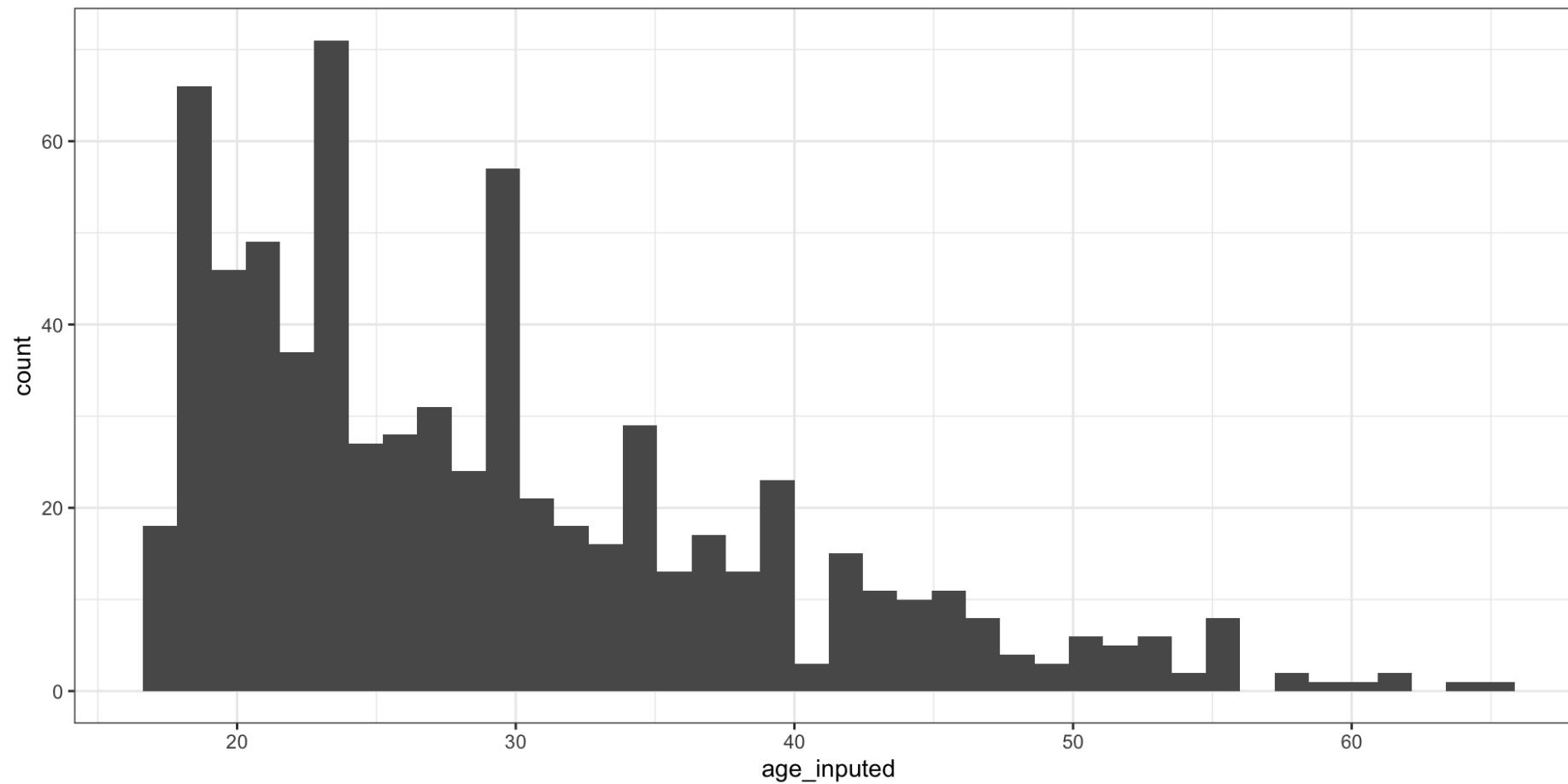
... define bin width

Histograms split the data into ranges (bins) and show the number of observations in each. Hence, it's important to pick widths that represents the data well.

- The default value is 30
- We can change it using the argument **bins = #**

```
1 autism_pids %>%
2   ggplot(aes(x = age_inputed )) +
3   # specify to avoid warning if we fail to specify the number of bins
4   geom_histogram(bins=40) +
5   theme_bw()
```

... define bin width

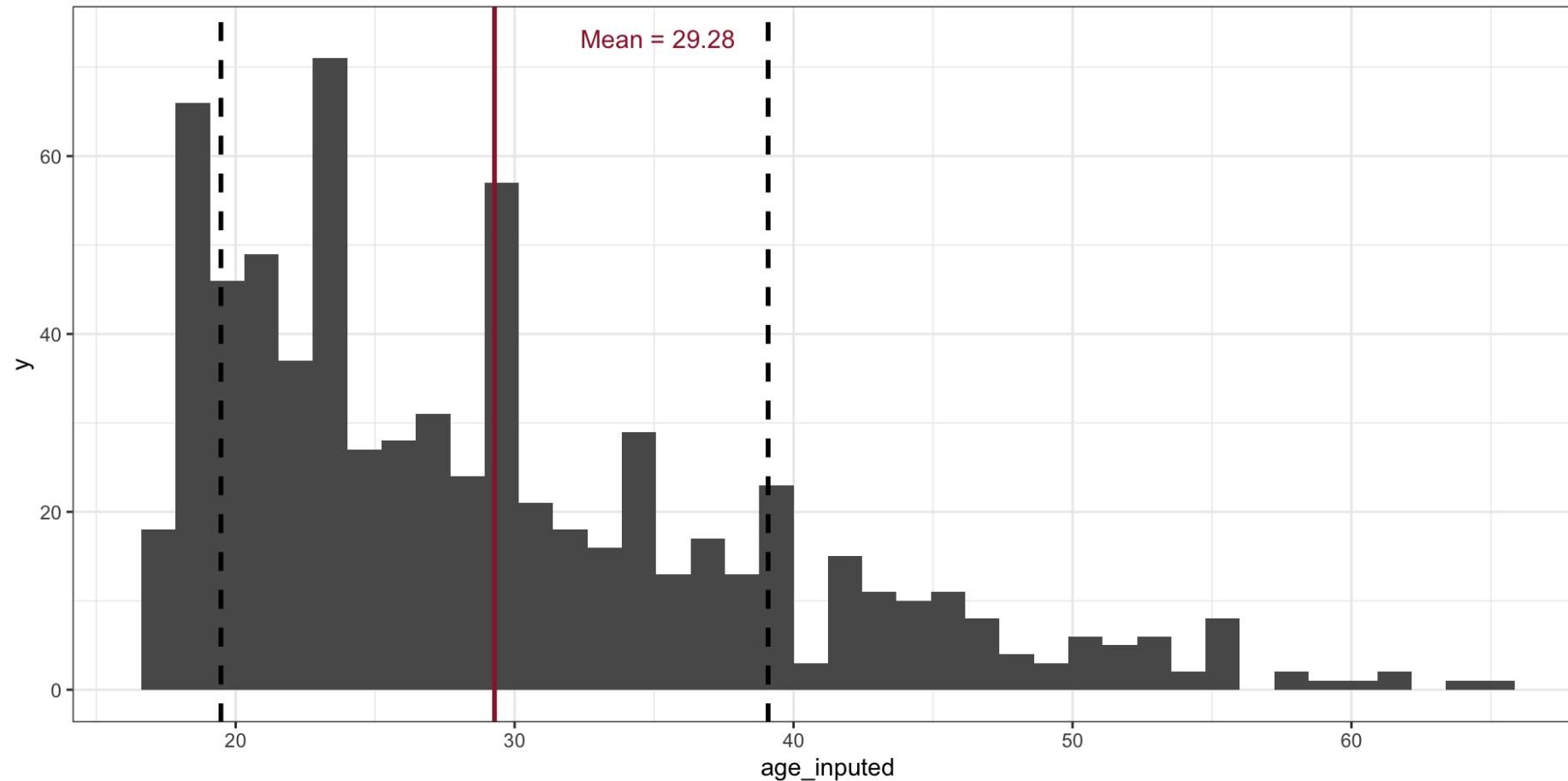


... add mean and std dev vertical lines

- using `geom_vline()` to add a vertical line for the *mean*, and the range between -1 and +1 *sd* from the mean.
- using `annotate()` for adding small annotations (such as text labels)

```
1 autism_pids %>%
2   ggplot(aes(x = age_inputed)) +
3   geom_histogram(bins=40) +
4   # add mean vertical line
5   geom_vline(xintercept = mean(autism_pids$age_inputed),
6             na.rm = FALSE,
7             lwd=1,
8             color="#9b2339") +
9   # add annotations with the mean value
10  annotate("text",
11            x = mean(autism_pids$age_inputed) * 1.2, # coordinates for positioning
12            y = mean(autism_pids$age_inputed) * 2.5,
13            label = paste("Mean =", round(mean(autism_pids$age_inputed), digits = 2)),
14            col = "#9b2339",
15            size = 4) +
16   # add also sd +1 and -1
17   geom_vline(aes(xintercept = mean(autism_pids$age_inputed) + sd(autism_pids$age_inputed)),
18             color = "#000000", size = 1, linetype = "dashed") +
19   geom_vline(aes(xintercept = mean(autism_pids$age_inputed) - sd(autism_pids$age_inputed)),
20             color = "#000000", size = 1, linetype = "dashed") +
21   theme_bw()
```

... add mean and std dev vertical lines

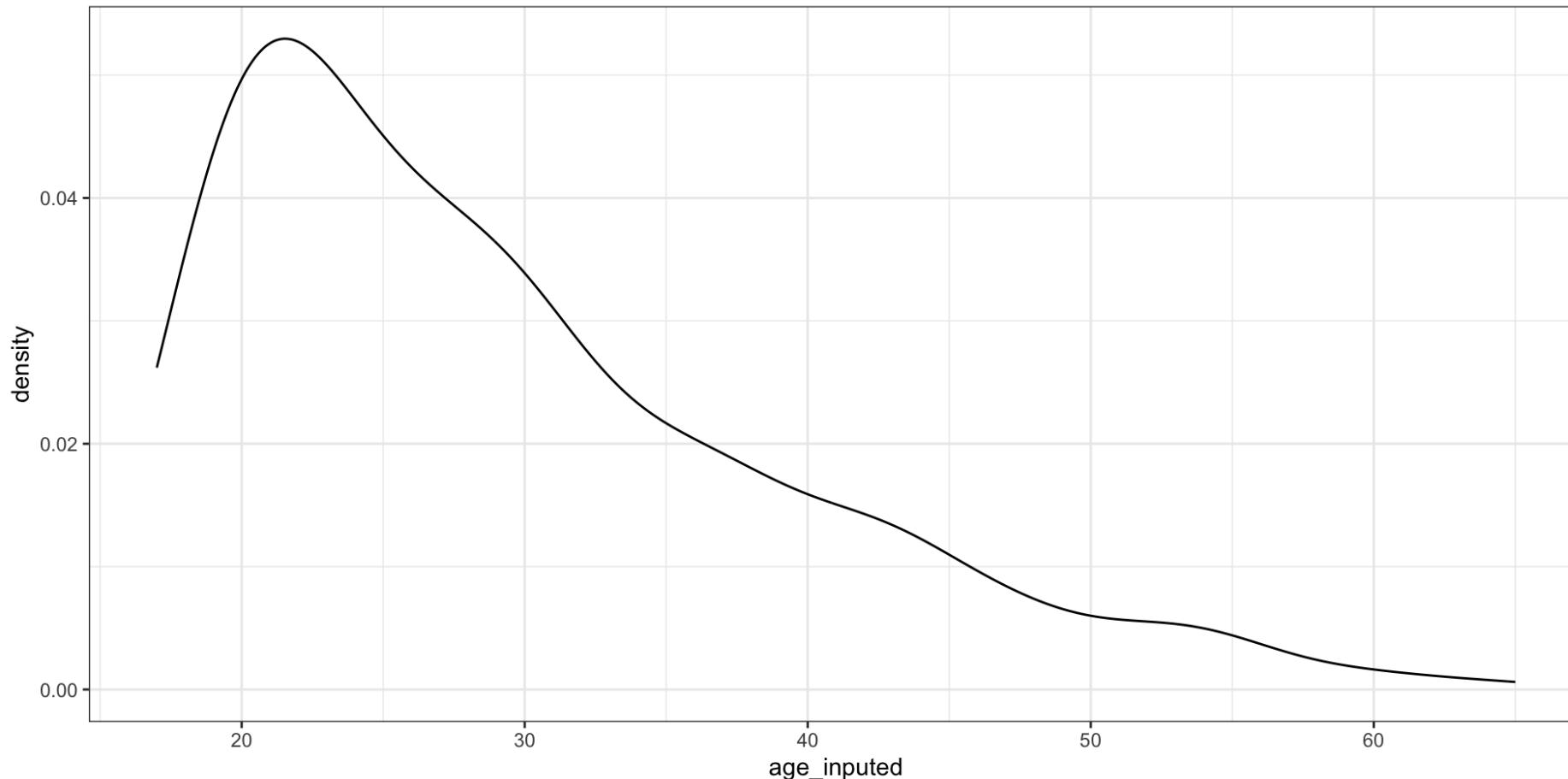


Density plot

- specifying `x` (the continuous variable)
- using `geom_density()`, in which we

```
1 autism_pids %>%
2   ggplot(aes(x = age_inputed)) +
3   geom_density() +
4   theme_bw()
```

Density plot

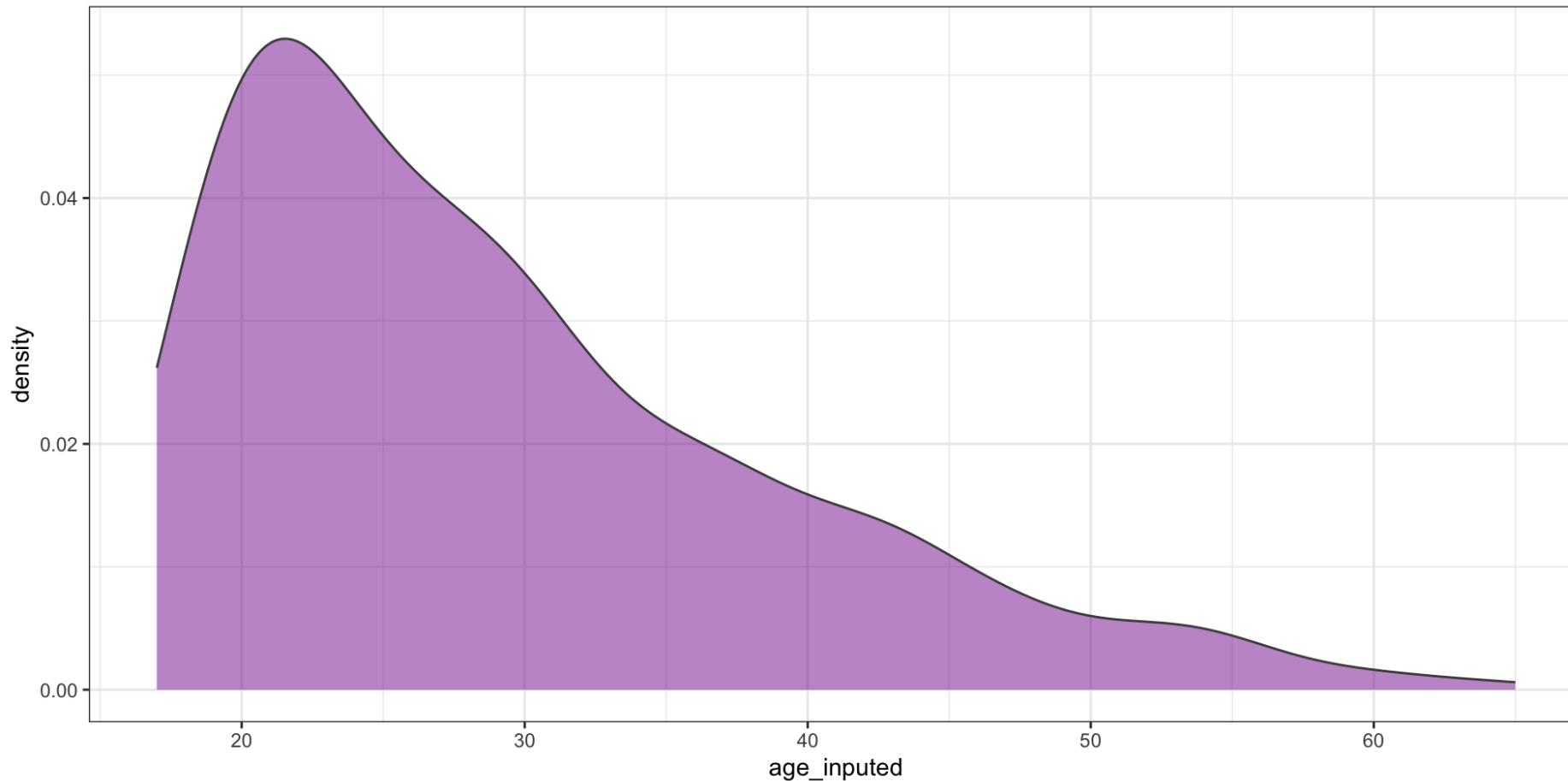


Density plot (cont.)

- specifying shape colors with the arguments inside `geom_density(...)`
- `color` for the line color
- `fill` for area color
- `alpha` to specify the degree of transparency in the density fill area

```
1 autism_pids %>%
2   ggplot(aes( x=age_inputed)) +
3   geom_density(fill="#85239b", color="#4c4c4c", alpha=0.5) +
4   theme_bw()
```

Density plot (cont.)

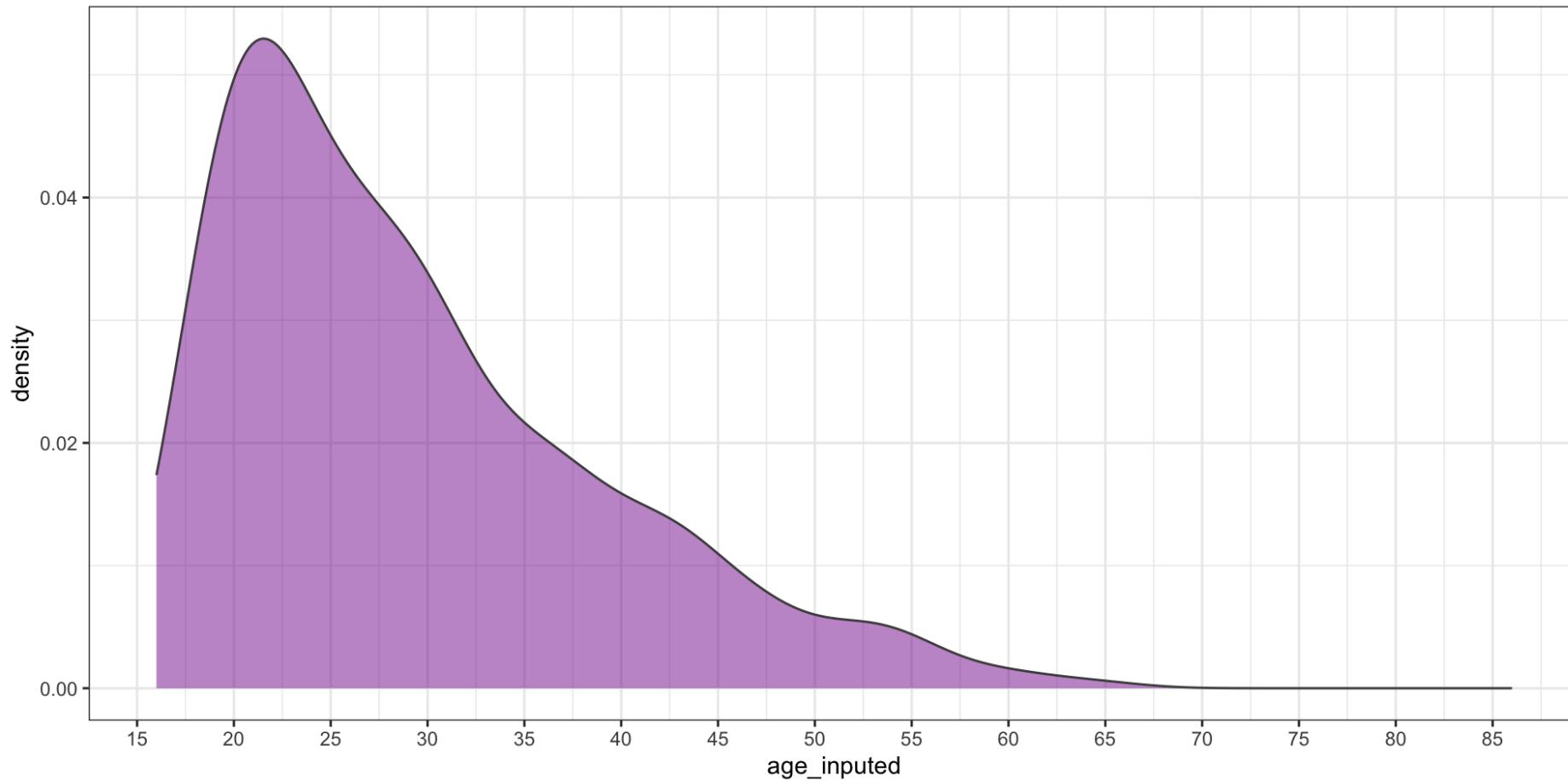


... increase # of x-axis ticks

- specifying the amount of breaks inside `scale_x_continuous()`

```
1 autism_pids %>%
2   ggplot(aes( x=age_inputed)) +
3   geom_density(fill="#85239b", color="#4c4c4c", alpha=0.5) +
4   theme_bw() +
5   # increase number of x axis ticks
6   scale_x_continuous(breaks = seq(10, 100, 5), limits = c(16, 86))
```

... increase # of x-axis ticks



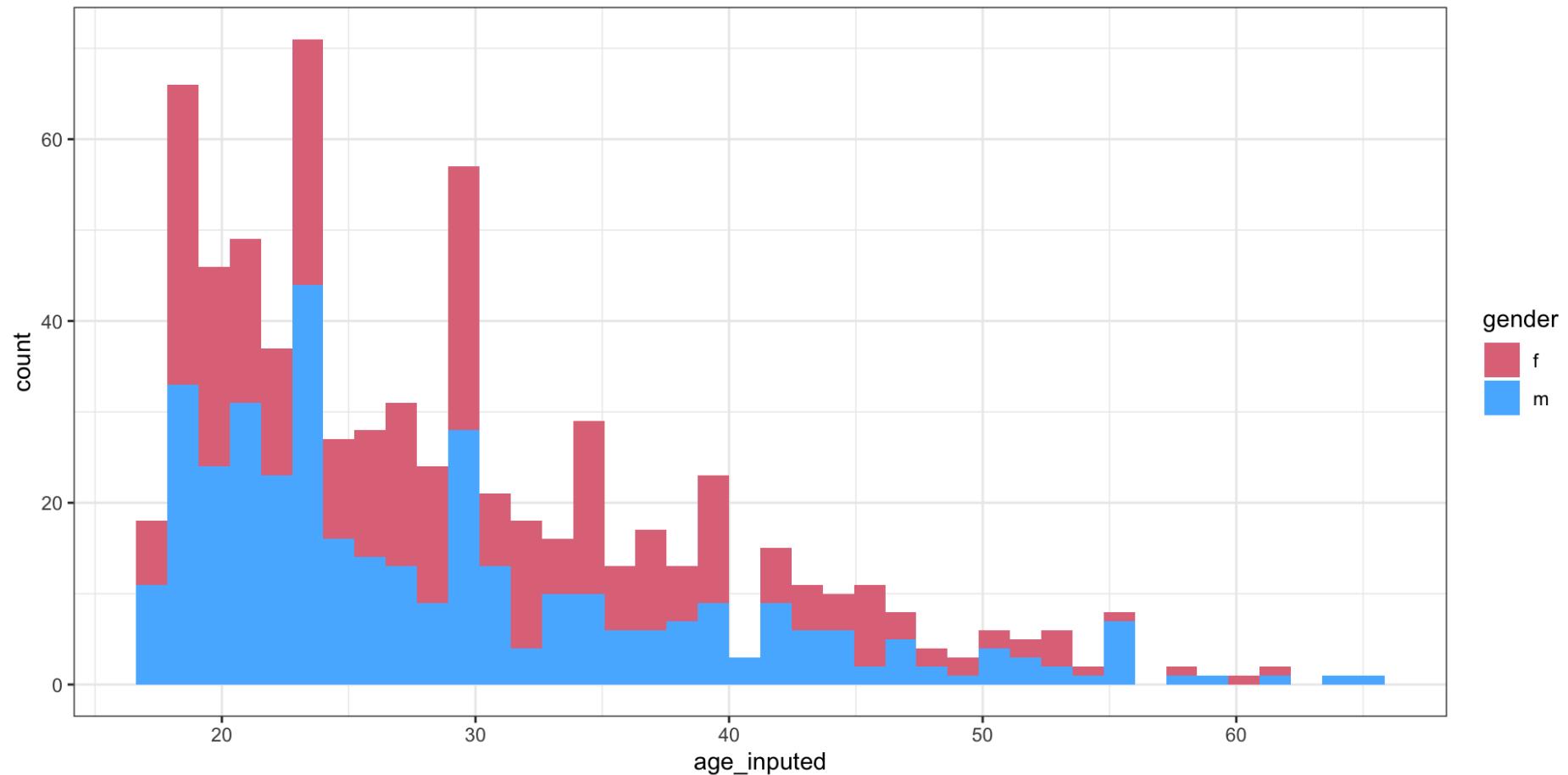
Distribution of continuous var split by categorical var

Histograms with `fill = category`

1. indicate the categorical group as `fill =` in the aesthetic mapping
2. specify custom colors for each group:
 - use `scale_color_manual()` for changing line color
 - use `scale_fill_manual()` for changing area fill colors.

```
1 autism_pids %>%
2   # specifying `fill` = gender
3   ggplot(mapping = aes(x = age_imputed, fill = gender)) +
4   geom_histogram(bins=40) +
5   scale_fill_manual(values = c("#e07689", "#57b7ff")) +
6   scale_color_manual(values = c("#9b2339", "#005ca1")) +
7   theme_bw()
```

Histograms with fill = category

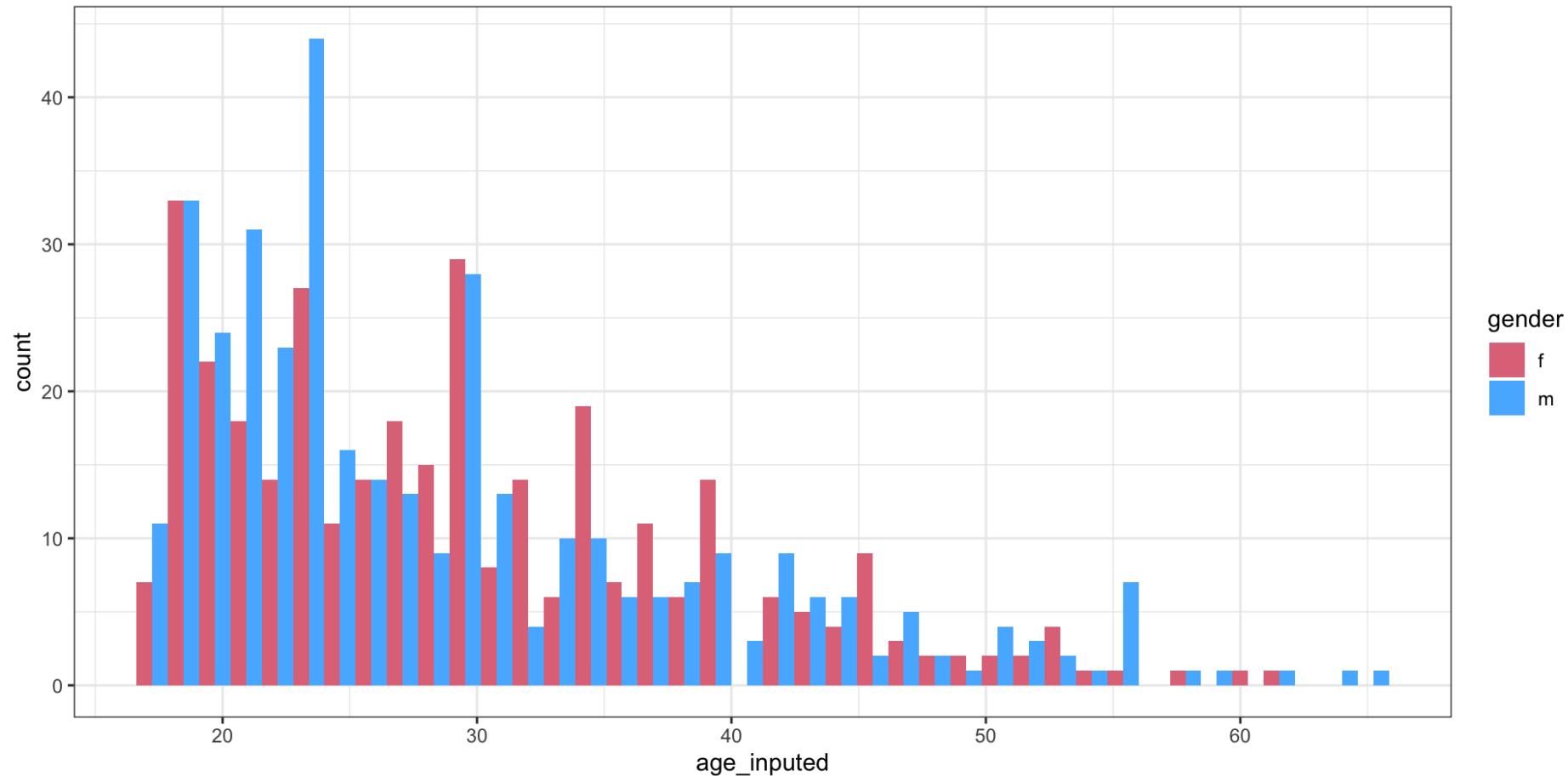


... shifting bars by group

- using the specification `position = 'dodge'` inside `geom_histogram()`

```
1 # trying to improve readability
2 autism_pids %>%
3   ggplot(mapping = aes(x = age_imputed, fill = gender)) +
4   # bars next to each other with `position = 'dodge'` 
5   geom_histogram(bins=40, position = 'dodge') +
6   scale_fill_manual(values = c("#e07689", "#57b7ff")) +
7   scale_color_manual(values = c("#9b2339", "#005ca1")) +
8   theme_bw()
```

... shifting bars by group



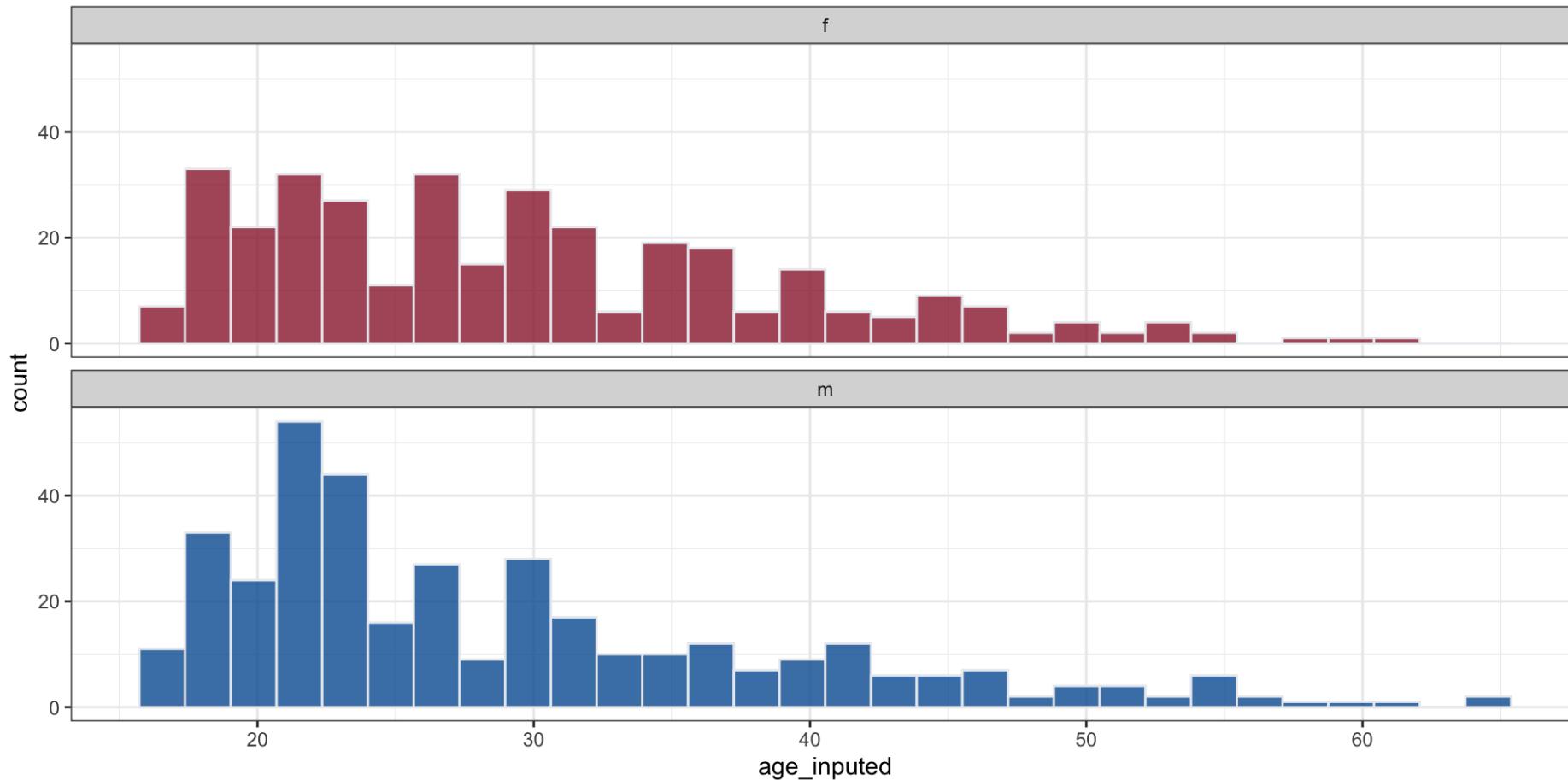
...facet by gender

That's still not very easy to digest. Instead of only filling, you can separate the data into multiple plots to improve readability

- adding `facet_wrap()` with the specification of `~categ_var`
- also `ncol = 1` requires the subplot to be in 1 column

```
1 autism_pids %>%
2   ggplot(aes(x = age_inputed, fill = gender)) +
3   geom_histogram(color="#e9ecf", alpha=0.8, position = 'dodge') +
4   theme_bw() +
5   # splitting the gender groups, specifying `ncol` to see one above the other
6   facet_wrap(~gender, ncol = 1) +
7   scale_fill_cyclical(values = c("#9b2339", "#005ca1"))
```

...facet by gender



... adding 2 mean/median vert lines (by gender)

I want to see the mean vertical line for each of the subgroups, but in this case, I need to create a small dataframe of summary statistics (`group_stats`).

I do so by using `dplyr` add a column `mean_age` with the group mean

```
1 group_stats <- autism_pids %>%
2   dplyr::group_by(gender) %>%
3   dplyr::summarize(mean_age = mean(age_inputed),
4                     median_age = median (age_inputed))
5
6 group_stats
```



```
# A tibble: 2 × 3
  gender mean_age median_age
  <fct>    <dbl>     <dbl>
1 f          29.6      28
2 m          29.0      26
```

(Introducing `tidyr::pivot_longer`)

First, I need to reformat `group_stats`, changing it from a “wide” format with each variable in its own column to a “long” format, where you use one column for all the *measures names* and another for all the *measures values*.

- This can be done using `tidyr::pivot_longer` function

```
1 group_stats_long <- group_stats %>%
  2   tidyr::pivot_longer(cols = mean_age:median_age,
  3                         names_to = "Stat",
  4                         values_to = "Value") %>%
  5   dplyr::mutate(label = as.character(glue::glue("{gender}_{Stat}")))
  6
  7 group_stats_long
```



```
# A tibble: 4 × 4
  gender Stat      Value label
  <fct>  <chr>    <dbl> <chr>
1 f      mean_age  29.6 f_mean_age
2 f      median_age 28   f_median_age
3 m      mean_age  29.0 m_mean_age
4 m      median_age 26   m_median_age
```

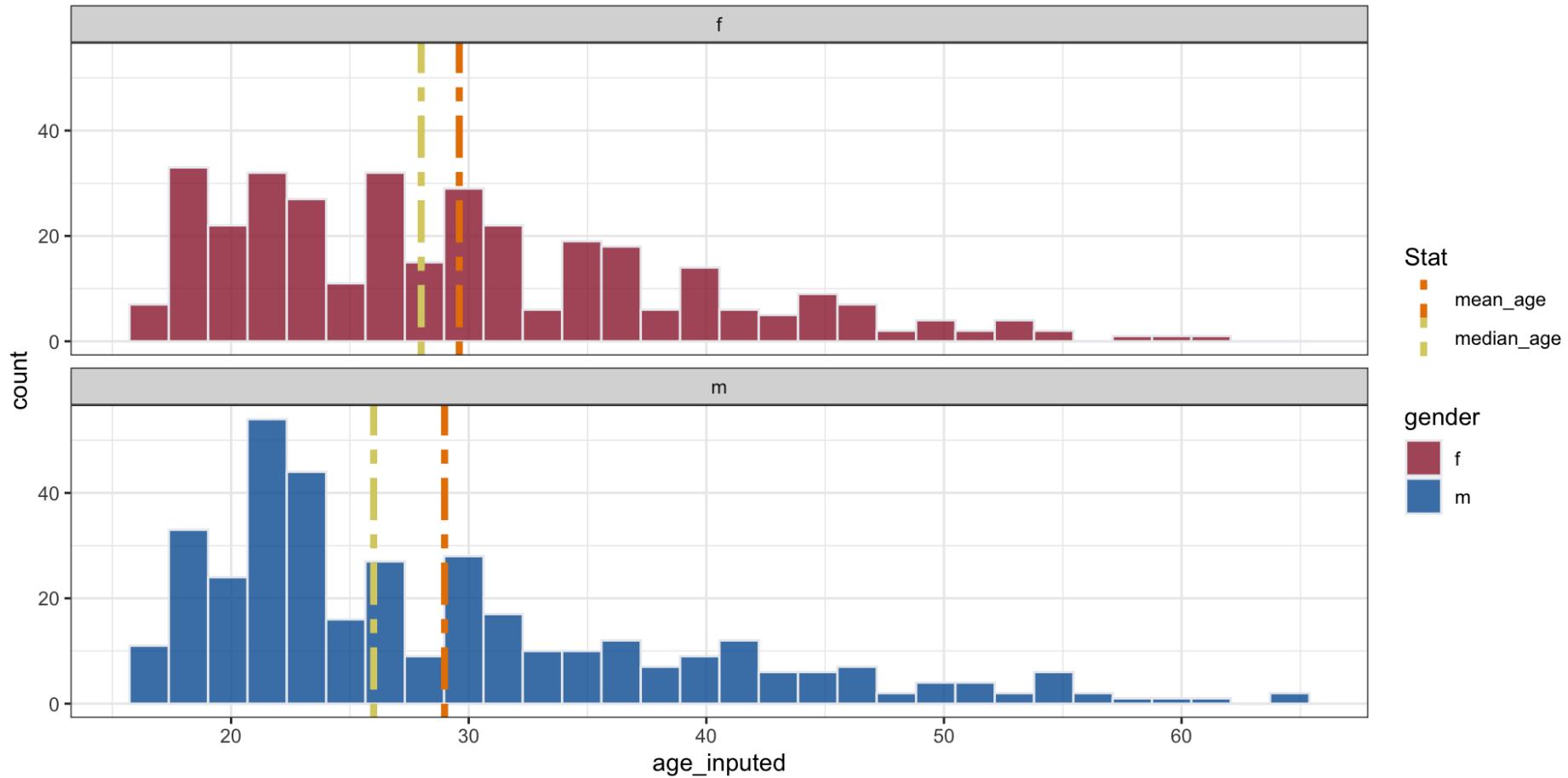
...facet by gender + vert lines by group

Notice that now the plot will have 2 **data** sources:

- **autism_pids**
- **group_stats_long**

```
1 autism_pids %>%
2   ggplot(aes(x = age_inputed, fill = gender)) +
3   # geom_histogram from dataframe 1
4   geom_histogram(bins=30,color="#e9ecef", alpha=0.8, position = 'dodge') +
5   facet_wrap(~gender, ncol = 1) +
6   scale_fill_manual(values = c("#9b2339","#005ca1")) +
7   # geom_vline from dataframe 2
8   geom_vline(data = group_stats_long,
9             mapping = aes(xintercept = Value, color = Stat),
10            lwd=1.5,
11            linetype=6) +
12   scale_color_manual(values = c( "#e68000", "#d8cf71")) +
13   theme_bw()
```

...facet by gender + vert lines by group



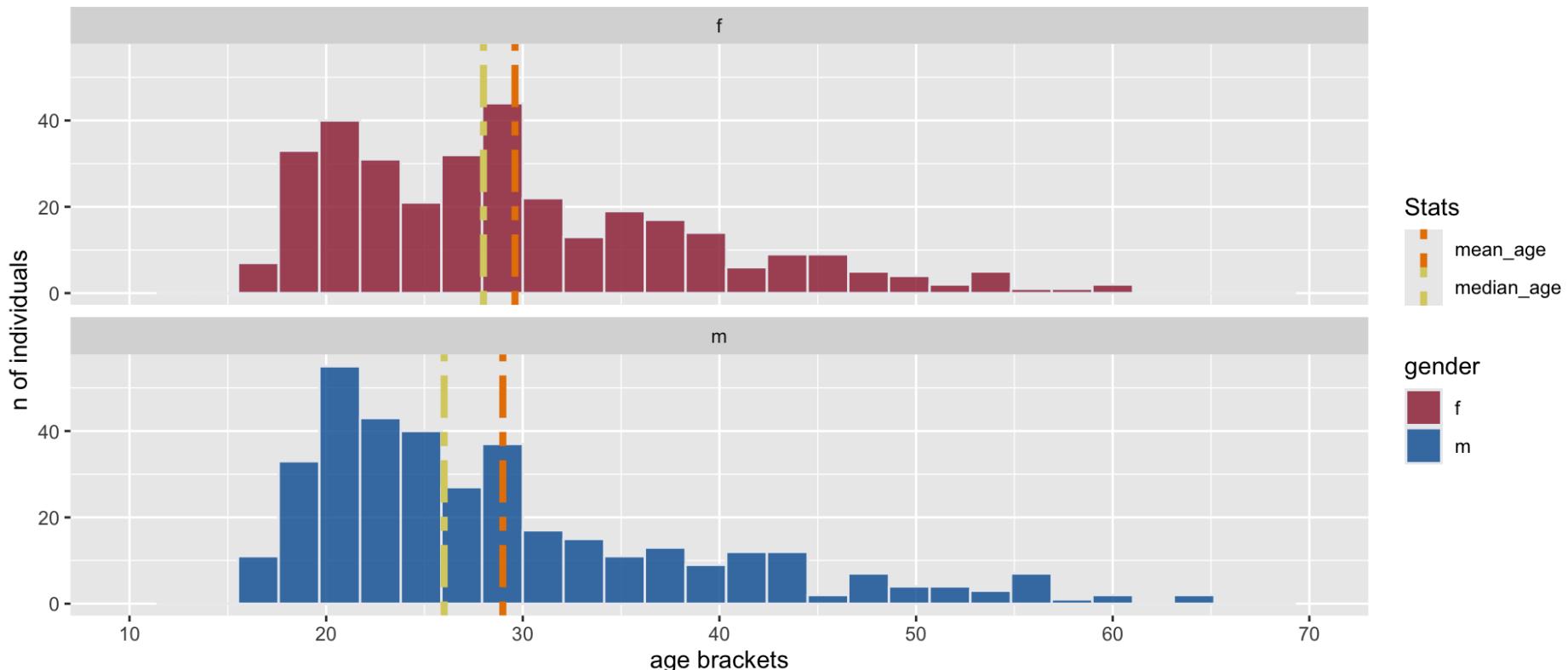
... finishing touches

- using `labs()` and `theme()` layers

```
1 hist_plot <- autism_pids %>%
2   ggplot(aes(x = age_inputed, fill = gender)) +
3   # geom_histogram from dataframe 1
4   geom_histogram(bins=30,color="#e9ecef", alpha=0.8, position = 'dodge') +
5   facet_wrap(~gender, ncol = 1) +
6   scale_fill_manual(values = c("#9b2339","#005ca1")) +
7   # geom_vline from dataframe 2
8   geom_vline(data = group_stats_long,
9             mapping = aes(xintercept = Value, color = Stat),
10            lwd=1.5,
11            linetype=6) +
12   scale_color_manual(values = c( "#e68000", "#d8cf71")) +
13   # increase number of x axis ticks
14   scale_x_continuous(breaks = seq(10, 100,10 ), limits = c(10,70)) +
15   # Additional theme details
16   labs(x = "age brackets", y = "n of individuals",
17         color = "Stats",
18         title = "Distribution of observations by gender",
19         subtitle = "",
20         caption = "Source: Thabtah,Fadi. (2017). Autism Screening Adult. UCI Machine Learnin
21   theme(legend.position = "right",
22         plot.title = element_text(face = "bold"))
23 hist_plot
```

... finishing touches

Distribution of observations by gender



Source: Thabtah,Fadi. (2017). Autism Screening Adult. UCI Machine Learning Repository. <https://doi.org/10.24432/C5F019>.

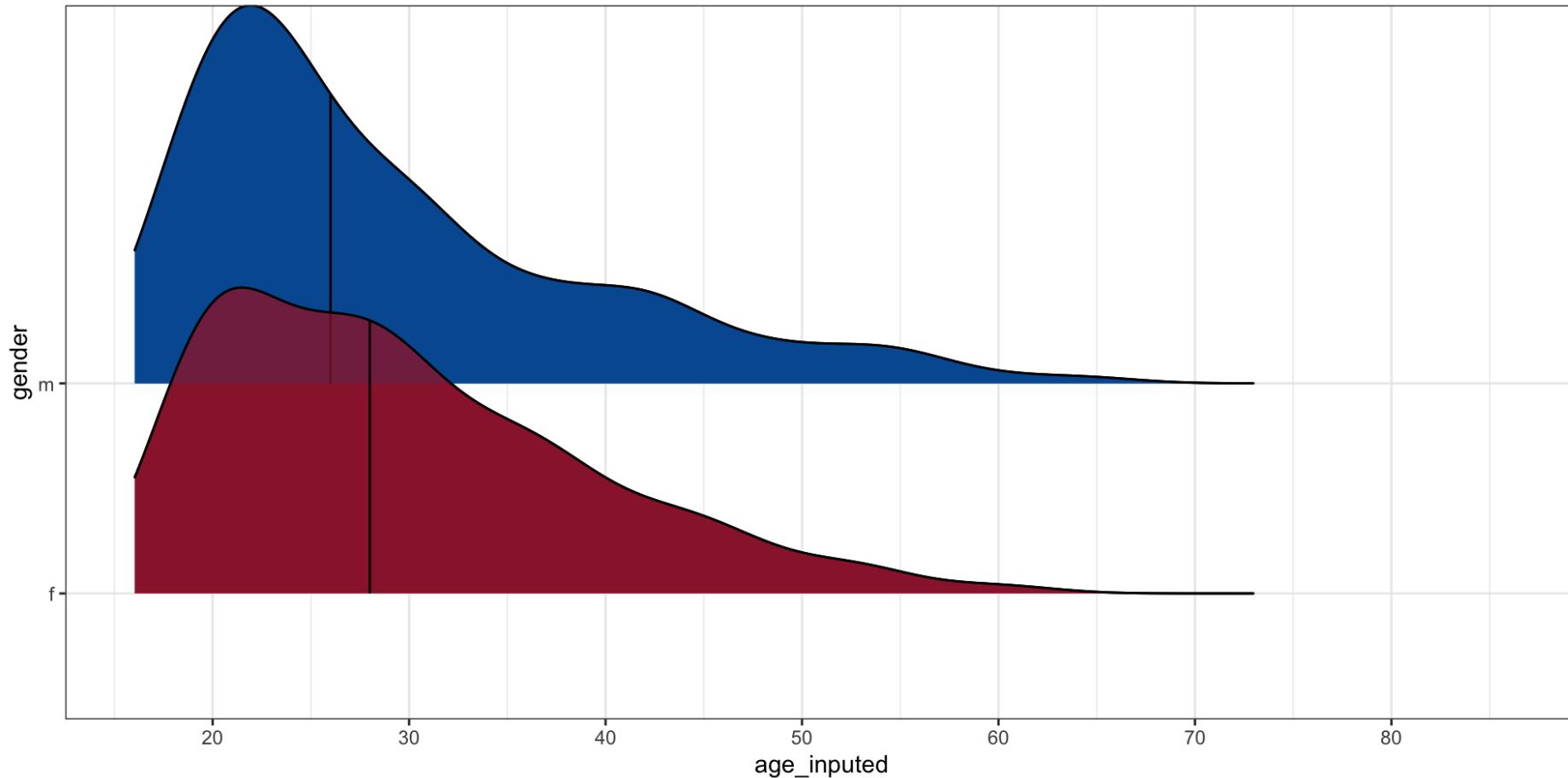
Density ggridges package

As an alternative, you can use the {ggridges} package to make ridge plots. The geom **geom_density_ridges** calculates density estimates from the provided data and then plots those, using the ridgeline visualization.

In this case I have added the median line.

```
1 autism_pids %>%
2   # this takes also `y` = group
3   ggplot(aes(x=age_inputed, y = gender, fill = gender)) +
4   ggridges::geom_density_ridges() +
5   # I can add quantile lines (2 is the median)
6   stat_density_ridges(quantile_lines = TRUE, quantiles = c(0.5), alpha = 0.75) +
7   # increase number of x axis ticks
8   scale_x_continuous(breaks = seq(10, 100, 10 ), limits = c(16, 86)) +
9   scale_fill_cyclical(values = c("#9b2339", "#005ca1")) +
10  theme_bw()
```

Density ggridges package



Barchart

Bar charts provide a visual presentation of categorical data, with `geom_bar()` (height of the bar proportional to the number of cases in each group)

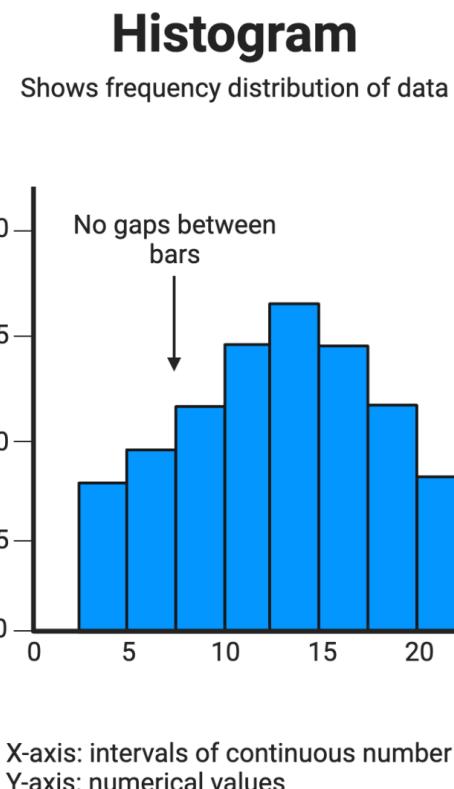
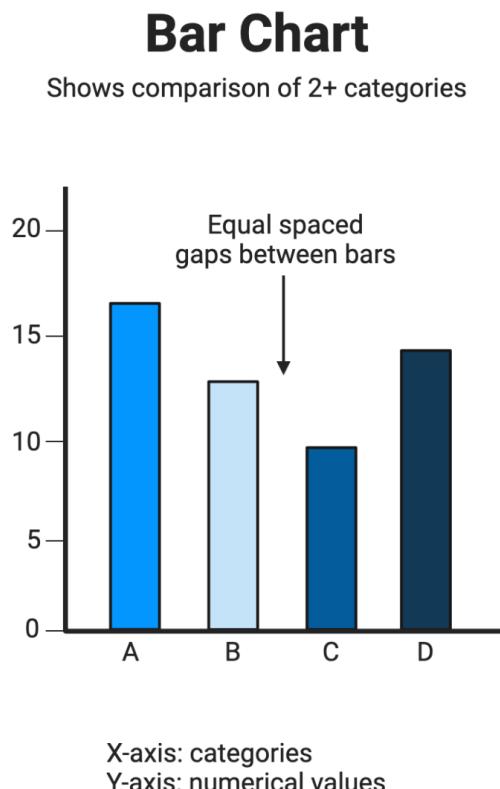


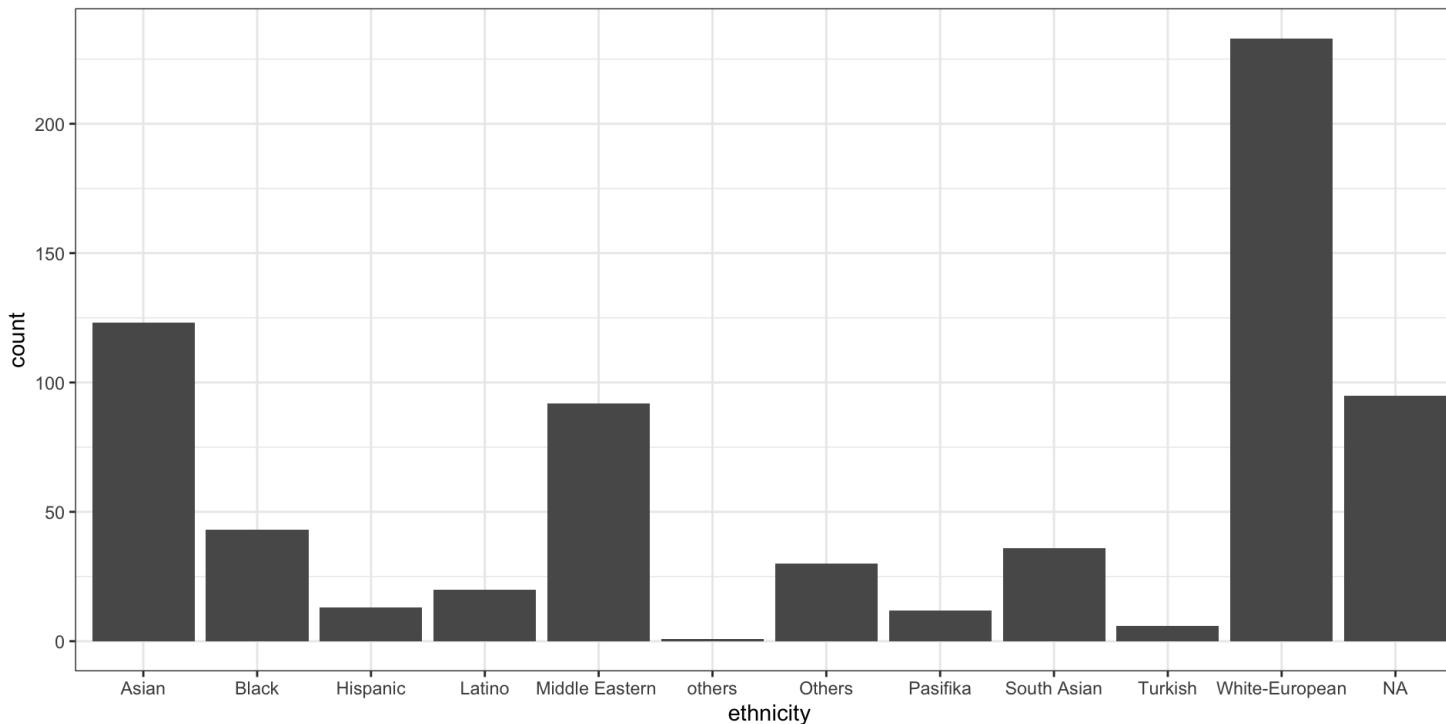
Figure 3: Difference barchart v. histogram Source: <https://www.biorender.com/>

Barchart (cont.)

```
1 # Let's take a variable that we recoded as `factor`  
2 class(autism_pids$ethnicity)
```

```
[1] "factor"
```

```
1 ##### ... no formatting -----  
2 autism_pids %>%  
3   ggplot(aes(x = ethnicity )) +  
4   geom_bar() +  
5   theme_bw()
```

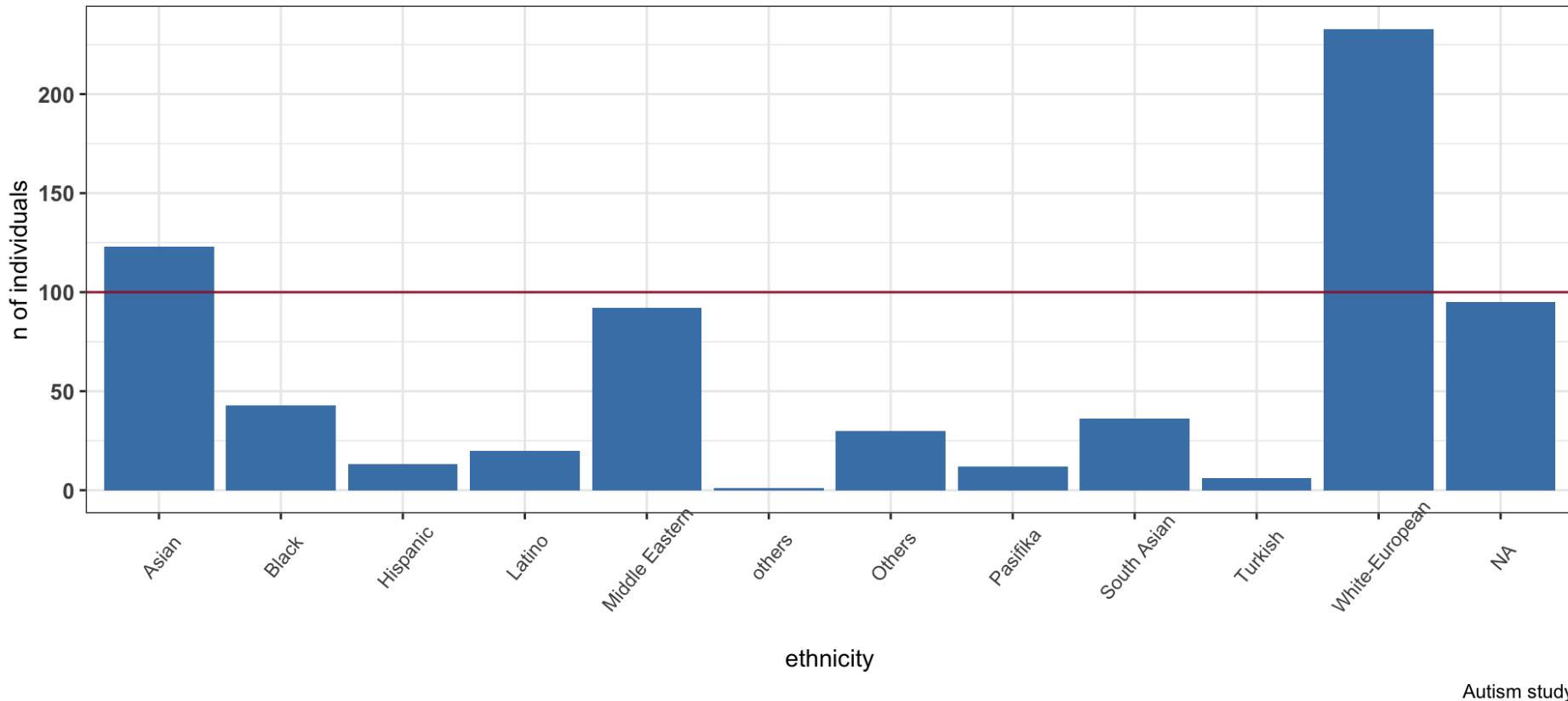


...improve theme

```
1 autism_pids %>%
2   ggplot(aes(x = ethnicity )) +
3   geom_bar(fill = "steelblue") +
4   # reference line
5   geom_hline(yintercept=100, color = "#9b2339", size=0.5, ) +
6   # labels, title, etc
7   labs(x = "ethnicity", y = "n of individuals",
8         color = "Stats",
9         title = "Distribution of observations by ethnicity",
10        subtitle = "",
11        caption = "Autism study") +
12   theme_bw() +
13   # specification son axis labels
14   theme(axis.text.x = element_text(angle=50, vjust=0.75),
15         axis.text.y = element_text(size=10,face="bold"))
```

...improve theme

Distribution of observations by ethnicity



Autism study

...improve readability (reorder bars)

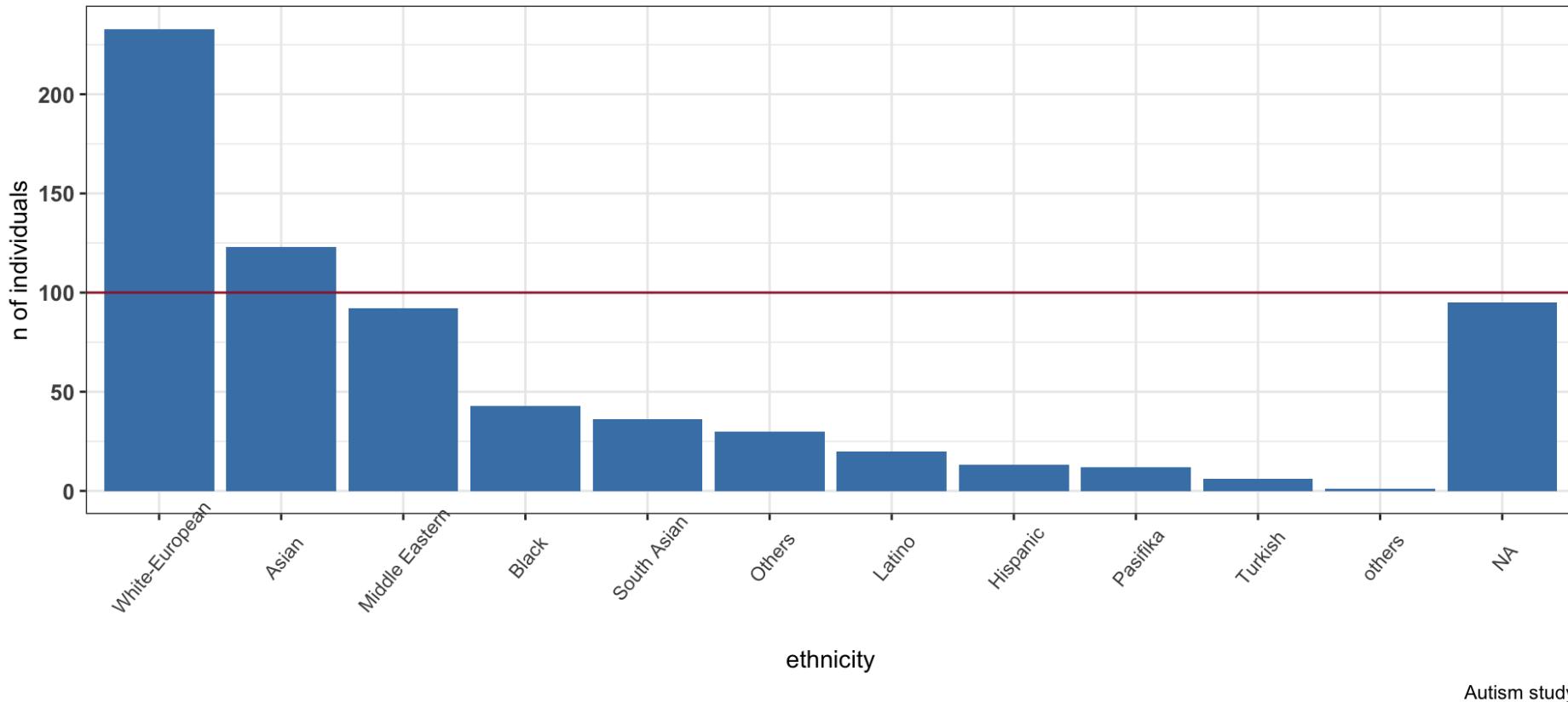
Reordering the bars by count using the package **forcats** and its function **fct_infreq**

- (which we can do because ethnicity is coded as factor)

```
1 autism_pids %>%
2   # we modify our x like so
3   ggplot(aes(x =forcats::fct_infreq(ethnicity ))) +
4   geom_bar(fill = "steelblue") +
5   geom_hline(yintercept=100, color = "#9b2339", size=0.5, ) +
6   labs(x = "ethnicity", y = "n of individuals",
7         color = "Stats",
8         title = "Distribution of observations by ethnicity",
9         subtitle = "",
10        caption = "Autism study") +
11  # --- wrap long x labels (flipped) !!!
12  # scale_x_discrete(labels = function(x) stringr::str_wrap(x, width = 10)) +
13  theme_bw() +
14  theme(axis.text.x = element_text(angle=50, vjust=0.75),
15        axis.text.y = element_text(size=10, face="bold"))
```

...improve readability (reorder bars)

Distribution of observations by ethnicity



...improve readability (highlight NA)

Let's highlight the fact that the last column (**NA**) represents missing values.

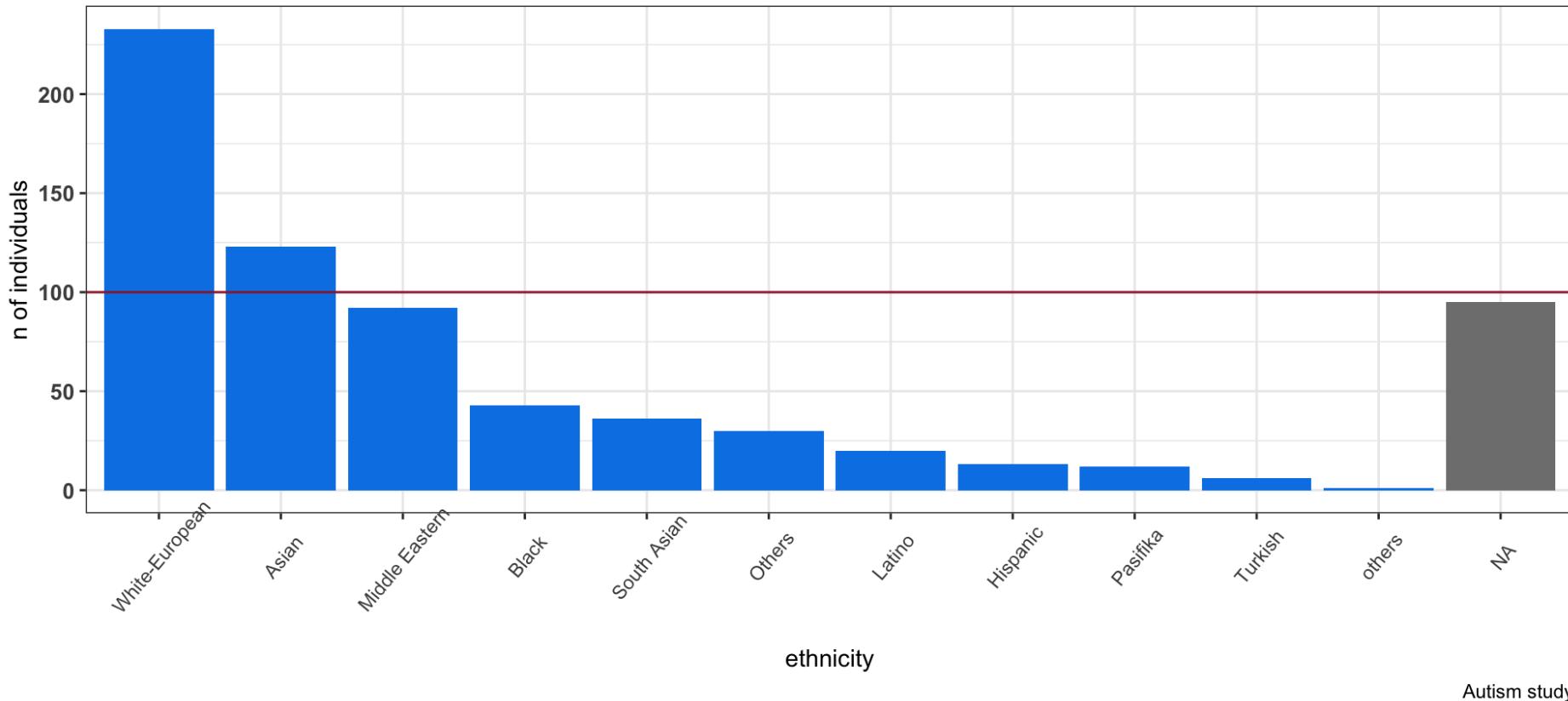
1. We create the **highlight** variable
2. We map color to a variable (**fill = highlight**)

...improve readability (highlight NA) code

```
1 autism_pids %>%
2   ## --- prep the dataframe
3   dplyr::mutate(# Add a factor variable with two levels
4     highlight = forcats::fct_other(ethnicity,
5                               keep = "NA",
6                               other_level = "All Groups")) %>%
7   ## --- now plot
8   # In `aes` mapping we map color to a variable (`fill = highlight`)
9   ggplot(aes(x = forcats::fct_infreq(ethnicity), fill = highlight)) +
10  geom_bar()+
11  # Use custom color palettes
12  scale_fill_manual(values=c("#0084e6")) +
13  # Add a line at a significant level
14  geom_hline(yintercept=100, color = "#9b2339", size=0.5, ) +
15  theme_bw() +
16  # make some more theme specifications
17  labs(x = "ethnicity", y = "n of individuals",
18        color = "Stats",
19        title = "Distribution of observations by ethnicity",
20        subtitle = "",
21        caption = "Autism study") +
22  theme(axis.text.x = element_text(angle=50, vjust=0.75),
23        axis.text.y = element_text(size=10,face="bold")) +
24  theme(legend.position = "none")
```

...improve readability (highlight NA) code

Distribution of observations by ethnicity



Autism study

Boxplot

The boxplot is one of the simplest ways of representing a distribution of a continuous variable and it is packed with information. It consists of two parts:

- **Box** — Extending from the 1st to the 3rd quartile (Q1 to Q3) with a line in the middle that represents the median.
- **Whiskers** — Lines extending from both ends of the box (minimum/maximum whisker values are calculated as $Q1/Q3 -/+ 1.5 * IQR$)
- Everything outside is represented as an **outlier**

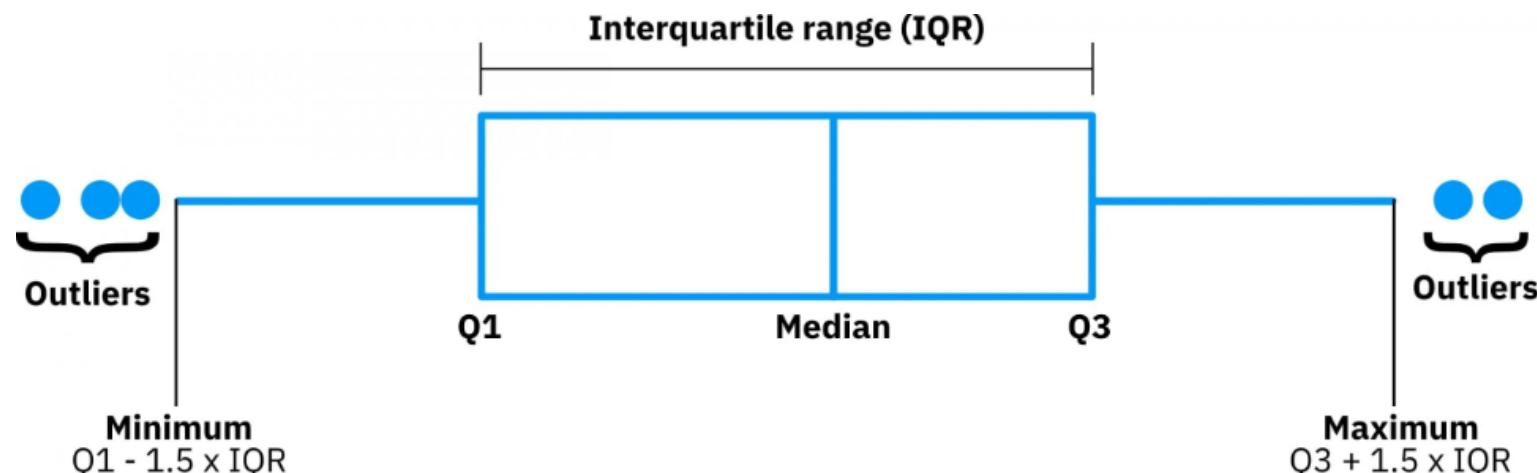


Figure 4: Boxplot Source: <https://www.appspot.com/post/ggplot2-boxplots>

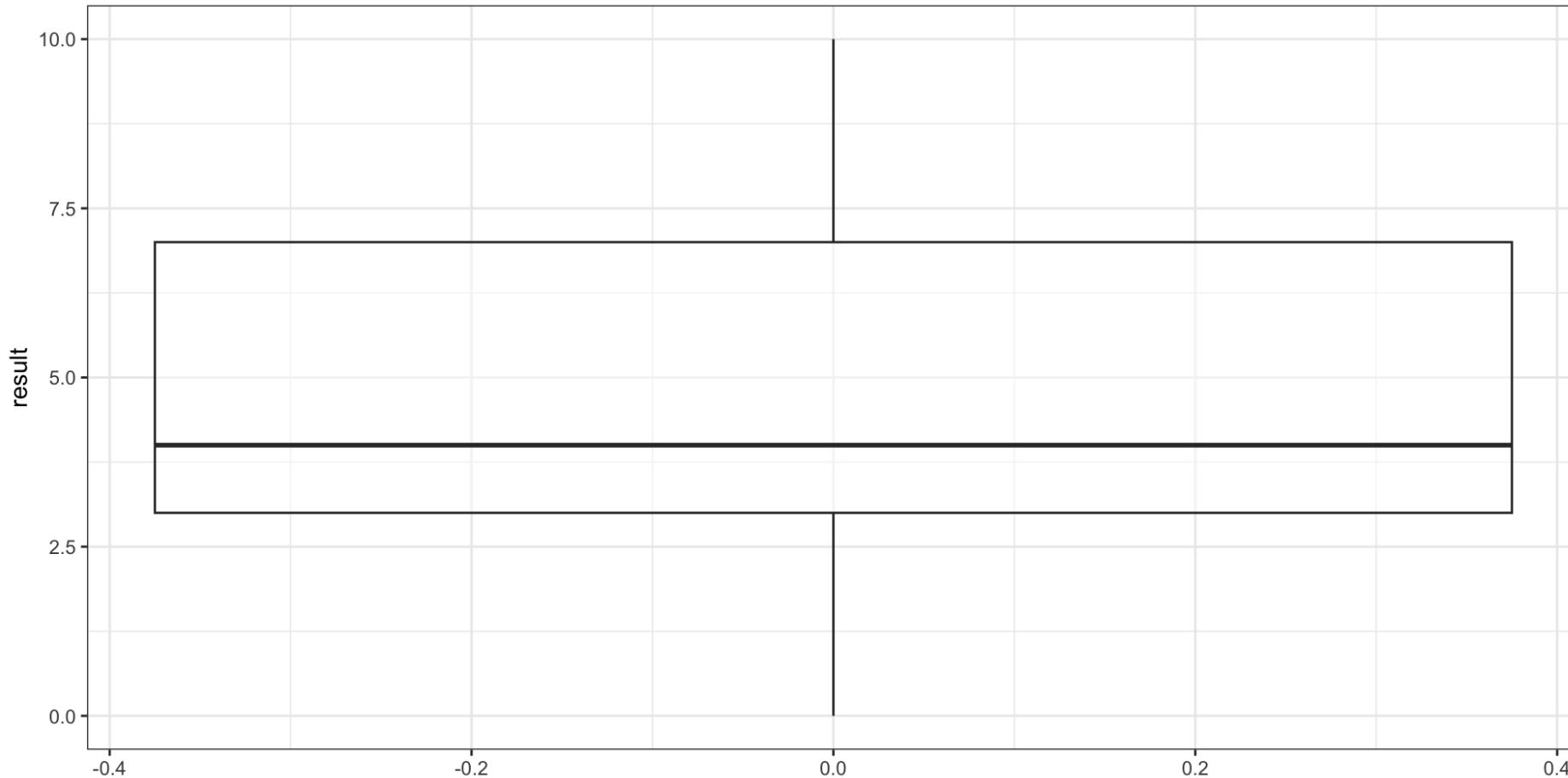
Boxplot example 1

Let's use a boxplot to explore how the continuous variable `result` is distributed in the autism dataset.

- in the aesthetic mapping we specify only `x` (continuous variable)
- switch to vertical orientation with `coord_flip()`

```
1 autism_pids %>%
2   ggplot(aes(x = result )) +
3   geom_boxplot(alpha=0.5) +
4   # switch to vertical orientation
5   coord_flip() +
6   theme_bw()
```

Boxplot example 1



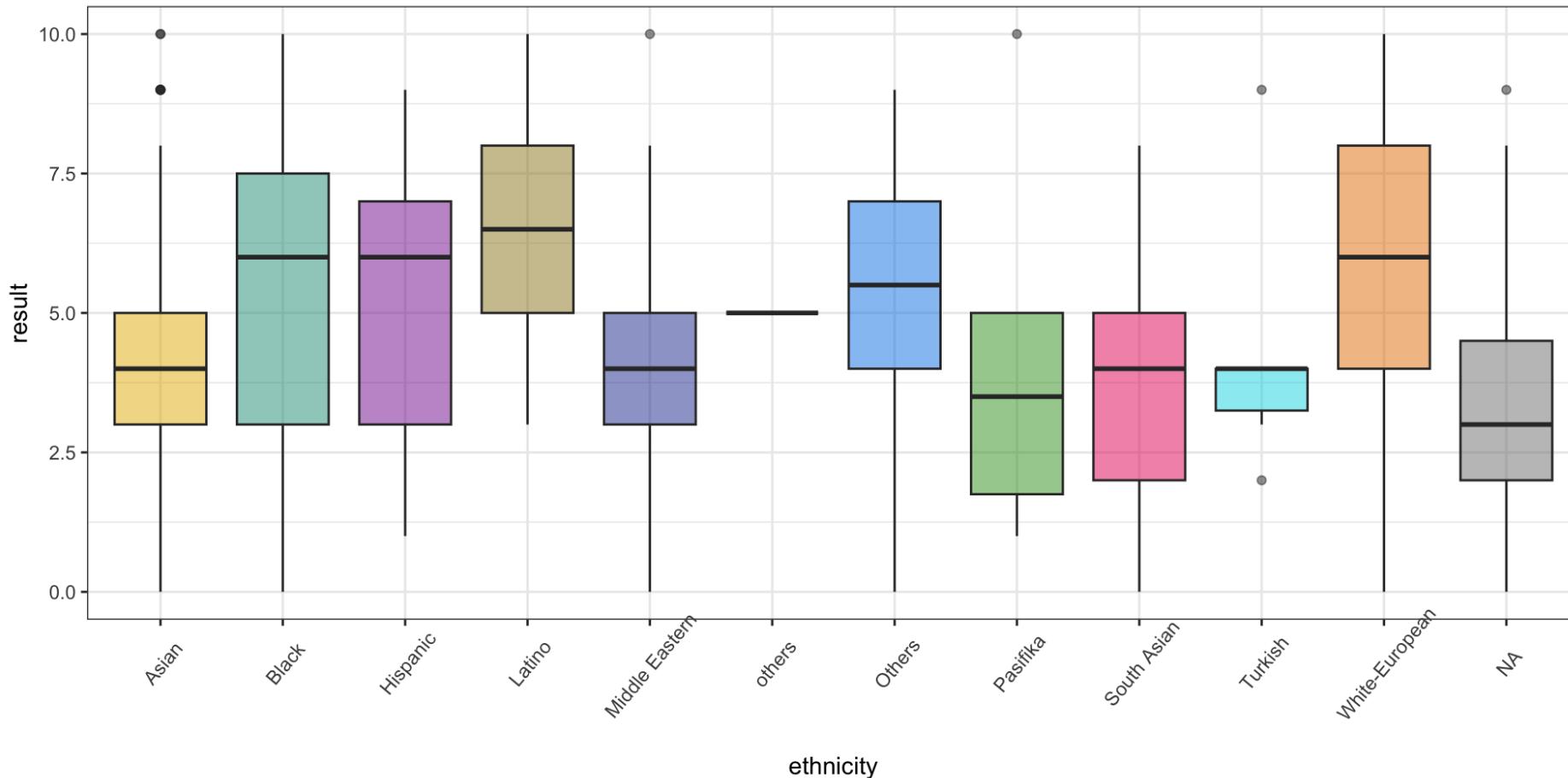
Boxplot example 2

Let's also explore how the continuous variable **result** is distributed by the categorical variable (factor) **ethnicity**.

- in the aesthetic mapping we specify **y** (continuous variable), plus **x** and **fill** (categorical variable)
- make x axis labels readable with **theme(axis.text.x (....))** layer
- I specify colors that I had previously saved in a color palette **contrast_cols_palette**

```
1 autism_pids %>%
2   ggplot(aes(x = ethnicity, y= result, fill = ethnicity)) +
3   geom_boxplot(alpha=0.5) +
4   scale_fill_manual(values = contrast_cols_palette) +
5   theme_bw() +
6   # make x axis labels readable
7   theme(axis.text.x = element_text(angle=50, vjust=0.75)) +
8   # drop legend and Y-axis title
9   theme(legend.position = "none")
```

Boxplot example 2



Violin plot

Similarly, the violin plot is an interesting alternative to show the distribution of a continuous variable along one or more categorical variables. Here, the kernel density plot shows the smoothed curve of the **probability density function (PDF)** of the data.

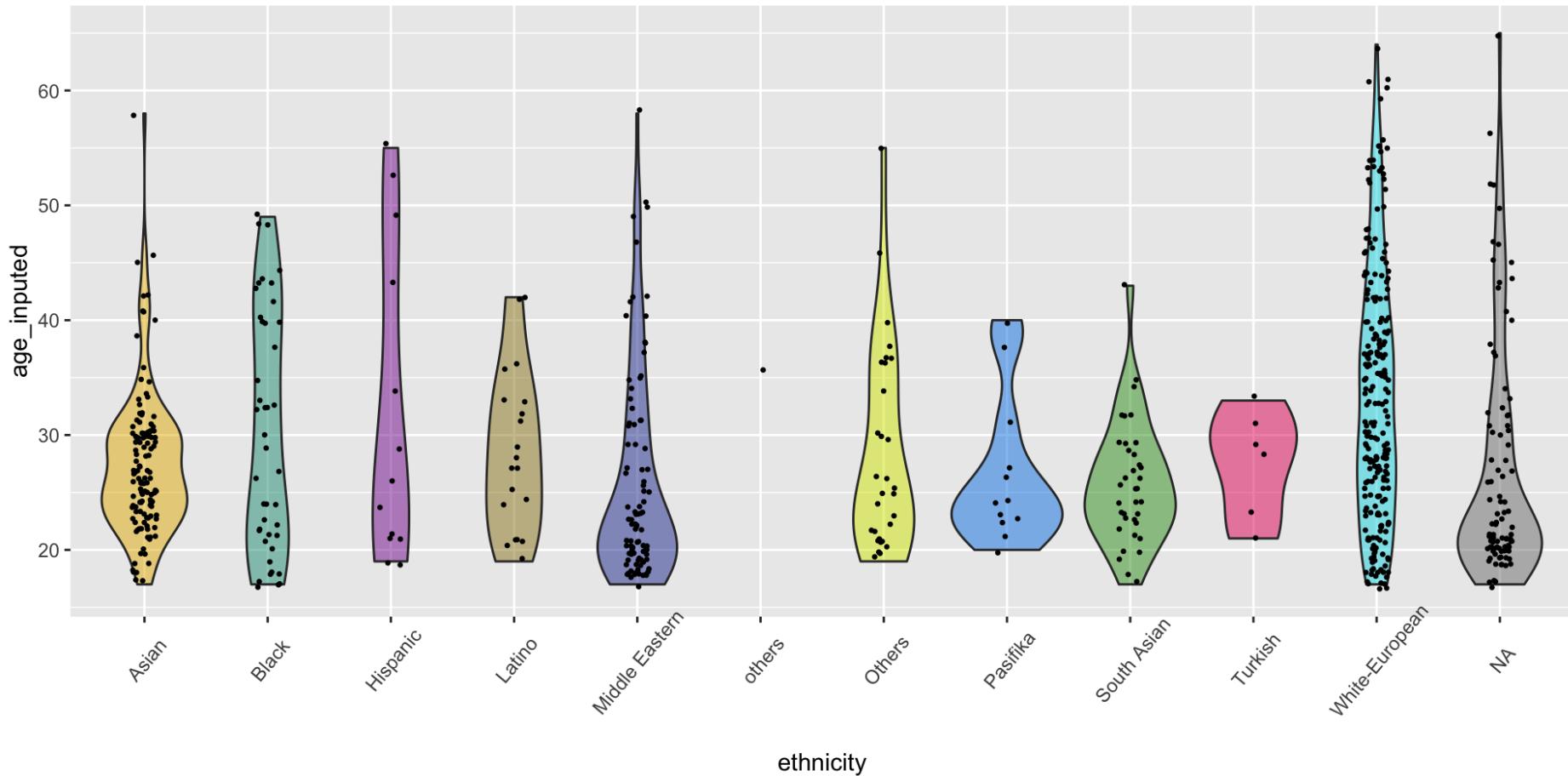
Compared to the box plot, a violin plot provides more information, as it shows not only the summary statistics but also the shape and variability of the data (i.e. helping to identify any **skewness** or **multimodality** in the data).

Violin plot example

- it requires the `geom_violin` function
- it can be enriched by adding with other `geoms`, such as points, lines, or box plots, to create more complex and informative plots
- let's add points with the `geom_point` layer

```
1 autism_pids %>%
2   ggplot(mapping = aes(y = age_inputed, x = ethnicity, fill = ethnicity)) +
3   geom_violin(alpha=0.5) +
4   geom_point(position = position_jitter(width = 0.1), size = 0.5) +
5   scale_fill_manual(values = contrast_cols_palette) +
6   # make x axis labels readable
7   theme(axis.text.x = element_text(angle=50, vjust=0.75)) +
8   # drop legend and Y-axis title
9   theme(legend.position = "none")
```

Violin plot example



SAVING & EXPORTING OUTPUT ARTIFACTS

Saving one plot

If I want to use these output files later, I can easily save in the output folder created at the beginning.

- save a plot with `ggplot2::ggsave`
- specifying the output directory with `here::here(...)`

```
1 ggsave (hist_plot,  
2         filename = here::here("practice", "data_output", "hist_plot.png"))
```

Saving a .Rds data file.

- save a dataframe with `base::saveRDS`
- specifying the output directory with `here::here(...)`

```
1 saveRDS (object = autism_pids,  
2           file = here::here("practice", "data_output", "autism_pids_v2.Rds"))
```

- (later) load a saved dataframe with `base::readRDS`

```
1 # to load it later I will use  
2 readRDS(here::here("practice", "data_output", "autism_pids_v2.Rds"))
```

Final thoughts/recommendations

- Always **read the documentation** (`?package::function`, especially the examples at the bottom)
- Always **inspect the data / variables before and after** making changes
- **Better rename** (i.e. create a new R object) when you recode/manipulate a column or a dataset
 - *(this promotes reproducibility, because you will be able to retrace your coding steps)*
- Always plot distributions for **visual data exploration**
- Make changes in **small increments** (like we saw in building `ggplot2` graph in subsequent layers)