

EPIDEMIOLOGICAL REPORTING WITH

BEST PRACTICE FROM TABLES TO TIME SERIES



The GRAPH Courses, Global Fund & WHO

This book is a compilation of training materials created by the GRAPH Network under a grant from the Global Fund to Fight AIDS, Tuberculosis and Malaria. The materials aim to build global capacity in epidemiological data analysis and decision-making for public health.

Demographic Pyramids for Epidemiological Analysis

Introduction
Learning Objectives
Introducing Demographic Pyramids
The Use of Population Pyramids in Epidemiology
Conceptualizing Demographic Pyramids
Packages
Data Preparation
Intro to the Dataset
Importing Data
Data Inspection
Creating Aggregated Data Subset
Plot Creation
Using <code>geom_col()</code> for demographic pyramids
Plot Customization
Axis Adjustments
Add custom labels
Enhance Color Scheme and Themes
WRAP UP!
Answer Key
References

Introduction

A demographic pyramid, also known as a population pyramid or an age-sex pyramid, helps to visualize the distribution of a population by two important demographic variables: **age** and **sex**.

Today you will learn about the importance of using demographic pyramids to help visualize the distribution of a disease by age and sex and how to create one using `{ggplot2}`.

Let's get into it!

Learning Objectives

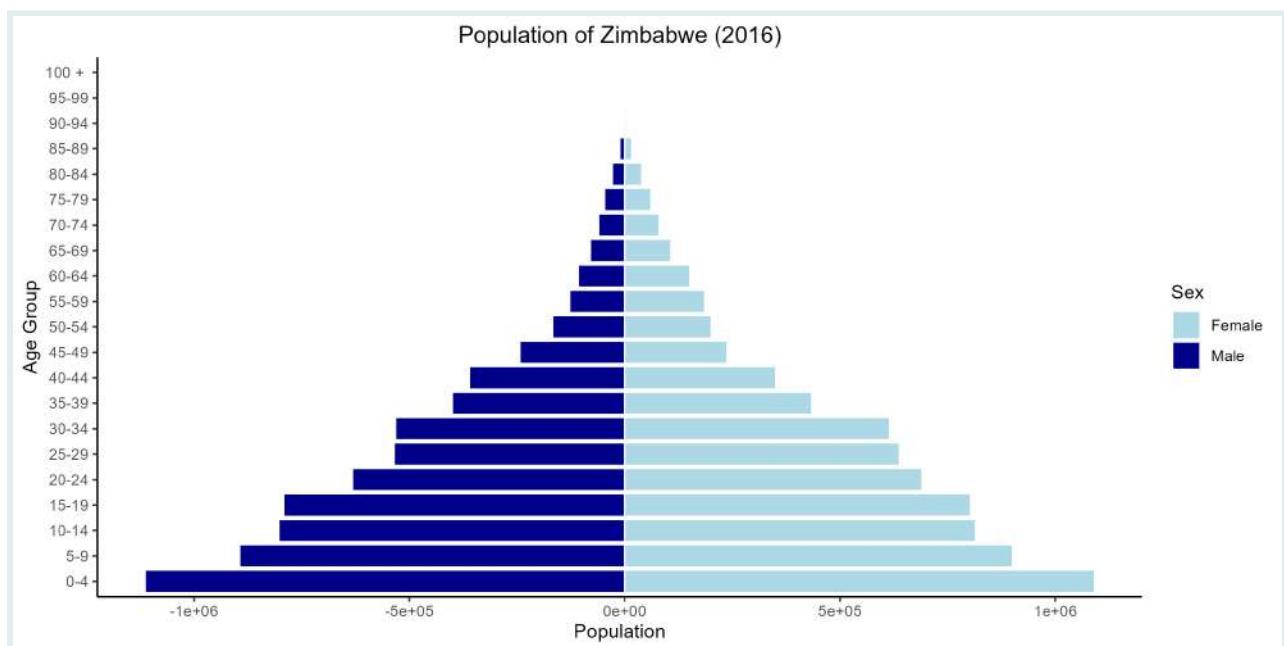
By the end of this lesson, you will be able to:

- Explain **the importance of demographic pyramids** for communicating **age and sex-specific patterns** of disease distribution.
- Understand the **components of a demographic pyramid** and conceptualize it as a modified version of a **stacked bar plot**.

- **Summarize and prepare data** into the appropriate format for plotting with `{dplyr}` functions.
- Use `{ggplot2}` code to **plot a demographic pyramid using `geom_col()`**, showing total counts or percentages on the x axis.
- **Customize the plot** by changing the color scheme, labels, and axis.

Introducing Demographic Pyramids

Population pyramids are graphs that show the distribution of ages across a population. The graph is divided down the center between male and female members of the population, where the y-axis shows the age groups and the x-axis the sex.



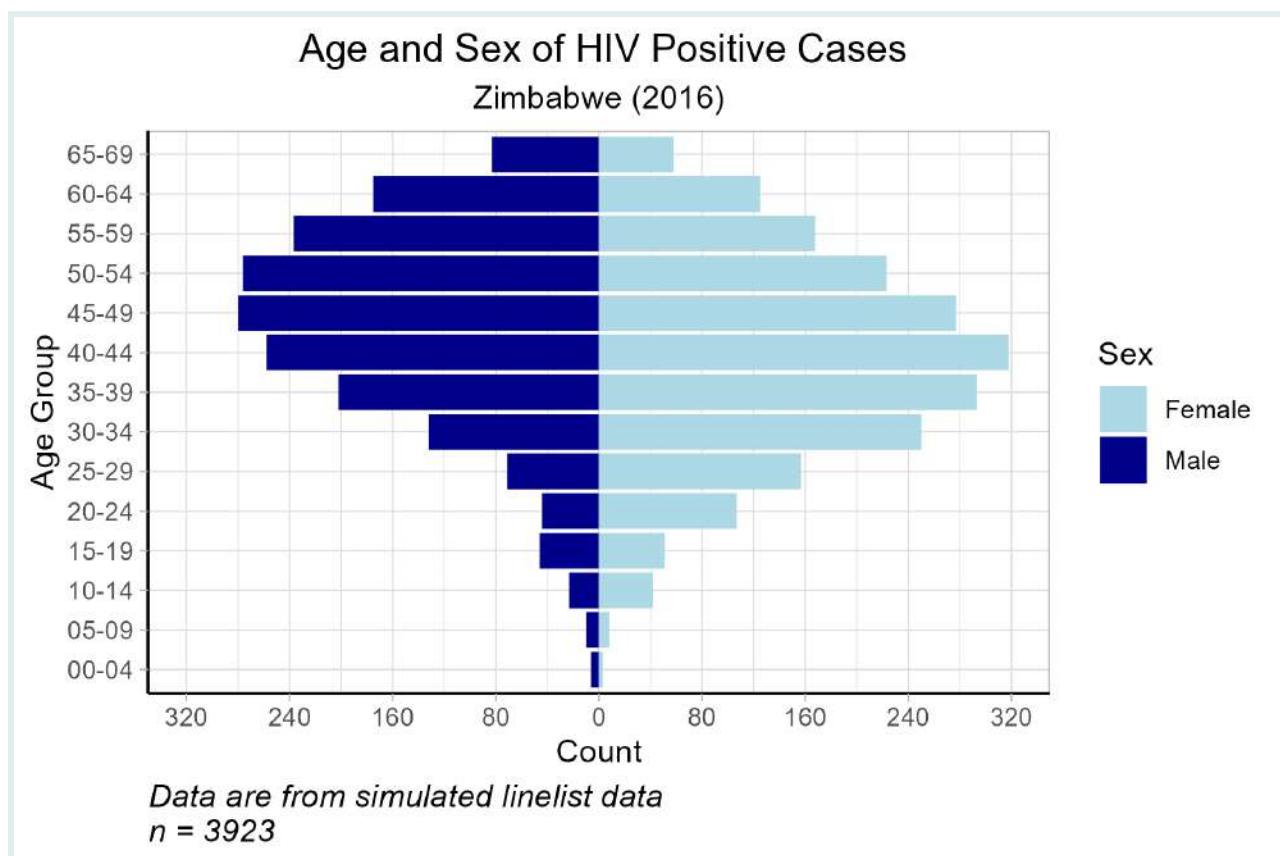
Population pyramid for a country with a young population has a pyramid-like shape.

The overall plot often takes the shape of a pyramid, hence its name.

Three key features of a demographic pyramid are:

1. **Age Groups:** The pyramid is divided into horizontal bars, each representing an age group, often in five-year increments (0-4, 5-9, 10-14, etc.).
2. **Gender Representation:** The left side of the pyramid typically represents males, and the right side represents females, allowing for a quick comparison between the gender distribution within each age group.
3. **Population Size:** The length of the bars indicates the size of the population in each age group. A longer bar suggests a larger population in that age group.

Through the use of `{ggplot2}` we are able to create pyramids while customizing it to our specific needs. At the end of this lesson, our final plot will look like this:



The Use of Population Pyramids in Epidemiology

Demographic pyramids are useful for describing and understanding the epidemiology of various diseases, by visualizing the distribution of disease by age and sex.

Age and Gender-Specific Vulnerability

Different diseases may affect age groups differently. We know that the incidence of certain communicable diseases can vary with age. In the case of tuberculosis (TB) in Africa, adolescents and young adults are primarily affected in the region. However, in countries where TB incidence has decreased significantly, such as the United States, it is mainly seen in older people, or the immunocompromised. Another disease that demonstrates age variation is malaria, where children under the age of 5 years old account for a high majority of deaths in the region of Africa. HIV has been shown to affect females more than males, especially in younger age groups. This could be due to biological vulnerability or social factors.

Therefore, when describing the epidemiology of communicable diseases such as HIV, Malaria, and TB, it is important to observe the distribution of cases or deaths by age group and sex. This information helps inform national surveillance programs by identifying which age and sex groups experience the highest burden, and who to target for intervention.

Using Demographic Distribution for Data Quality Assessment

Demographic pyramids can also play a crucial role in assessing the data quality of routine surveillance systems by helping assess the internal and external consistency.

SIDE NOTE

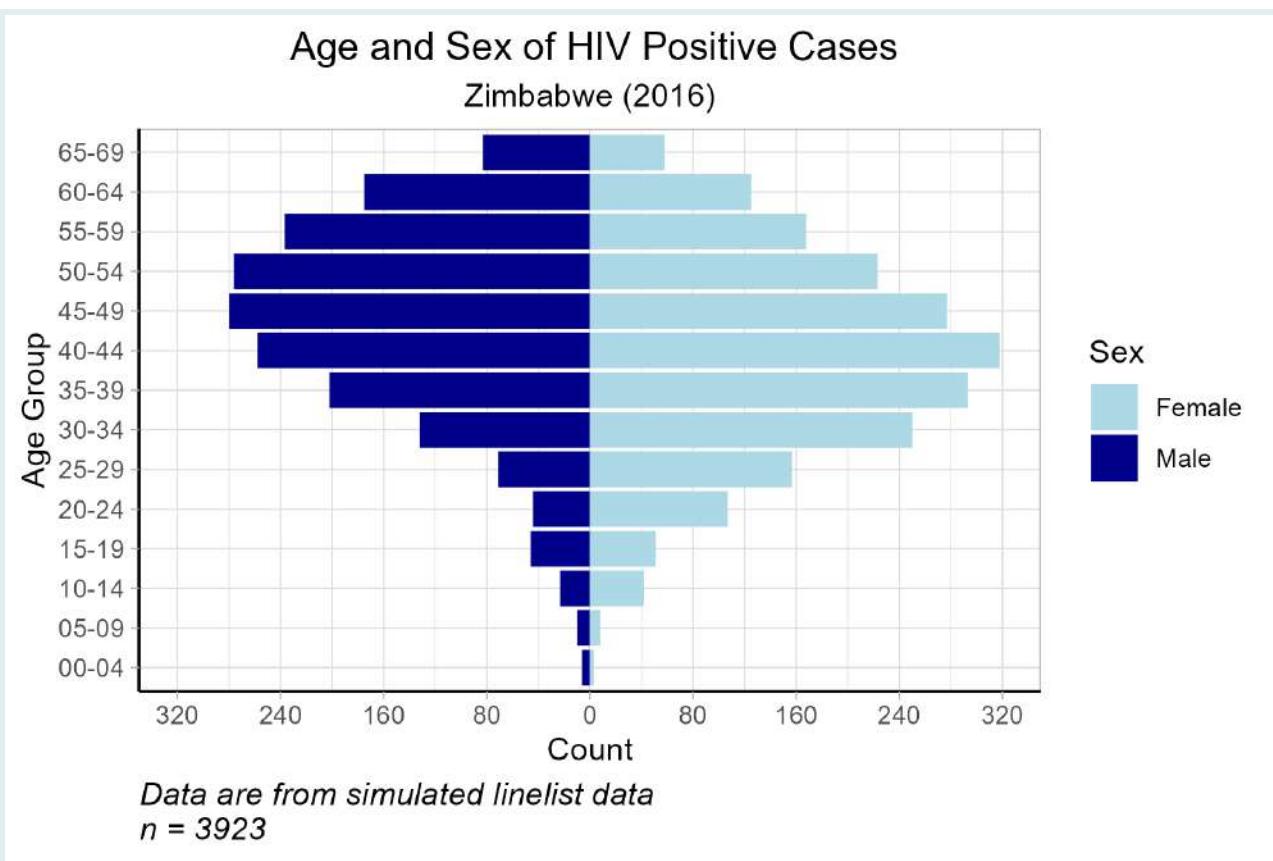


When trying to assess surveillance data quality standards of certain diseases, external consistency can be evaluated by comparing the national surveillance data with the global epidemiology of that disease. Data calculations based on demographic variables such as age group are sometimes used.

For the instance of TB surveillance data, external consistency can be evaluated by calculating the percentage of children diagnosed with TB within the program and comparing it with the global average cases.

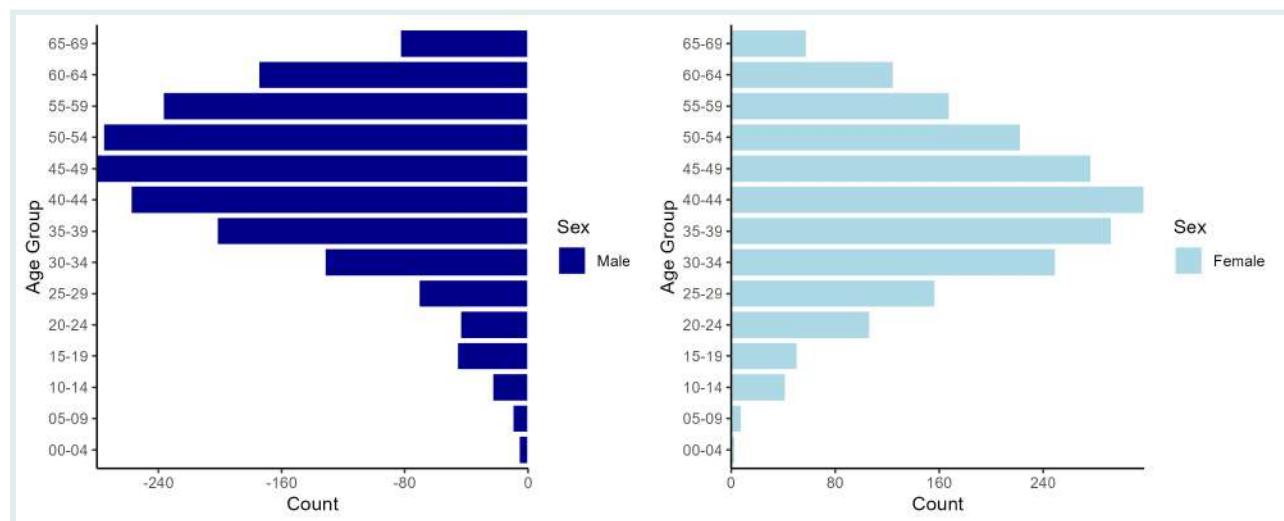
Conceptualizing Demographic Pyramids

Let's take a closer look at our target demographic pyramid, and break down how it can be graphed using `geom_col()` from `ggplot2`.



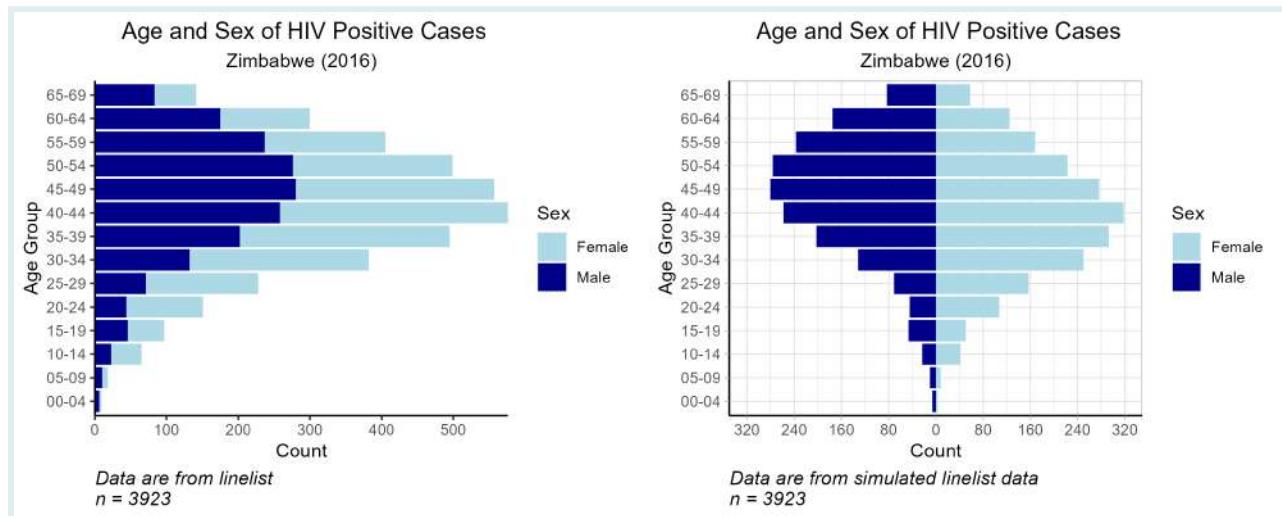
As you can see, the x-axis is divided in two halves (males and females), which are plotted in opposite directions starting at 0. The units on the x-axis are symmetrical on either side, and the age groups are labelled along the y-axis.

In other words, we can think of it two **bar graphs** – one for males and one for females.



A demographic pyramid can also be conceptualized as a specialized form of a stacked bar chart. In a traditional stacked bar chart, the segments are piled on top of one another, starting from a baseline of zero and extending outward.

In contrast, a demographic pyramid aligns the bars back-to-back along a central axis. This axis represents the point of division between two categories (male and female populations) with the bars extending in opposite directions to illustrate the proportion of each age group within these categories.



A demographic pyramid is akin to a modified stacked bar chart with two key distinctions:

KEY POINT



- 1. Central Axis:** Instead of stacking the bar segments on top of each other, bars are aligned back-to-back along a **central axis**, which acts as the point of division between male and female populations.
- 2. Bar Orientation:** Unlike traditional stacked bars that build from zero outward in one direction, demographic pyramid bars extend **in opposite directions** to represent the size of each age group within the categories.

Packages

This lesson will require the following packages to be installed and loaded:

```
# Load packages
pacman:::p_load(here,          # to locate files
                tidyverse,      # to wrangle and plot data (includes ggplot2)
                apyramid)       # package dedicated to creating age pyramids
```

REMINDER

Every lesson from The GRAPH Courses comes with a code-along RMarkdown script, which you should download and use to follow along with the lesson video or lesson notes.

Data Preparation

Intro to the Dataset

For this lesson, we will use a simulated HIV dataset imitating a linelist of HIV cases in Zimbabwe during 2016. For this specific lesson, we will focus on the **age-related** and **sex** variables to create our demographic pyramid.

Importing Data

Let's start by importing our data into our RStudio environment and taking a closer look at it to better understand the variables we will be using for the creation of our demographic pyramid.

```
hiv_data <- read_csv(here::here("data/hiv_zw_linelist_2016.csv"))  
hiv_data
```

Our imported dataset contains **28000** rows and **3** columns containing the `age_group` and `sex` variables we will be using for the creation of our demographic pyramid. Each line (row) corresponds to one patient, while each column represents different variables of interest. The linelist only contains demographic and HIV-related variables (HIV status). In addition, the `hiv_status` variable provides us with information on the status of individuals (*positive* or *negative*).

Since we are interested on creating a demographic pyramid on HIV prevalence, we first need to filter for HIV positive individuals.

Let's filter data!

```
# Create subset with only HIV positive individuals  
hiv_cases <- hiv_data %>%  
  filter(hiv_status == "positive")  
  
hiv_cases
```

Notice that we now have data subset of **3923** rows and **3** columns where all of the individuals are **HIV positive!**

Data Inspection

Now, before moving to the creation of our demographic pyramid, let's inspect the data by creating a table summarizing the `age_group` and `sex` columns!

For this step, we will use `count()` from `{dplyr}`.

```
hiv_cases %>%  
  count(age_group, sex)
```

We can see that the data is clean and the `age_group` column is correctly organized in ascending order (youngest to oldest).

Always verify your data's order before plotting a demographic pyramid!

WATCH OUT



Before creating your demographic pyramid, make sure to check that your data is clean and correctly organized in **ascending order!** This is important when using categorical variables as the order of your `age_group` will affect the order it will be plotted in your pyramid.

In the case of demographic pyramids, we want the youngest age group to be located at the bottom of the y-axis and the oldest age group to be at the top of the y-axis.

Creating Aggregated Data Subset

Before we get started, we need to create an aggregated data frame that calculates the total number cases per age group, divided by sex. We want the aggregated data frame to look like this:

# A tibble: 28 × 5					
	age_group	sex	total	axis_counts	axis_percent
	<chr>	<chr>	<int>	<int>	<dbl>
1	00-04	female	3	3	0.1
2	00-04	male	6	-6	-0.2
3	05-09	female	8	8	0.2
4	05-09	male	10	-10	-0.3
5	10-14	female	42	42	1.1
6	10-14	male	23	-23	-0.6
7	15-19	female	51	51	1.3
8	15-19	male	46	-46	-1.2
9	20-24	female	107	107	2.7
10	20-24	male	44	-44	-1.1

(we have 14 age groups and two sexes, therefore 28 rows. We'll first calculate the sum of each group - that's the `total` column, then we'll negate the male values and create a the new column called `axis_counts`, then add the last column, `axis_percent`, which presents the totals as percentages. They're called axis counts bc this is purely for plotting purposes. We don't actually have -6 cases, that's not possible.)

The reason `male` values are negated is to plot the male bars on the *left side* of the axis!

KEY POINT



When using the `geom_col()` function, the count for each group **needs to be pre-calculated** and specified in `aes()` as the `x` or `y` variable. In other words, you will need to convert linelist data into summary table with the aggregated number of occurrences for each categorical level. If your data is pre-aggregated, you can skip this aggregation step.

Let's start by using `{dplyr}` to create the summarized data frame, with the 28 rows and 5 column we saw above! We'll use the `group_by()` and `summarise()` functions to do this.

REMINDER



Don't forget to negate the male counts in order to obtain the male bar plot on the *left side* of the graph! We can do this with the `mutate()` function.

	age_group	sex	total	axis_counts	axis_percent
	<chr>	<chr>	<int>	<int>	<dbl>
1	00-04	female	3	3	0.1
2	00-04	male	6	-6	-0.2
3	05-09	female	8	8	0.2
4	05-09	male	10	-10	-0.3
5	10-14	female	42	42	1.1
6	10-14	male	23	-23	-0.6
7	15-19	female	51	51	1.3
8	15-19	male	46	-46	-1.2
9	20-24	female	107	107	2.7
10	20-24	male	44	-44	-1.1

```
# Create new subset
pyramid_data <-
  hiv_cases %>%
    # Count total cases by age group and gender
    count(age_group, sex, name = "total") %>%
    # Create new columns for x-axis values on the plot
    mutate(
      # New column with axis values - convert male counts to negative
      axis_counts = ifelse(sex == "male", -total, total),
      # New column for percentage axis values
      axis_percent = round(100 * (axis_counts / nrow(hiv_cases)),
                           digits = 1))
head(pyramid_data)
```

Notice that the male values in the `axis_counts` and `axis_percent` columns are **negative!**

Now that the data is summarized and in the appropriate format, we can use our new `pyramid_data` data frame to plot the pyramid with `{ggplot2}`!



PRACTICE Let's test your understanding with the following multiple-choice question:

- When preparing data for plotting with `geom_col()`, what modification must be made to the count values?
 - All total counts must be negated.
 - Counts must be multiplied by 2.



- c. Counts must be converted into percentages.
- d. Male counts must be negated (multiplied by -1).

Answer key can be found at the end of this document.

Plot Creation

As we mentioned earlier, a demographic pyramid can be thought of as a modified version of a stacked bar plot.

To create a basic stacked bar plot with `geom_col()`, we plot a categorical variable (e.g., `age_group`) against a continuous variable (e.g., `total`), and set `fill` color to a second categorical variable (e.g., `sex`).

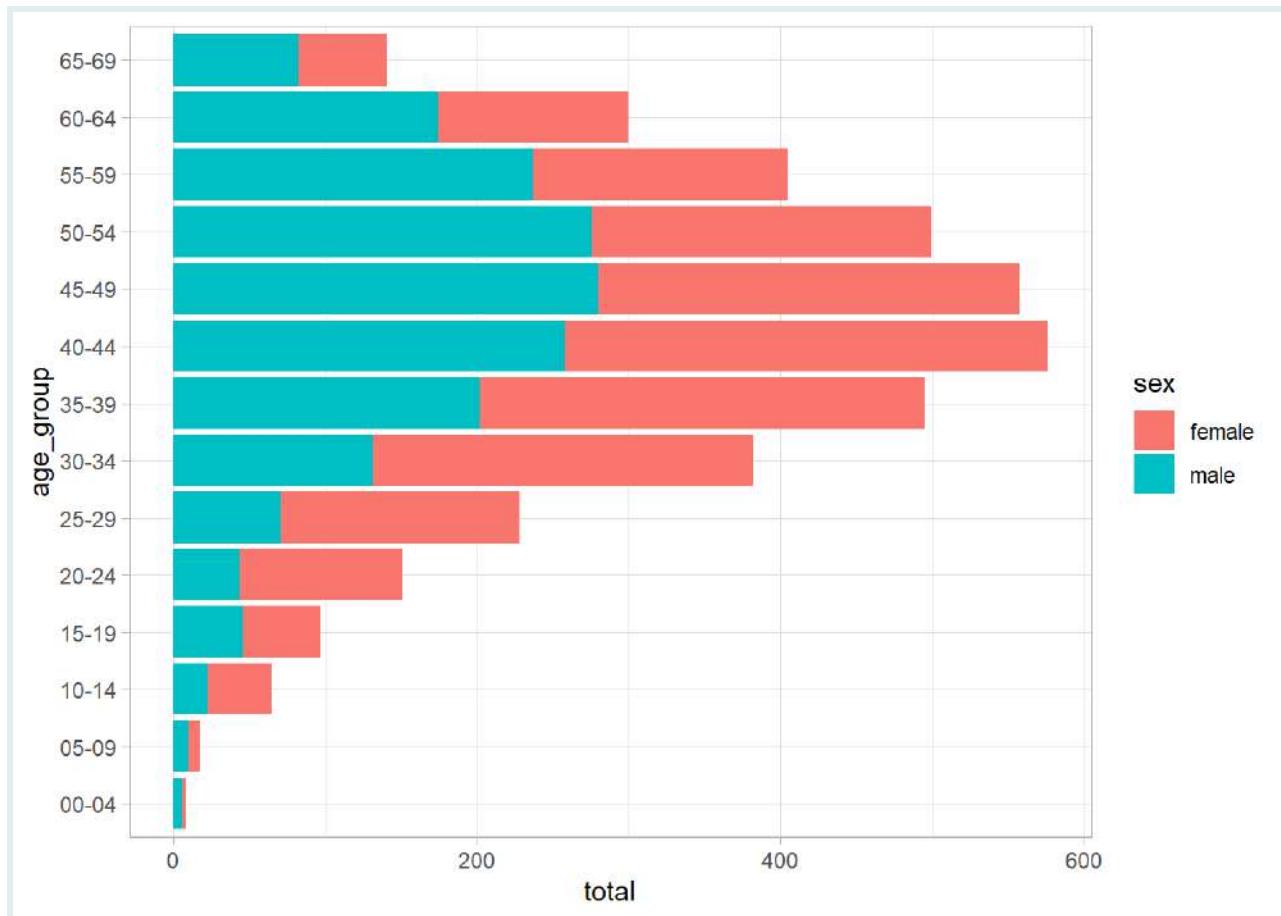
```
# Basic stacked bar plot

# Begin ggplot
ggplot() +

# Create bar graph using geom_col()
geom_col(data = pyramid_data,      # specify data to graph
          aes(x = age_group,       # indicate categorical x variable
              y = total,           # indicate continuous y variable
              fill = sex)) +      # fill by second categorical variable

# Modify theme
theme_light() +

# Flip X and Y axes
coord_flip()
```



Here we used the `total` variable from `pyramid_data`, where all counts are positive.

	age_group	sex	total	axis_counts	axis_percent
	<chr>	<chr>	<int>	<int>	<dbl>
1	00-04	female	3	3	0.1
2	00-04	male	6	-6	-0.2
3	05-09	female	8	8	0.2
4	05-09	male	10	-10	-0.3
5	10-14	female	42	42	1.1
6	10-14	male	23	-23	-0.6
7	15-19	female	51	51	1.3
8	15-19	male	46	-46	-1.2
9	20-24	female	107	107	2.7
10	20-24	male	44	-44	-1.1

RECAP



RECAP

In order to use the `geom_col()` function for a stacked bar chart, your dataset needs to include the total sums aggregated by each categorical variable (`age_group` and `sex`)!

The basics of plotting bar graphs with `{ggplot2}` is covered in our introductory data visualization course, Data on Display. Further applications of bar charts for epidemic reporting are taught in our lesson on Visualizing Comparisons and Compositions. All content can be found on our website.

Using `geom_col()` for demographic pyramids

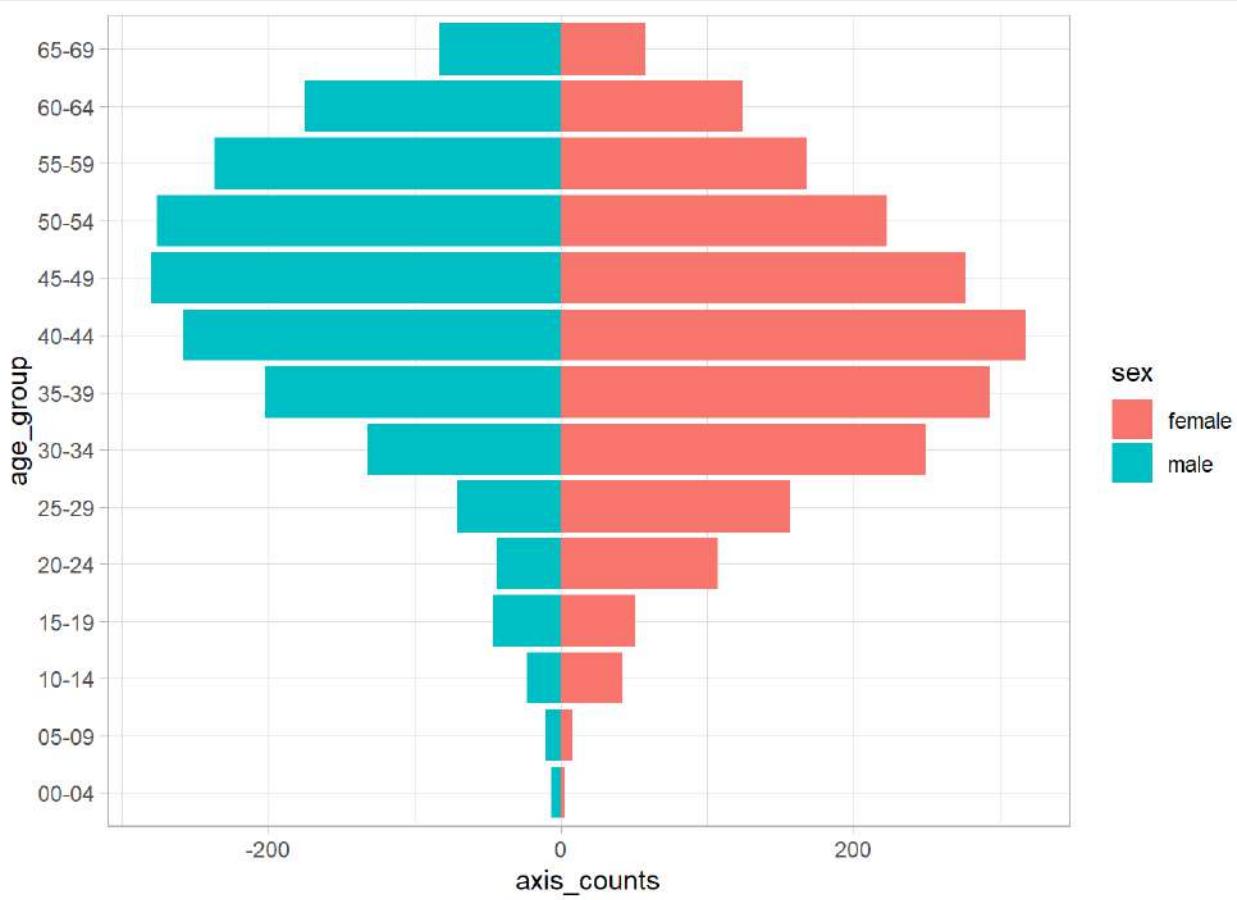
Now we will build on the basic stacked bar code above to create a demographic pyramid.

This time we use `axis_counts` for the y axis, which has negative male counts.

# A tibble: 28 × 5					
	age_group	sex	total	axis_counts	axis_percent
1	00-04	female	3	3	0.1
2	00-04	male	6	-6	-0.2
3	05-09	female	8	8	0.2
4	05-09	male	10	-10	-0.3
5	10-14	female	42	42	1.1
6	10-14	male	23	-23	-0.6
7	15-19	female	51	51	1.3
8	15-19	male	46	-46	-1.2
9	20-24	female	107	107	2.7
10	20-24	male	44	-44	-1.1

```
demo_pyramid <-
  ggplot() +
  geom_col(data = pyramid_data, #specify data to graph
    aes(
      x = age_group,      # indicate x variable
      y = axis_counts,   # indicate NEGATED y variable
      fill = sex)) +    # fill by sex
  theme_light() +
  coord_flip()
```

```
demo_pyramid
```



KEY POINT



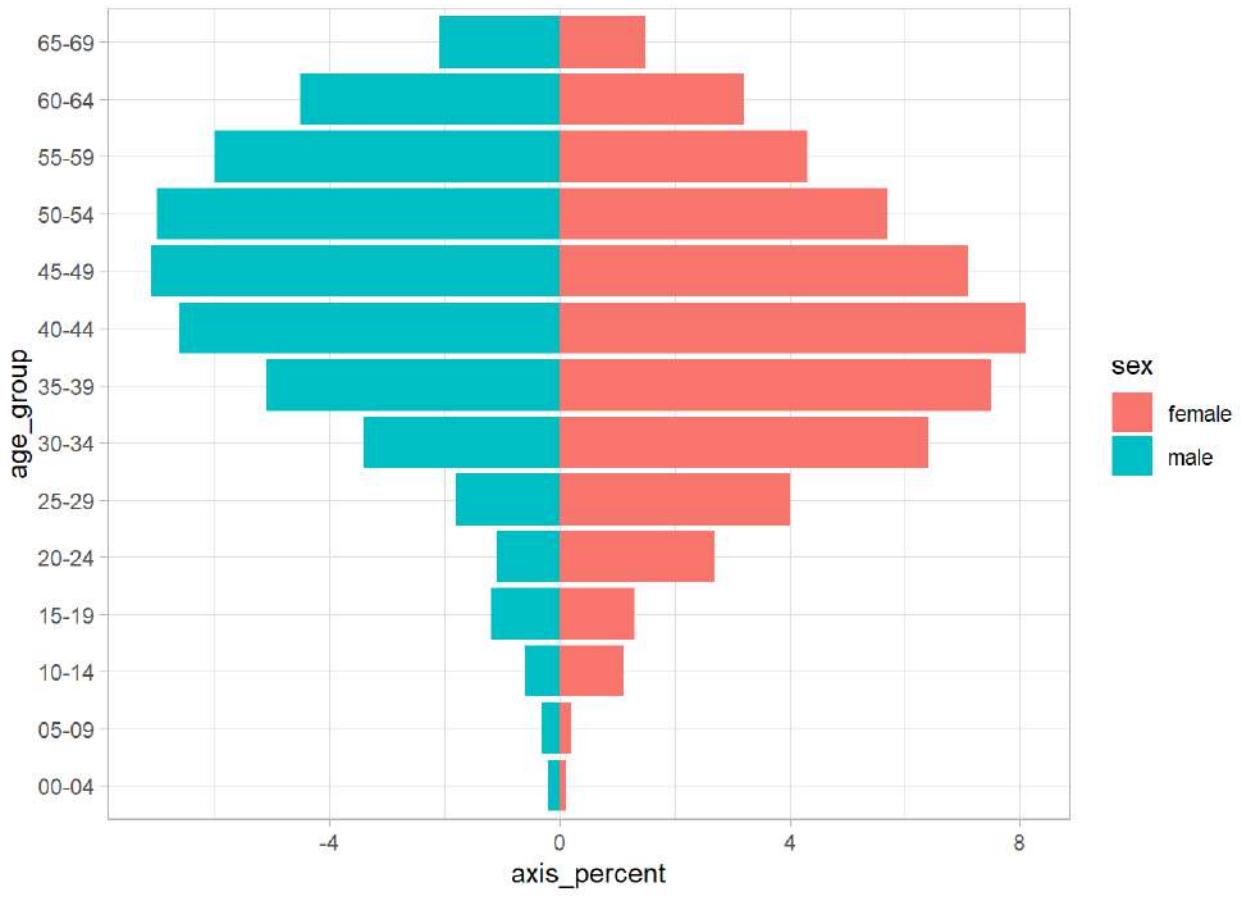
A demographic pyramid is a version of a stacked bar chart, with the bars' horizontal alignment adjusted along the x axis.

We can also create the same population pyramid using the percent totals on our y-axis.

	age_group	sex	total	axis_counts	axis_percent
	<chr>	<chr>	<int>	<int>	<dbl>
1	00-04	female	3	3	0.1
2	00-04	male	6	-6	-0.2
3	05-09	female	8	8	0.2
4	05-09	male	10	-10	-0.3
5	10-14	female	42	42	1.1
6	10-14	male	23	-23	-0.6
7	15-19	female	51	51	1.3
8	15-19	male	46	-46	-1.2
9	20-24	female	107	107	2.7
10	20-24	male	44	-44	-1.1

```
demo_pyramid_percent <-
  ggplot() +
  geom_col(data = pyramid_data,
            aes(x = age_group,
                 y = axis_percent, # use the pre-calculated percentages
                 fill = sex)) +
  coord_flip() +
  theme_light()

demo_pyramid_percent
```

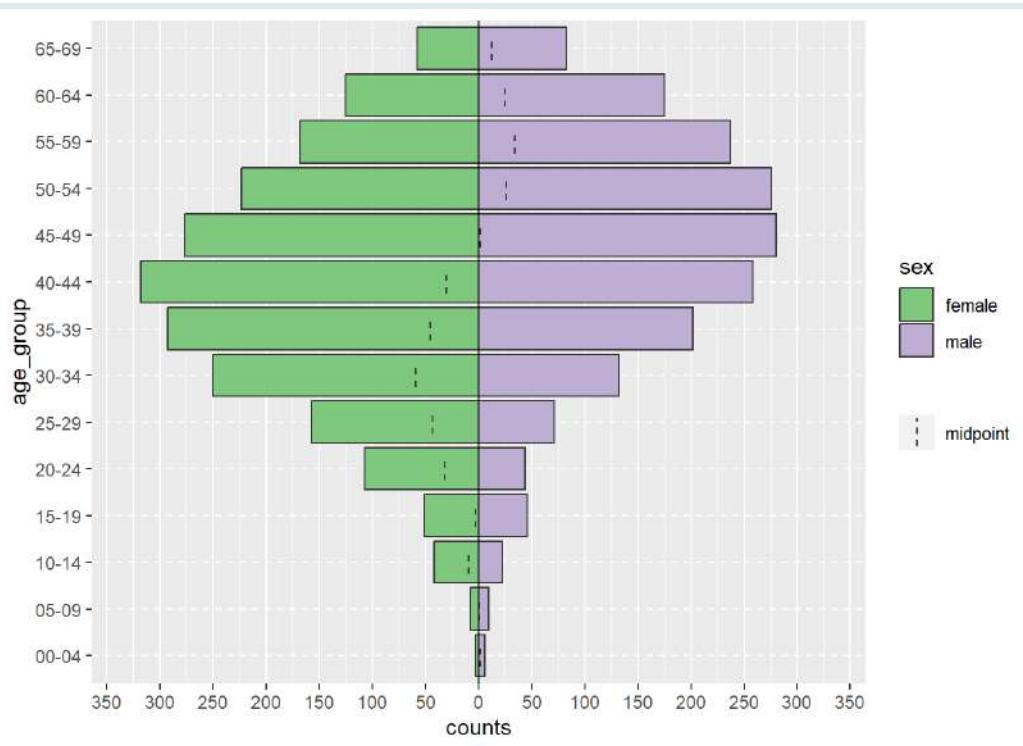


SIDE NOTE



Multiple packages are available to facilitate data analysis and data visualization. In the case of demographic pyramids, the `{apyramid}` package can be a useful tool. This package from the **R4Epis** project contains a function `age_pyramid()`, which allows for the rapid creation of population pyramids:

```
hiv_cases %>% # can use the original linelist
  # Grouping variable must be a factor
  mutate(age_group = factor(age_group)) %>%
  apyramid::age_pyramid(# 2 required arguments:
    age_group = "age_group",
    split_by = "sex")
```

SIDE NOTE

However, this package has limited functionality and few options for customization. `{ggplot2}` is a much more versatile approach, and uses syntax we are already familiar with.

Additional information about the function can be read [here](#) or by entering `?age_pyramid` in your R console.

Let's test your understanding with the following multiple-choice questions (Answer Key is located at the end):



2. When using `geom_col()`, what type of x variable should your dataset include?

- a. Continuous variables
- b. Categorical variables
- c. Binary variables
- d. Ordinal variables

3. Which `{ggplot2}` function can you use to flip the x and y axes?

- a. `coord_flip()`
- b. `x_y_flip()`
- c. `geom_flip()`
- d. Any of the above

Now let's test your understanding with the following coding practice question:

We will be using a cleaned and prepared dataset containing the total population of Zimbabwe in 2016 grouped by age group and sex.

Start by loading the prepared dataset as:

```
zw_2016 <- readRDS(here::here("data/population_zw_2016.rds"))

zw_2016
```



```
## # A tibble: 28 × 3
## # Groups:   age_group [14]
##       age_group sex    total_count
##   <fct>     <fct>      <int>
## 1 0-4        Female    1091129
## 2 0-4        Male     -1114324
## 3 10-14      Female    815362
## 4 10-14      Male     -803657
## 5 15-19      Female    803754
## 6 15-19      Male     -792474
## 7 20-24      Female    690940
## 8 20-24      Male     -632342
## 9 25-29      Female    638192
## 10 25-29     Male     -535444
## # i 18 more rows
```

Note that the male total count is already negated!

5. Create a demographic pyramid for the total population of Zimbabwe in 2016 using the `geom_col()` function from the `{ggplot2}` package. Make sure to add a white border around each bar!



```
Q4_pyramid_zw_2016 <-

# Begin ggplot
ggplot() +

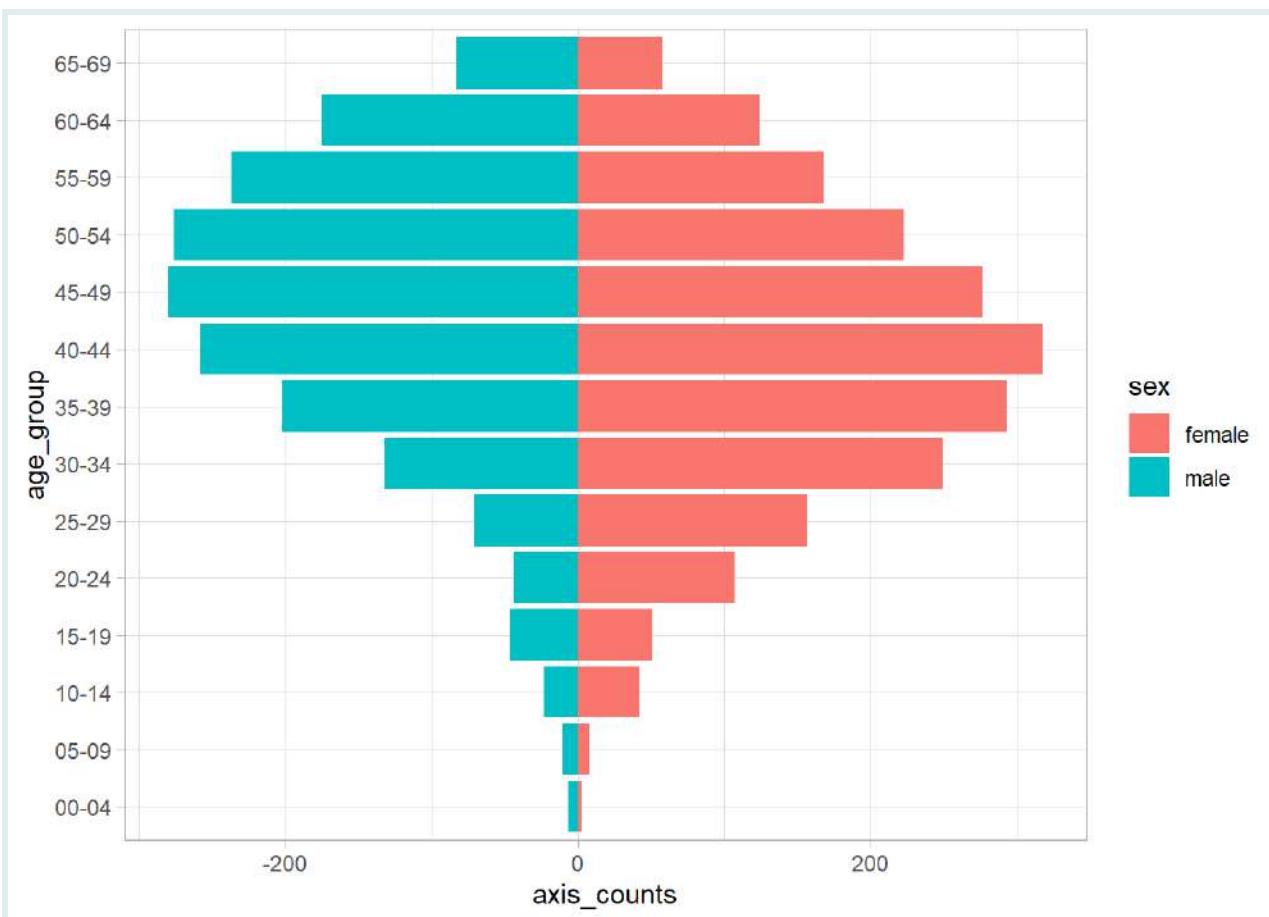
# Create bar graph using geom_bar
geom_col(data = ____,
          aes(x = ____,
              y = ____,
              fill = ____),
          color = ____) +

# Flip x and y axes
_____
```

Plot Customization

So far, you have learned how to create a demographic pyramid using `{ggplot2}` as shown below:

```
demo_pyramid
```



However, in order to create an informative graph, a certain level of plot customization is needed. For instance, it is important to include informative labels and to re-scale the x and y axis for better visualization.

Let's learn some useful `{ggplot2}` customization!

Axis Adjustments

The current axis settings result in an asymmetrical representation, with the right side (female) extending further than the left (male) due to a higher case count in the largest female age group. For an accurate comparison, it's essential that both sides mirror each other in range.

We will adjust the axis limits to equal positive and negative values, ensuring that the data is symmetrically visualized for a clear and balanced comparison.

We will start by re-scaling the *total count* axis, or in the case of our plot, the **y-axis**. For this, we will start by identifying the maximum and saving it as an object.

```
max_count <- max(pyramid_data$total)

max_count
```

```
## [1] 318
```

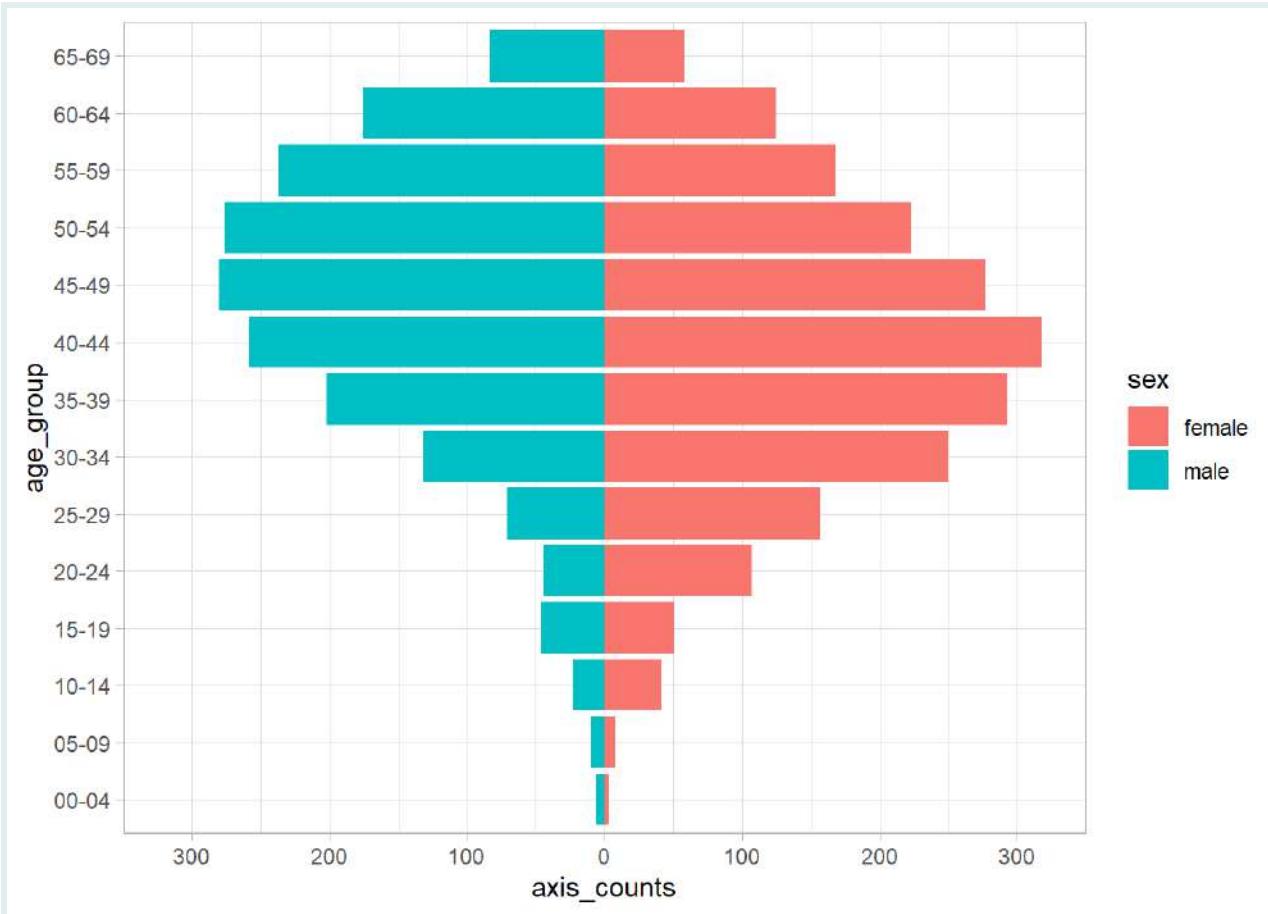
Now that we have identified that the maximum value for *total count* is **318**, we can use it to re-scale our y-axis accordingly.

In this particular case, we want to rescale our y axis to be symmetrical. Therefore we will take the biggest absolute value and use it as our limit for both the *positive* and *negative* sides.

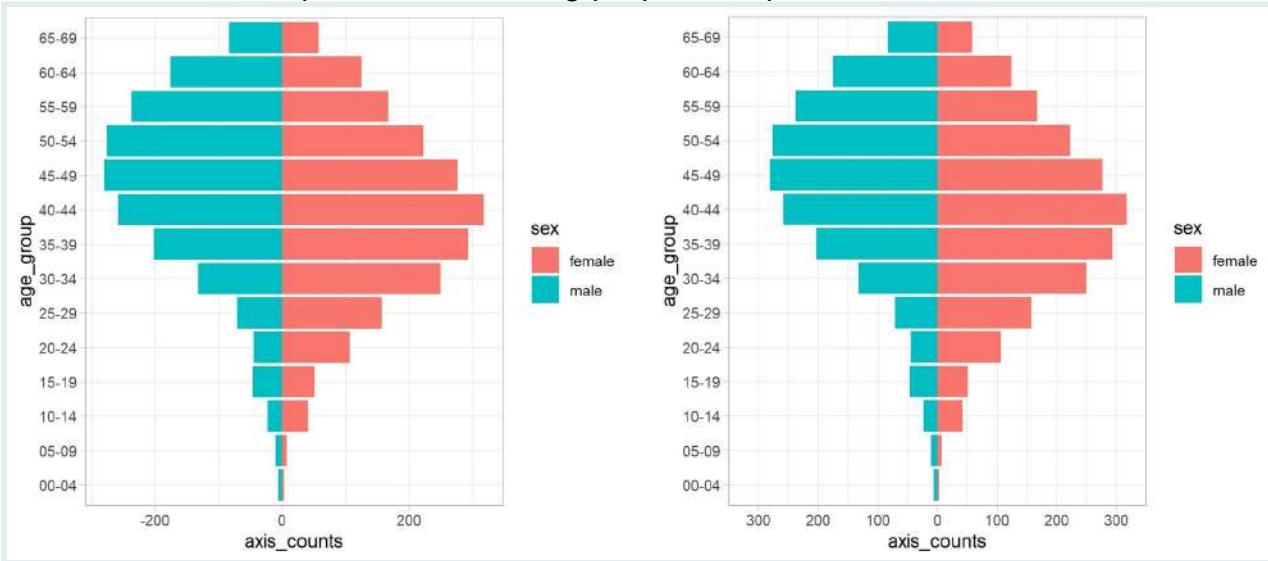
In this case, we will use our maximum value.

```
custom_axes <-
  # Use previous graph
  demo_pyramid +
    # Adjust y-axis (total count)
    scale_y_continuous(
      # Specify limit of y-axis using max value and making positive and negative
      limits = c(-max_count, max_count),
      # Specify the spacing between axis labels
      breaks = scales::breaks_width(100),
      # Make axis labels absolute so male labels appear positive
      labels = abs)

custom_axes
```



Adjusting axis limits to equal extents on both sides ensures a symmetrical and accurate visual comparison, facilitating proper interpretation of the data.



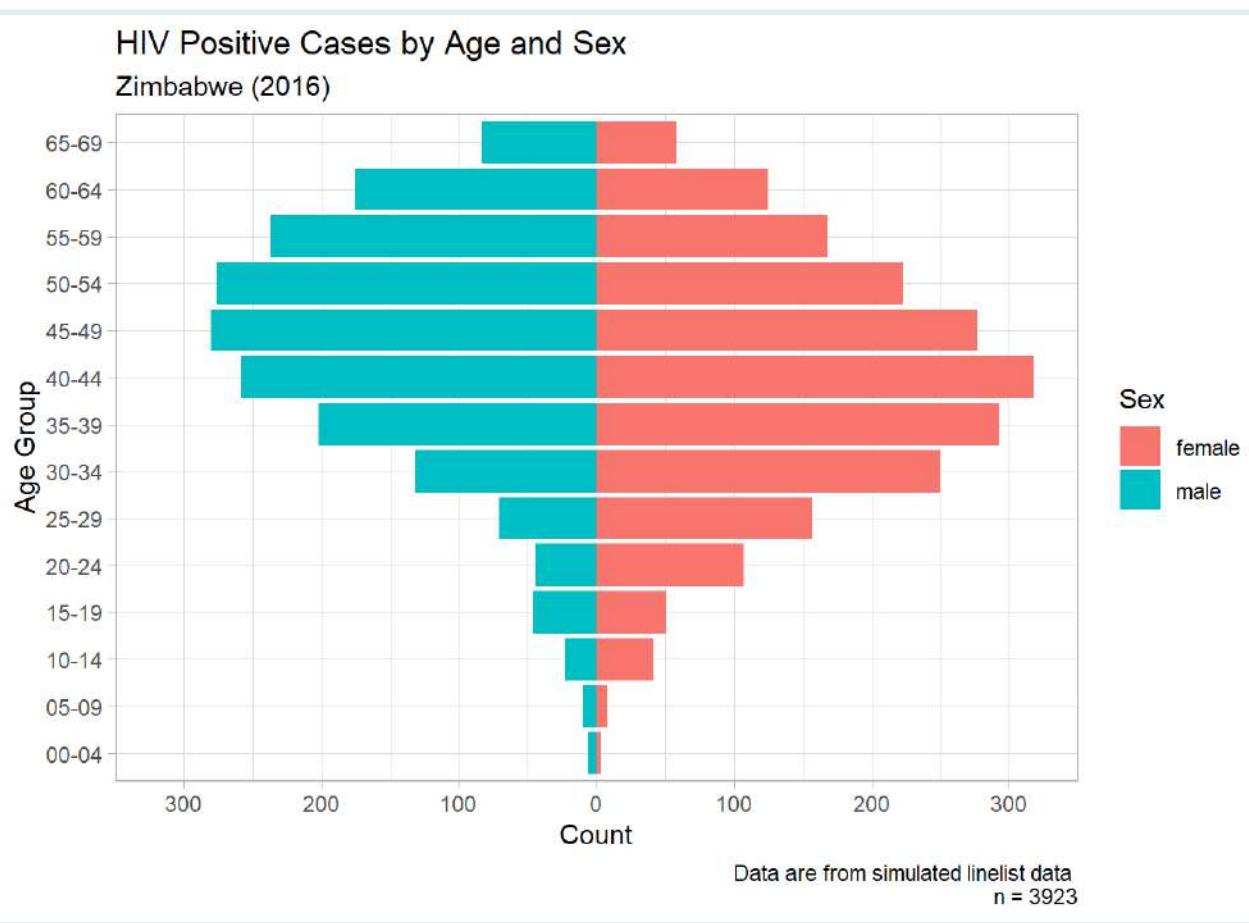
Note that we also changed the axis labels to their absolute value, so that the male case counts no longer appear as negative numbers.

Add custom labels

Let's use the population pyramid we previously created using the `geom_col()` function and build upon it.

We can start by adding an informative title, axes, and caption to our graph:

```
custom_labels <-
# Start with previous demographic pyramid
custom_axes +
# Adjust the labels
labs(
  title = "HIV Positive Cases by Age and Sex",
  subtitle = "Zimbabwe (2016)",
  x = "Age Group",
  y = "Count",
  fill = "Sex",
  caption = stringr::str_glue("Data are from simulated linelist data \nn =
  {nrow(hiv_cases)}"))
custom_labels
```

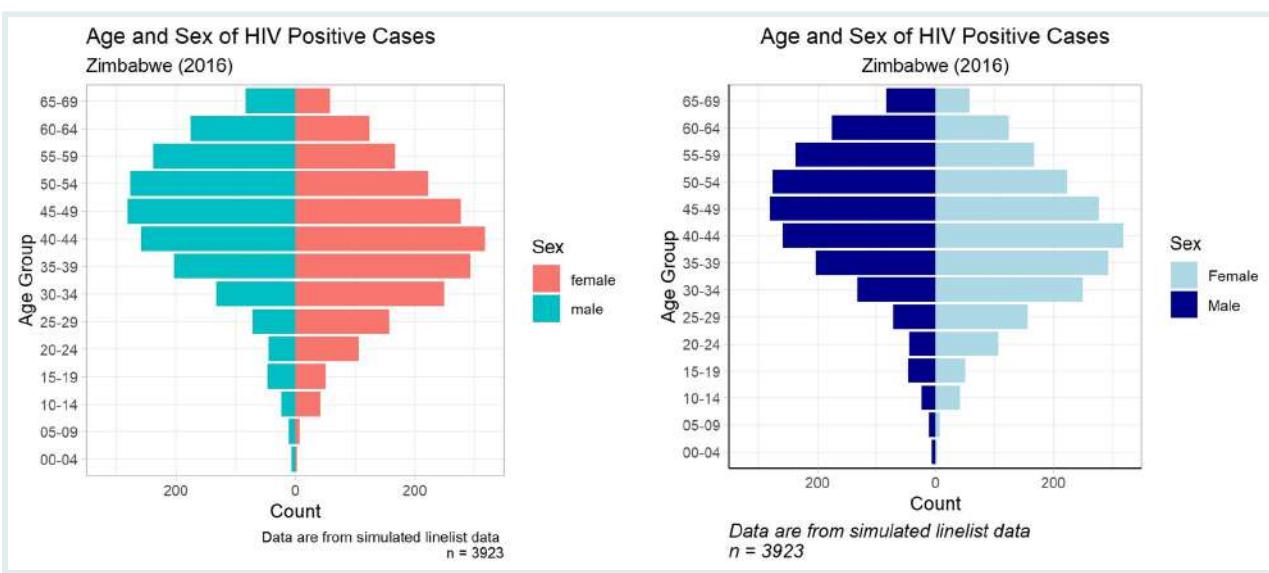
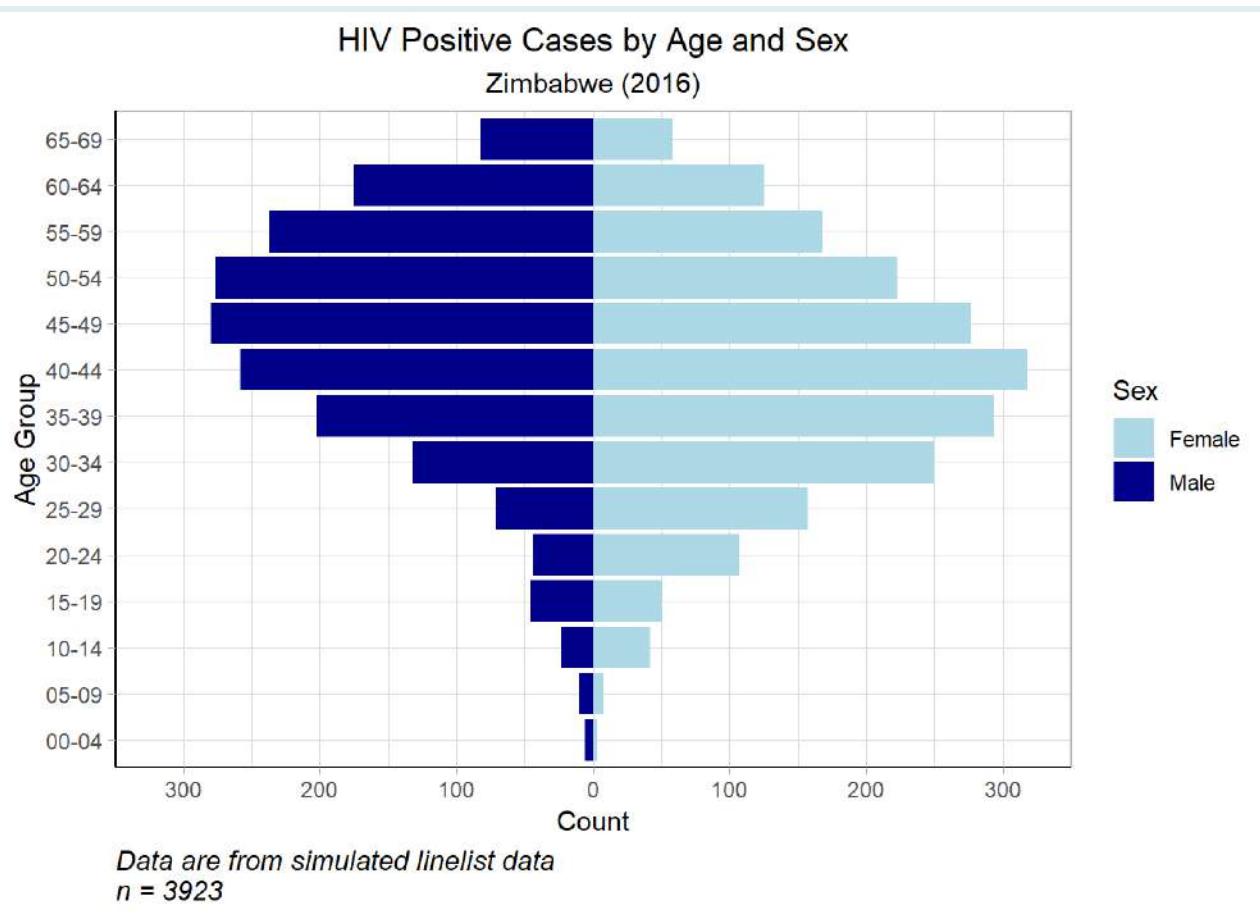


Enhance Color Scheme and Themes

We can also make necessary adjustments to the color scheme and theme of the graph.

Below is an example of some changes, we can perform:

```
custom_color_theme <-
  # Use previous graph
  custom_labels +
  # Designate colors and legend labels manually
  scale_fill_manual(
    # Select color of sex fill
    values = c("female" = "lightblue",
              "male" = "darkblue"),
    # Capitalize legend labels
    labels = c("Female", "Male")) +
  # Adjust theme settings
  theme(
    axis.line = element_line(colour = "black"), # make axis line black
    plot.title = element_text(hjust = 0.5), # center title
    plot.subtitle = element_text(hjust = 0.5), # center subtitle
    plot.caption = element_text(hjust = 0, # format caption text
                                size = 11,
                                face = "italic"))
custom_color_theme
```



- centered title and subtitle
- left align caption and increase font size
- capitalize legend text
- colors
- bold axis lines

WRAP UP!

As you can see, demographic pyramids are an essential visualization tool to understand the distribution of specific diseases across age groups and sex.

The concepts learned in this lesson can also be applied to create other types of graphs that require both negative and positive outputs such as percentage change in case notification rates and more.

Now that you have learned the concept behind the creation of demographic pyramids, the possibilities are endless! From plotting the **cases** per age group and sex over the **baseline/true** population to graphing the change (positive and negative) of interventions in a population, you should be able to apply these concepts to create informative epidemiological graphs.

Congratulations on finishing this lesson. We hope you can now apply the knowledge learned in today's lesson during the analysis and creation of epidemiological review reports.

Answer Key

1. d
2. b
3. a
4. c
- 5.

```
Q4_pyramid_zw_2016 <-
  ggplot() +
  geom_col(data = zw_2016,
            aes(x = age_group,
                 y = total_count,
                 fill = sex),
            color = "white") +
  coord_flip()
```

Contributors

The following team members contributed to this lesson:



SABINA RODRIGUEZ VELÁSQUEZ

Project Manager and Scientific Collaborator, The GRAPH Network
Infectiously enthusiastic about microbes and Global Health



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

1. Adjusted lesson content from: Batra, Neale, et al. The Epidemiologist R Handbook. 2021. <https://doi.org/10.5281/zenodo.4752646>
2. Adjusted lesson content from: WHO. Understanding and Using Tuberculosis Data. 2014. https://apps.who.int/iris/bitstream/handle/10665/129942/9789241548786_eng.pdf
3. Referenced package from: <https://r4epis.netlify.app/>

Visualizing Comparisons and Compositions

Introduction
Learning objectives
Load packages
Data: TB treatment outcomes in Benin
Visualizing comparisons
Bar charts
Stacked bar charts
Grouped bar charts
Adding error bars
Area charts
Visualizing comparisons with normalized bar charts, pie charts, and donut charts
Percent-stacked bar chart
Circular plots: Pie and Donut charts
Wrap Up!

Introduction

Welcome to our tutorial on visualizing comparisons and compositions using `{ggplot2}` in R. Analyzing and comparing categories, as well as visually representing the composition of elements, are fundamental aspects of data visualization in R.

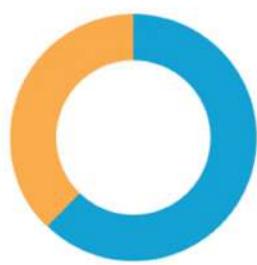
In this lesson, we'll explore a variety of visualization techniques for these two applications!

We will start our journey by examining bar charts, a foundational element in data visualization. Bar charts offer exceptional versatility, allowing straightforward comparisons across diverse categories. However, when our focus shifts to understanding proportions or compositions, pie charts and donut charts come into play. These charts prove invaluable in visually representing data where the cumulative sum of all parts constitutes a whole.

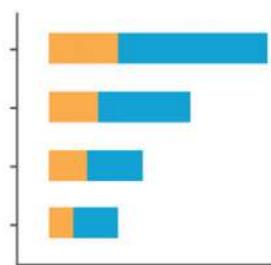
This tutorial aims to equip you with the skills to effectively visualize aggregated data and make comparisons. By the end, you'll have a solid understanding of how to use `{ggplot2}` functions to visualize and compare groups in your data, enhancing your ability to draw meaningful insights from your datasets. Let's get started!



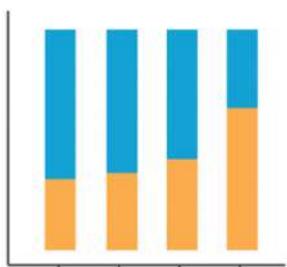
Pie chart



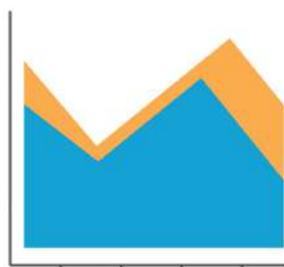
Donut chart



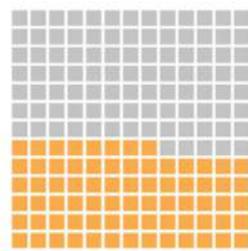
Stacked bar chart



100% stacked
bar chart



Area chart

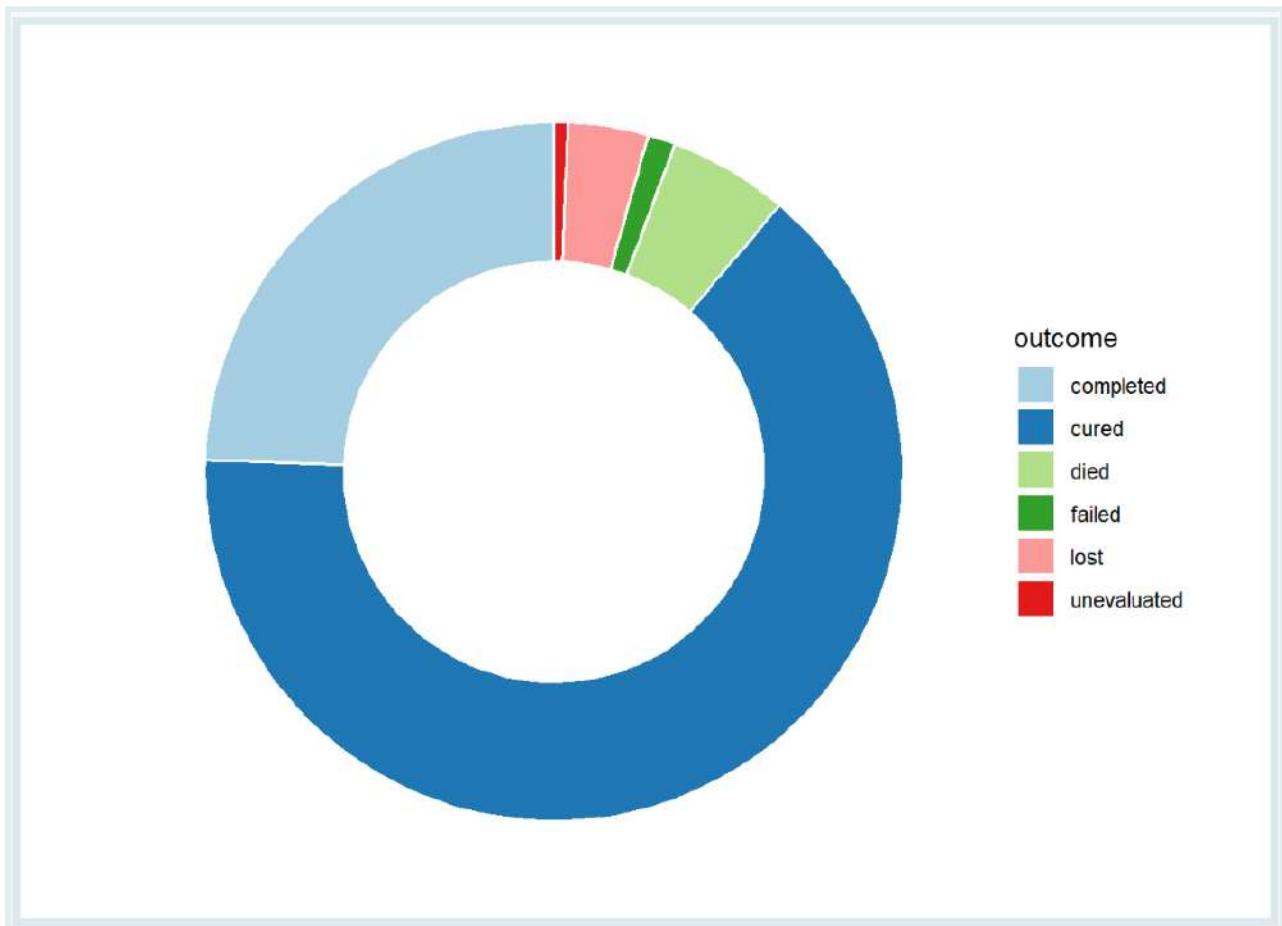


Waffle chart

Chart types covered in this lesson.

Learning objectives

1. Understand the difference between visualizing comparisons and visualizing compositions, and recall the appropriate chart types for these two types of analysis.
2. Create and customize bar charts using `{ggplot2}` for comparing categorical data, with `geom_col()`, `geom_errorbar()`, and `position` adjustments.
3. Create area plots with `geom_area()` for visualizing comparisons.
4. Understand the principles of creating effective pie and donut charts for visualizing compositions.
5. Create and customize pie and donut charts using `coord_polar()` with `geom_col()`.



Load packages

In this lesson we will use the following packages:

- `{tidyverse}` for data wrangling and data visualization
- `{here}` for project-relative file paths

```
pacman::p_load("tidyverse", "here")
```

Data: TB treatment outcomes in Benin

In the realm of public health, data often comes in forms that require us to compare metrics between subgroups, or understand the relative contributions to a total count.

Today we'll be looking at sub-national tuberculosis (TB) data from Benin. The data was provided by WHO and hosted on this [DHIS2 dashboard](#). We will be looking at a subset of this data, with records of treatment outcomes for TB patients in hospitals, from 2015 to 2017.

Let's import the `tb_outcomes` data subset.

```
# Import data from CSV
tb_outcomes <- read_csv(here::here('data/benin_tb.csv'))

# Print data frame
tb_outcomes
```

The data contains records of the number of new and relapse cases started on treatment (`cases`). The case counts are disaggregated by a number of groupings: time period, health facility, treatment outcome, and diagnosis type.

Here are the detailed variable definitions for each column:

1. `period` and `period_date`: records the time frame for each entry in the dataset. The periods are marked quarterly, starting from the first quarter of 2015 (represented as `2015Q1`) up to the last quarter of 2017 (`2017Q4`). This allows us to track the progression and changes in TB cases over time.
2. `hospital`: indicates the specific health facility where the TB cases were recorded. These facilities represent different geographical and administrative areas, each with unique characteristics and capabilities. This subset of the data contains treatment outcome records from five health facilities: St Jean De Dieu, CHPP Akron, CS Abomey-Calavi, Hopital Bethesda, Hopital Savalou, and Hopital St Luc. This information can be used to analyze and compare the prevalence and treatment outcomes of TB across different facilities.
3. `outcome`: This column categorizes the TB cases based on the diagnosis type and the stage of their treatment journey. Each variable corresponds to a different aspect of the patient's diagnosis and treatment progress:
 - `completed`: started on treatment and the respective outcome is indicated as completed.

- **cured**: Started on treatment and for whom the respective outcome is indicated as cured (and backed by at least two clear sputum smear results)
- **died**: Represents the TB cases that resulted in the death of the patient during treatment, including both bacteriologically confirmed and clinically diagnosed cases. died while under treatment
- **failed**: These are the cases where treatment failed, which is confirmed for bacteriologically tested patients and observed in clinically diagnosed cases.
- **unevaluated**: These are the TB cases that started treatment but do not have an evaluated treatment outcome available. This applies to both bacteriologically confirmed and clinically diagnosed cases.

4. **diagnosis_type**: This column categorizes the TB cases based on the method of diagnosis. There are two types of diagnosis included in this dataset:

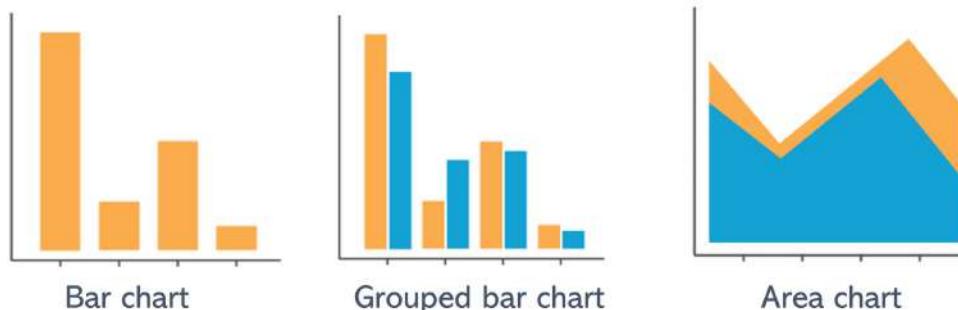
- **bacteriological**: These are the cases where the presence of TB bacteria is confirmed through bacteriological examination methods, such as sputum smear microscopy or culture methods.
- **clinical**: These are the cases where TB diagnosis is made based on clinical signs and symptoms, without bacteriological confirmation. This usually happens when bacteriological tests are either unavailable or inconclusive, and the patient presents TB symptoms.

5. **cases**: the number of new or relapse cases started on treatment. This allows for quantitative analysis of the TB cases, such as the total number of new cases over a specific period or the number of cases that completed treatment in a particular health facility.

Visualizing comparisons

Visualizations allow you to ask questions of your data, and the type of question you want to ask will determine the chart type. A ton of questions you might ask about your data will hinge on a comparison between values. Which region was responsible for the most cases or deaths? What year had the best treatment outcomes? These questions are answered quickly by gathering and comparing different values in your data.

Charts for visualizing comparisons



A selection of chart types for making comparisons between groups of data.

Bar charts

One of the most common chart types are bar charts, and for good reason! They are often the most efficient and effective way of conveying counts per group or category so that comparisons can be made between the bars.

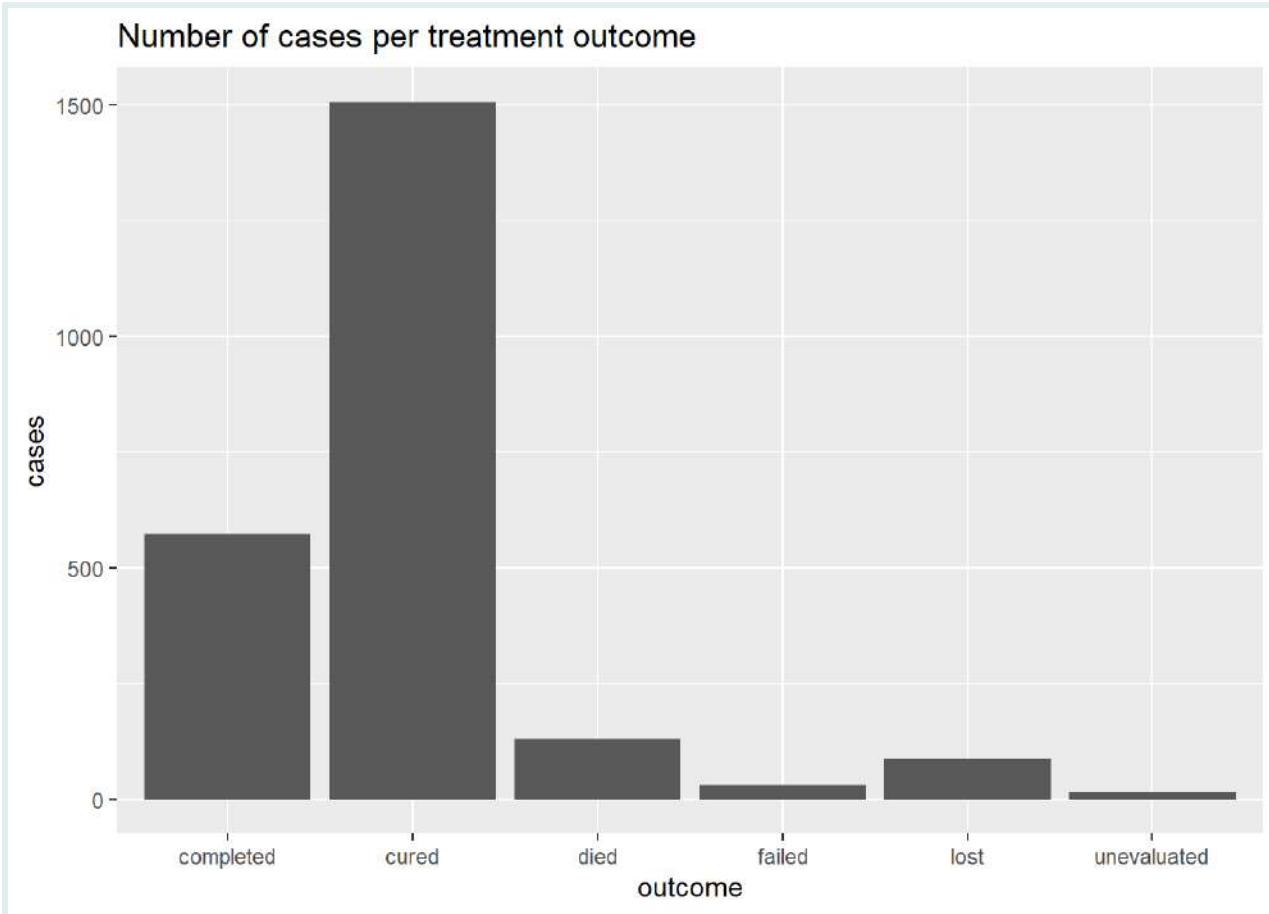
They are especially effective when the categories are ordinal (i.e. contain an inherent order; e.g., age groups), or time based (e.g., months of the year).

But when should we consider choosing a bar/column chart? Bar charts are ideally used when dealing with data that can be **grouped into categories** and when wanting to make **comparisons** between these categories.

When using `{ggplot2}`, we can use the `geom_col()` function to create a bar plot of a categorical variable against a numerical variable.

Let's exemplify this by visualizing the *Number of cases per treatment outcomes* in the `tb_outcomes` dataset:

```
# Basic bar plot example 1: Frequency of treatment outcomes
tb_outcomes %>%
  # Pass the data to ggplot as a basis for creating the visualization
  ggplot(
    # Specify that the x and y axis variables
    aes(x = outcome, y = cases)) +
  # geom_col() creates a bar plot
  geom_col() +
  labs(title = "Number of cases per treatment outcome")
```

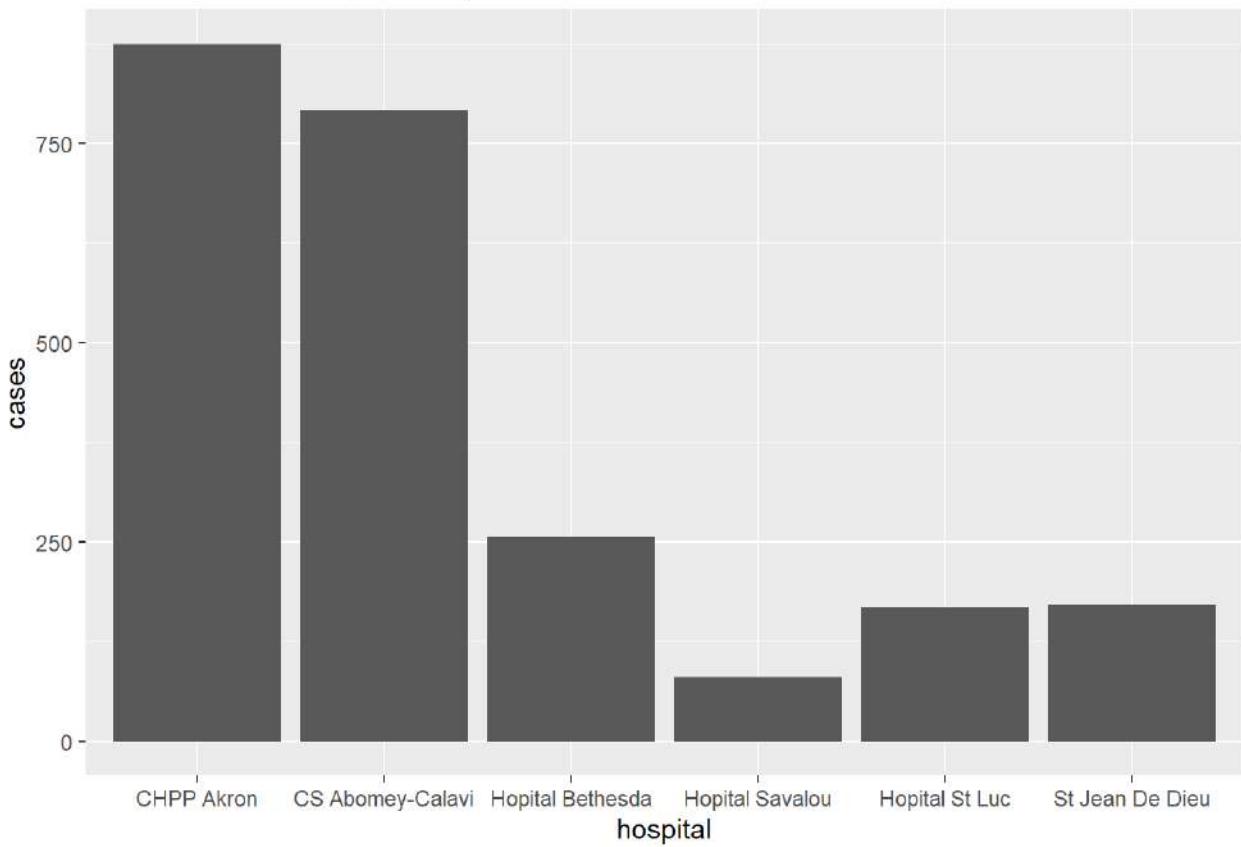


As we can see from the graph above, `geom_col()` has automatically tallied the total number of cases per outcomes, across all periods, hospitals, and diagnosis types.

We can also change the x-axis variable to any other categorical variable in the data as below:

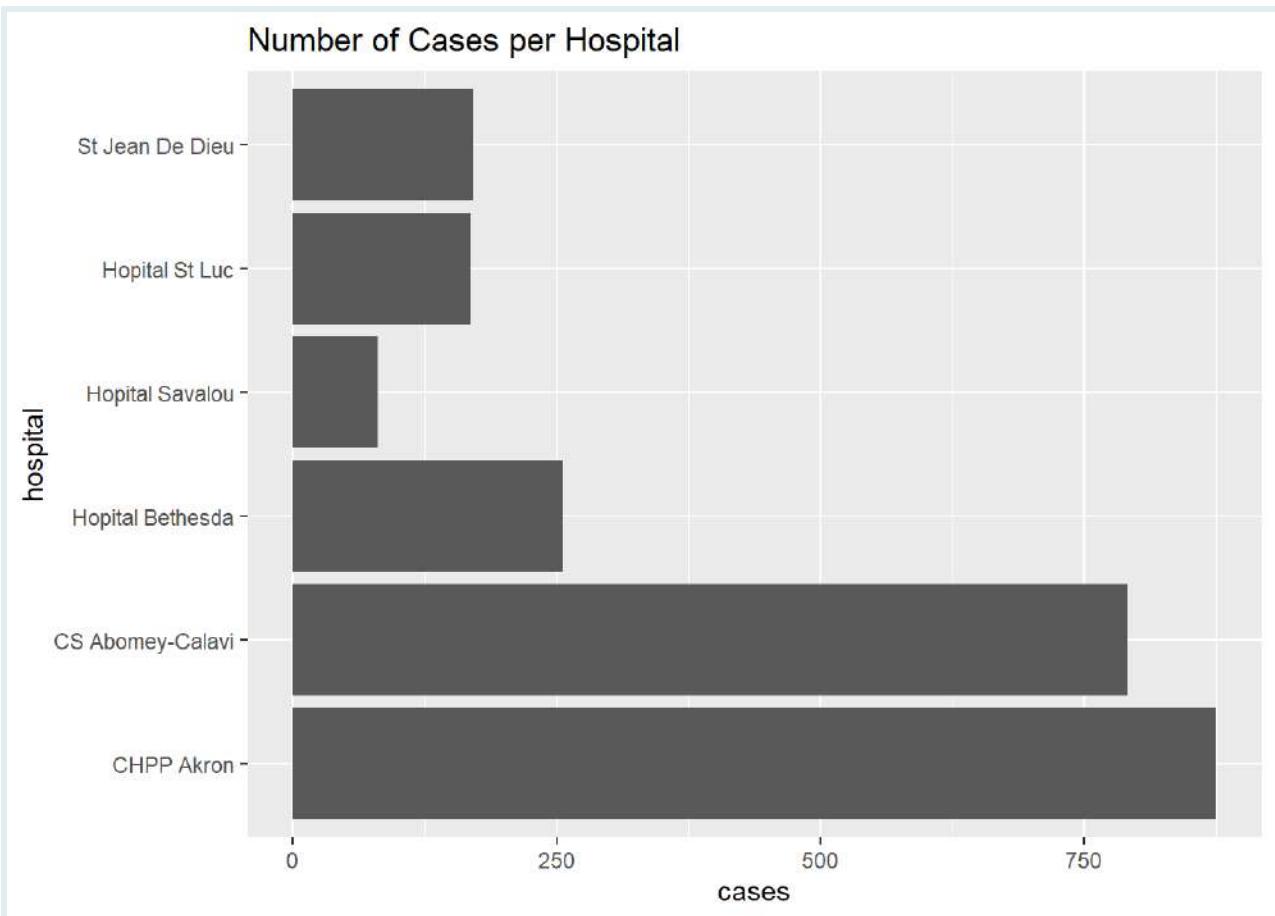
```
# Basic bar plot example 2: Case counts per hospital
tb_outcomes %>%
  ggplot(aes(x = hospital, y = cases)) +
  geom_col() +
  labs(title = "Number of Cases per Hospital")
```

Number of Cases per Hospital



In order to properly visualize the various categories, we can generate a horizontal bar plot, by integrating the `coord_flip()` function into our previous code.

```
# Basic bar plot example 3: Horizontal bars
tb_outcomes %>%
  ggplot(aes(x = hospital, y = cases)) +
  geom_col() +
  labs(title = "Number of Cases per Hospital") +
  coord_flip()
```



SIDE NOTE



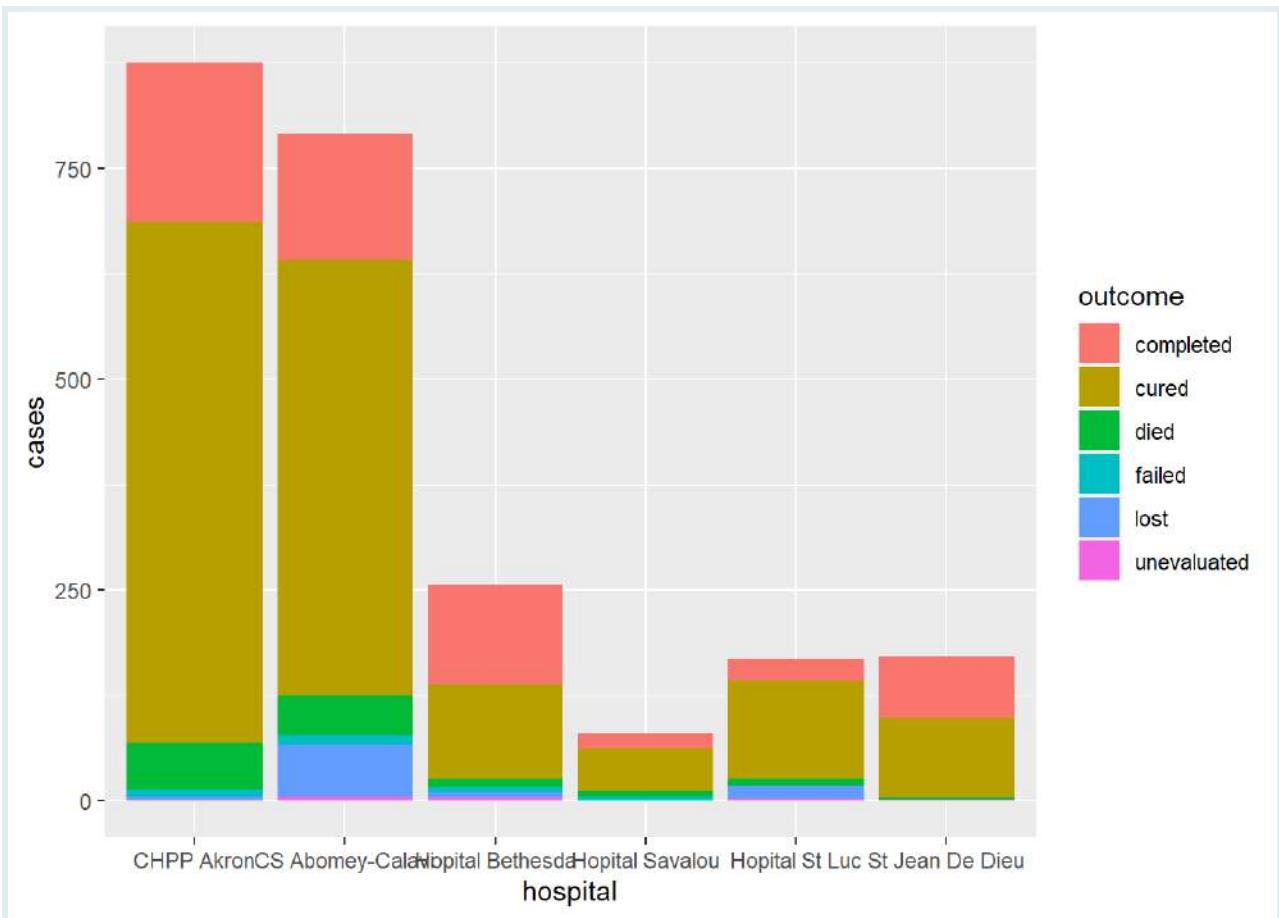
We will use `coord_*` functions later in the lesson to create circular plots.

Stacked bar charts

While the previous bar charts depicted case distribution across a *single categorical variable*, we can elevate our understanding by introducing a *second categorical variable* and creating **stacked bar charts**.

This can be done in `ggplot()` by setting fill color to a categorical variable:

```
# Stacked bar plot:
tb_outcomes %>%
  ggplot(
    # Fill color of bars by the 'outcome' variable
    aes(x = hospital,
        y = cases,
        fill = outcome)) +
  geom_col()
```

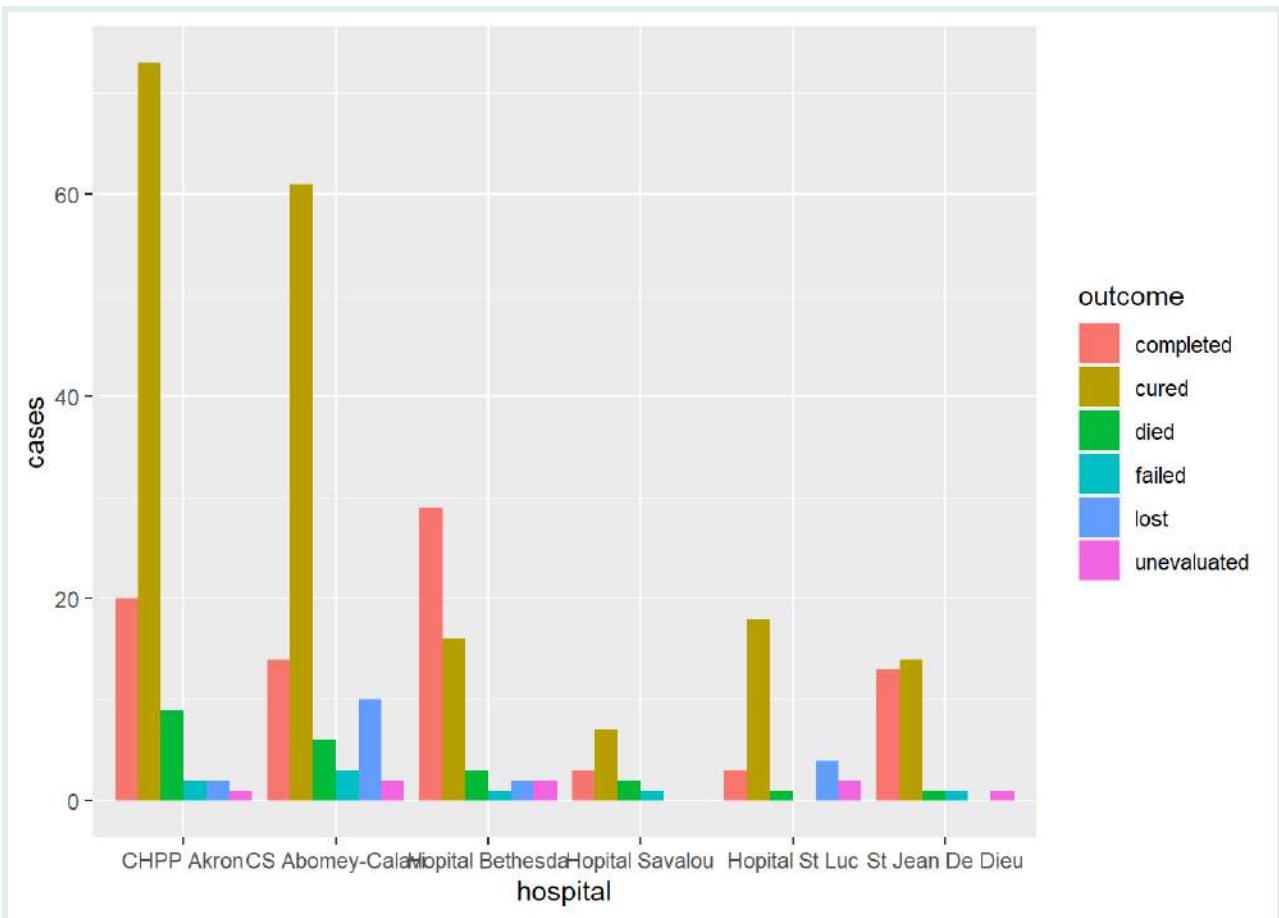


As seen above, stacked plots maintain our *primary categories* on the axis while visually **segregating contributions** from different subgroups by splitting the bars into smaller segments.

Grouped bar charts

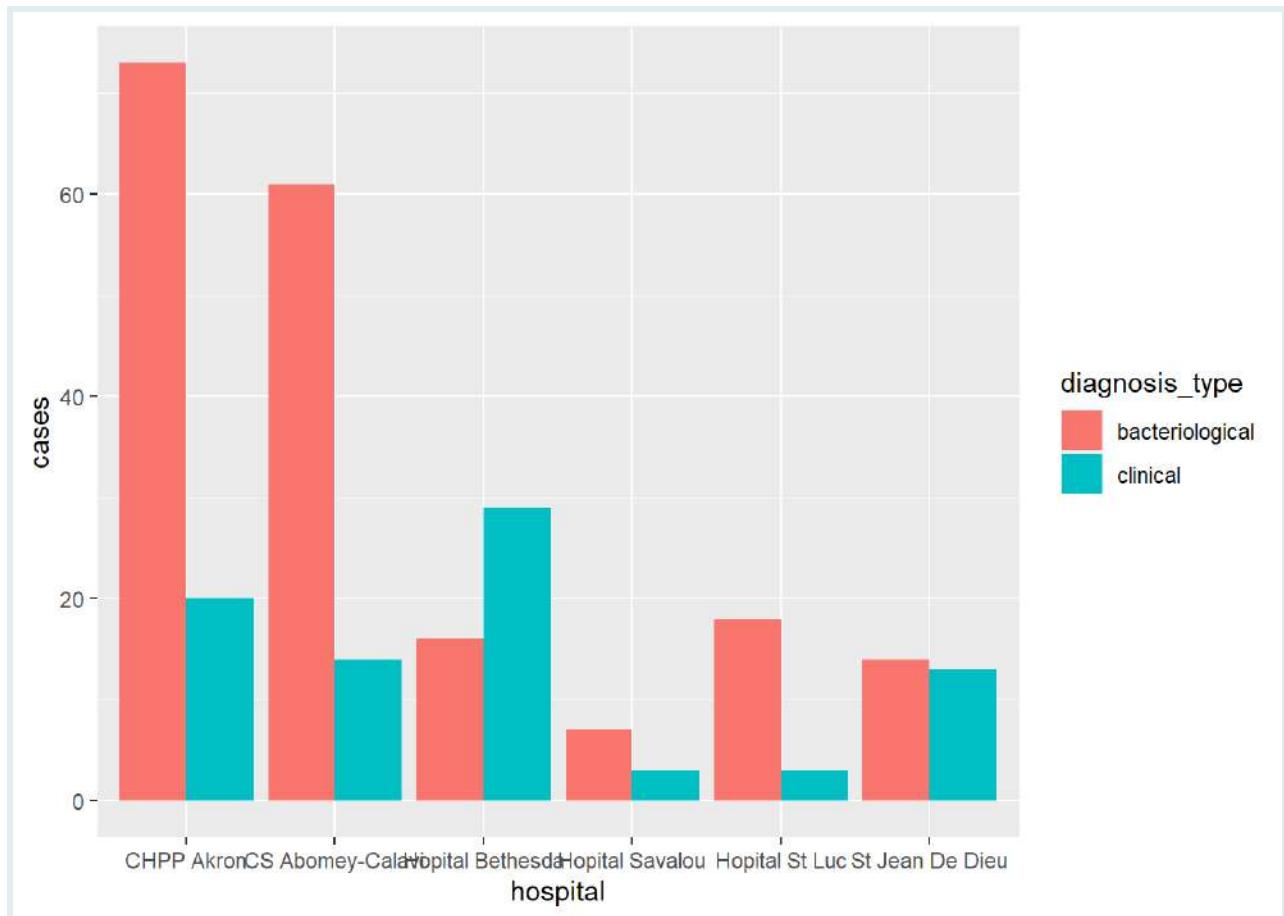
- Grouped bar plots provide a side-by-side representation of subgroups within each main category.
- We can set the `position` argument to "dodge" in `geom_col()` to display bars side by side:

```
# Grouped bar plot:
tb_outcomes %>%
  ggplot(
    aes(x = hospital,
        y = cases,
        fill = outcome)) +
  # Add position argument for side-by-side bars
  geom_col(position = "dodge")
```



- Grouped bar charts are not ideal when there are too many groups.
- We can try this again but with a different grouping variable that has fewer categories:

```
# Grouped bar plot: split into 2 bars
tb_outcomes %>%
  ggplot(
    # Fill color of bars by the 'diagnosis_type'
    aes(x = hospital,
        y = cases,
        fill = diagnosis_type)) +
  geom_col(position = "dodge")
```



Question 1: Basic bar plot

Write the adequate code that generates a basic bar chart of the number of `cases` per quarter with `period_date` on the x axis



```
# PQ1 answer:
tb_outcomes %>%
  ggplot(aes(-----, -----)) +
  geom_col()

## Error: <text>:3:20: unexpected ','
## 2: tb_outcomes %>%
## 3:   ggplot(aes(-----,
##
```

Question 2: Stacked bar plot

Create a stacked bar chart to display treatment outcomes over different time periods



```
# PQ2 answer:  
tb_outcomes %>%  
  ggplot(  
    aes(-----, -----, -----)) +  
    geom_col()  
  
## Error: <text>:4:14: unexpected ',',  
## 3:   ggplot(  
## 4:     aes(-----,  
##           ^
```

Adding error bars

Visualizing data with error bars allows for a clearer understanding of the variability or uncertainty inherent in the dataset. Error bars can indicate the reliability of a mean score or an individual data point, providing context to the plotted values.

To implement error bars in {ggplot2}, we use the `geom_errorbar()` function. This requires a value for the range of your error, typically defined by the standard deviation, standard error, or confidence intervals.

Here's an example of how to add error bars to our grouped `geom_col()` plot above.

First, let's create the necessary summary data since we need to have some kind of error measurement. In our case we will compute the standard deviation:

```
hosp_dx_error <- tb_outcomes %>%  
  group_by(period_date, diagnosis_type) %>%  
  summarise(  
    total_cases = sum(cases, na.rm = T),  
    error = sd(cases, na.rm = T))  
  
hosp_dx_error
```

```
## # A tibble: 24 × 4  
## # Groups:   period_date [12]  
##       period_date diagnosis_type  total_cases error  
##       <date>        <chr>            <dbl> <dbl>  
## 1 2015-01-01  bacteriological      143 11.9  
## 2 2015-01-01  clinical              47  4.40  
## 3 2015-04-01  bacteriological      163 13.0  
## 4 2015-04-01  clinical              35  3.84  
## 5 2015-07-01  bacteriological      146 11.2  
## 6 2015-07-01  clinical              34  3.33
```

```

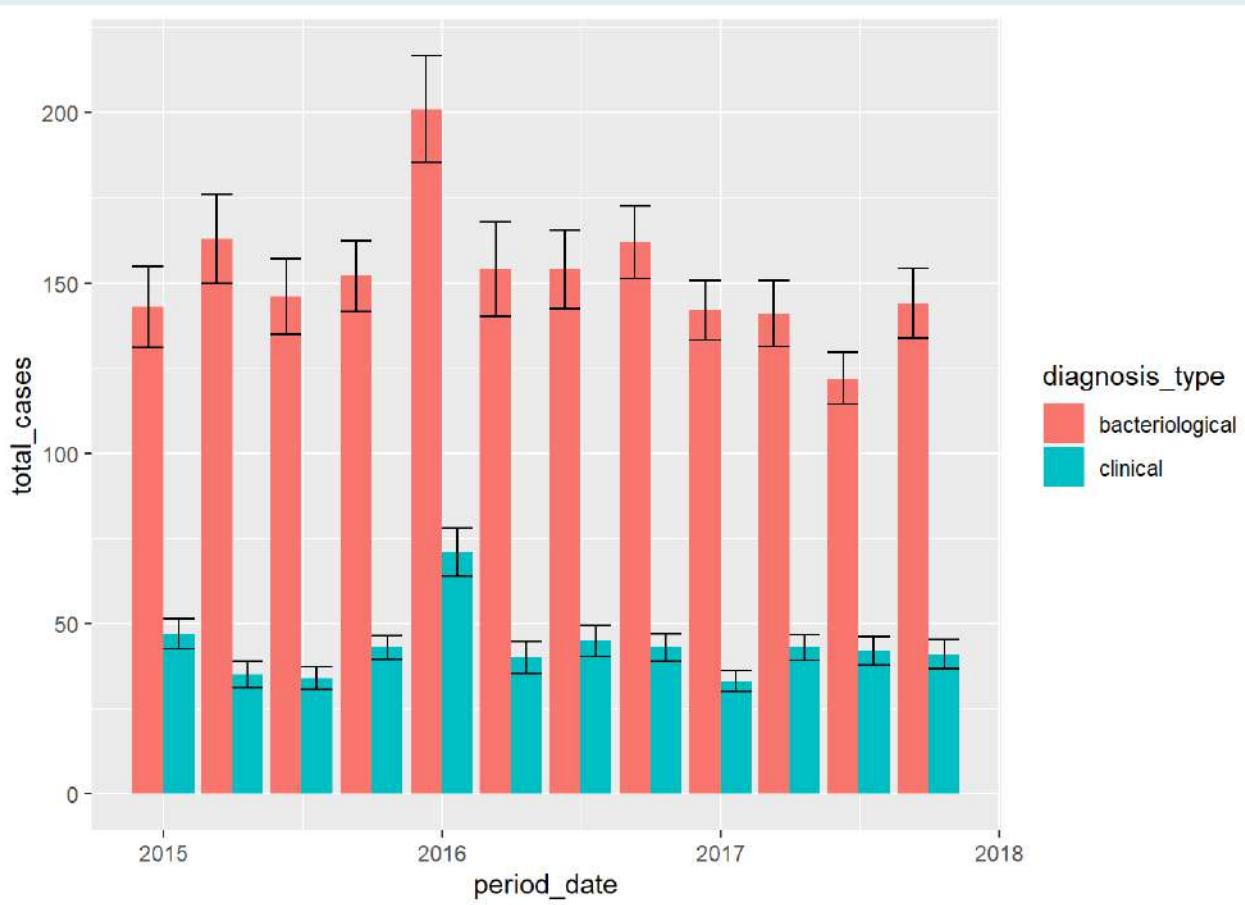
## 7 2015-10-01 bacteriological      152 10.4
## 8 2015-10-01 clinical            43  3.55
## 9 2016-01-01 bacteriological    201 15.7
## 10 2016-01-01 clinical          71  7.01
## # i 14 more rows

```

```

# Recreate grouped bar chart and add error bars
hosp_dx_error %>%
  ggplot(
    aes(x = period_date,
        y = total_cases,
        fill = diagnosis_type)) +
  geom_col(position = "dodge") + # Dodge the bars
  # geom_errorbar() adds error bars
  geom_errorbar(
    # Specify upper and lower limits of the error bars
    aes(ymin = total_cases - error, ymax = total_cases + error),
    position = "dodge" # Dodge the error bars to align them with side-by-side
    bars
  )

```

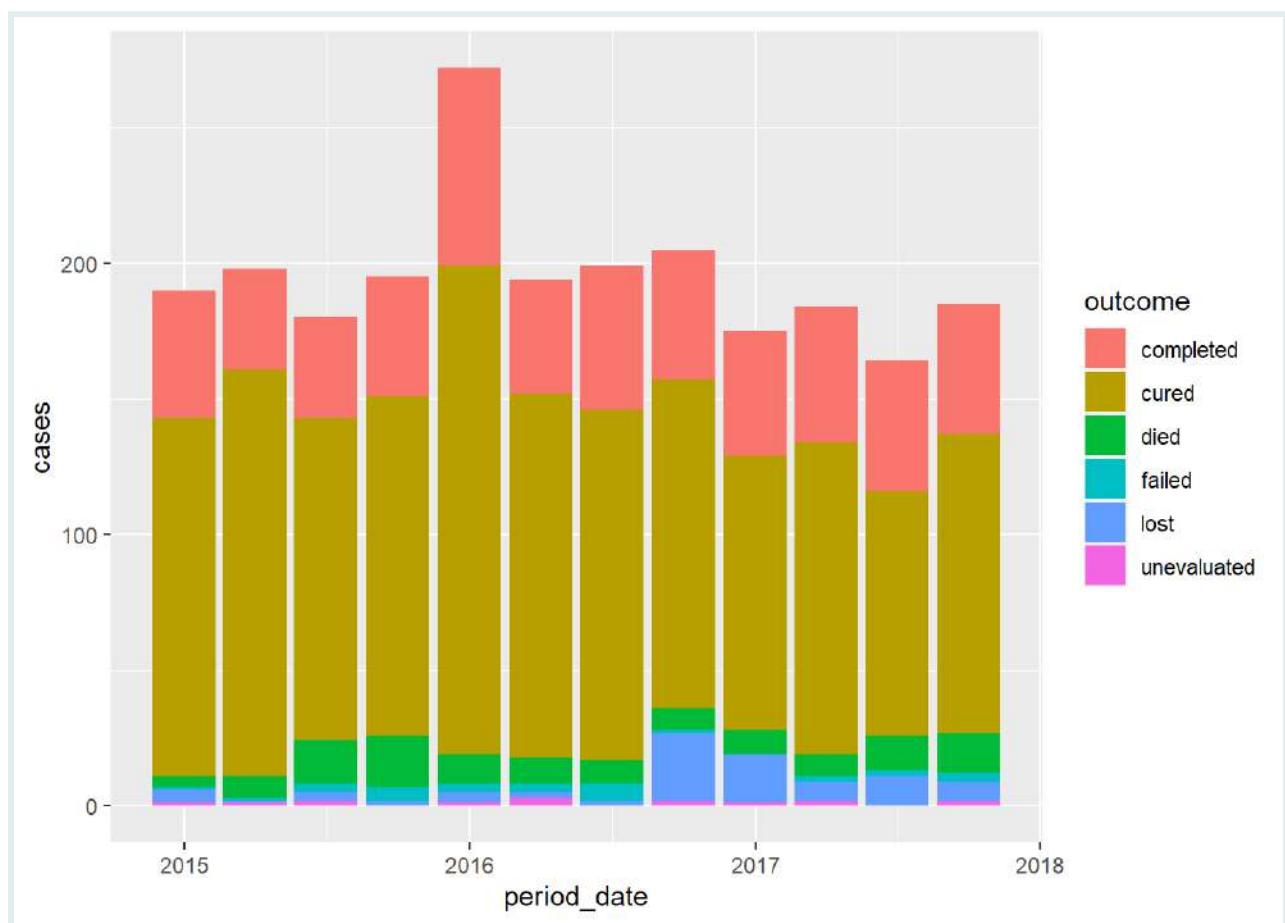


Area charts

In this section, we'll be exploring area plots with `geom_area()`. These plots are especially advantageous for visualizing a range of data values or the progression of data across time, making them ideal for use cases like tracking the number of new treatment cases per quarter.

Let's start by visualizing this distribution with a stacked bar chart:

```
tb_outcomes %>%
  ggplot(
    aes(x = period_date,
        y = cases,
        fill = outcome)) +
  geom_col()
```



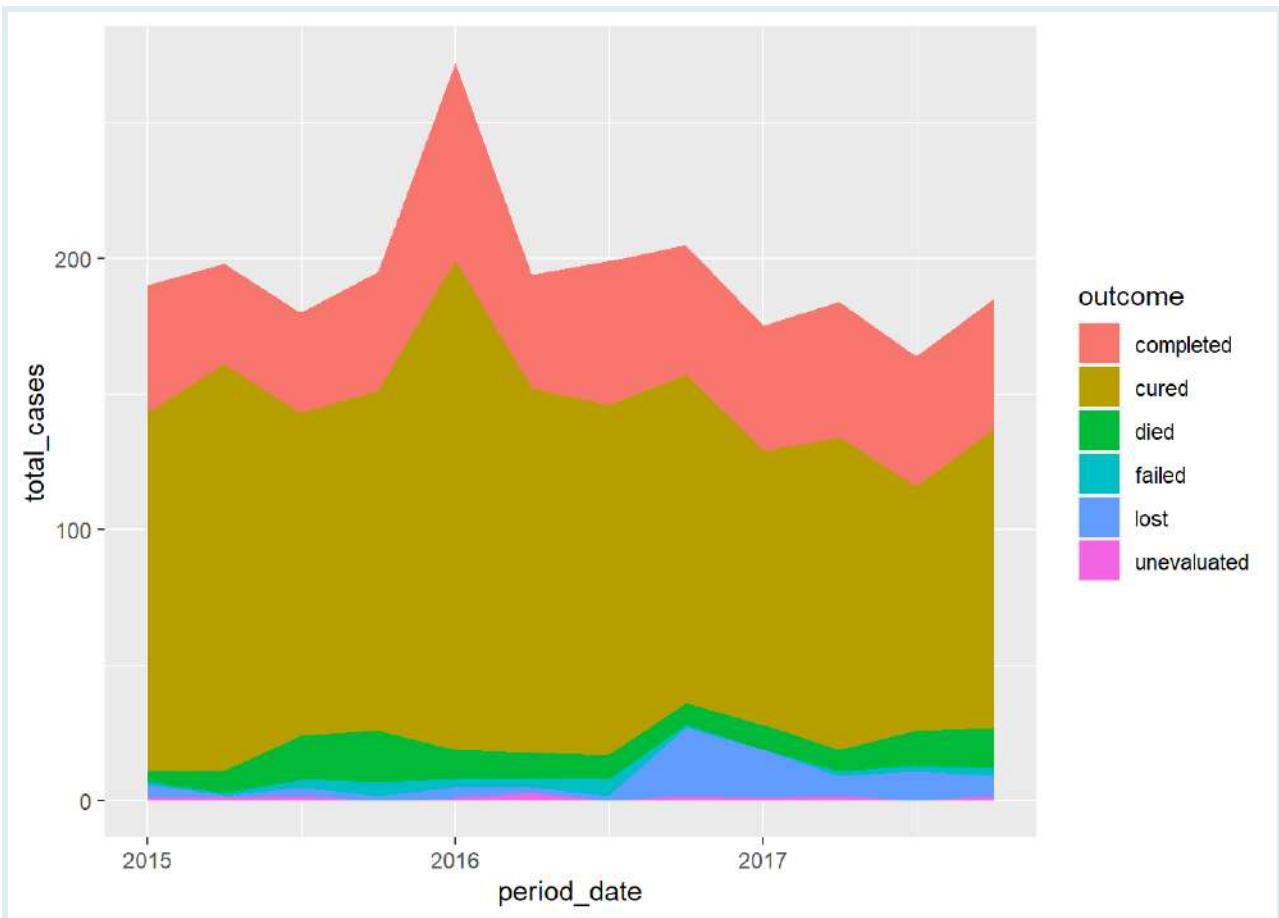
- Unlike `geom_col()`, `geom_area()` will not automatically compute the sums for each category.
- We need to group summarize our data so we have a data frame with the sum of cases for every treatment outcome per quarter:

```
# Create summary data frame
outcome_by_period <- tb_outcomes %>%
  group_by(period_date, outcome) %>%
  summarise(
    total_cases = sum(cases, na.rm = T))

outcome_by_period
```

```
## # A tibble: 72 × 3
## # Groups:   period_date [12]
##       period_date outcome     total_cases
##       <date>      <chr>        <dbl>
## 1 2015-01-01 completed     47
## 2 2015-01-01 cured       132
## 3 2015-01-01 died          4
## 4 2015-01-01 failed        1
## 5 2015-01-01 lost          5
## 6 2015-01-01 unevaluated   1
## 7 2015-04-01 completed     37
## 8 2015-04-01 cured       150
## 9 2015-04-01 died          8
## 10 2015-04-01 failed        1
## # i 62 more rows
```

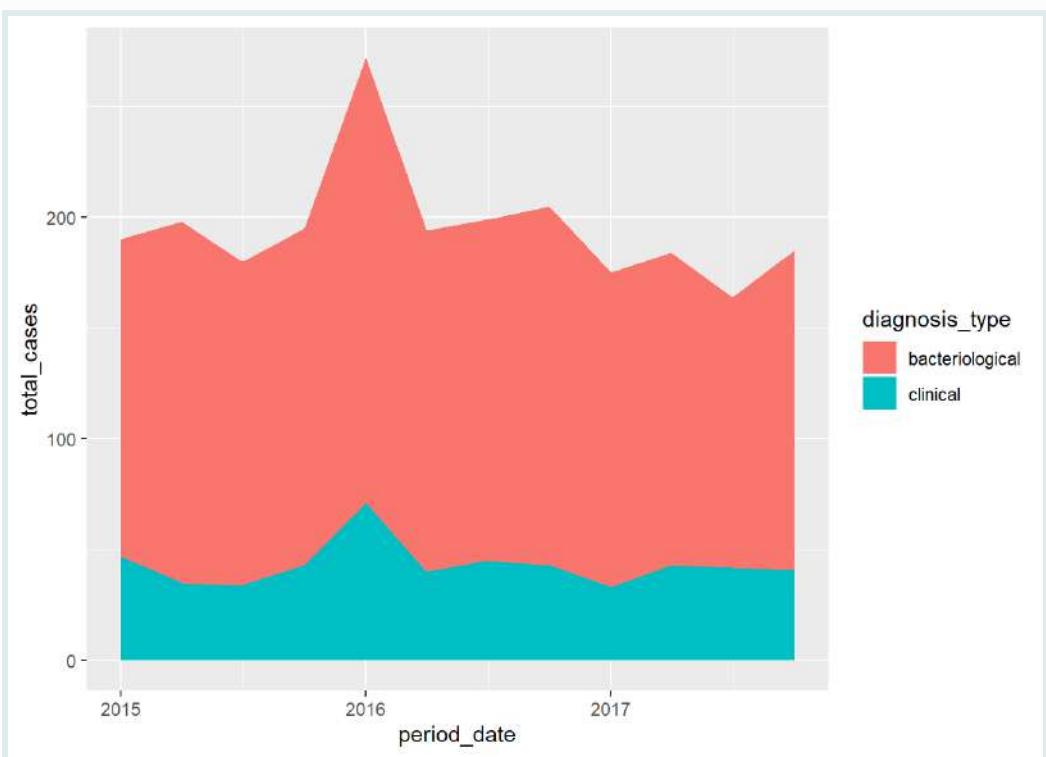
```
# Area plot of treated cases over time
outcome_by_period %>%
  ggplot(
    aes(
      x = period_date,      # Map 'period' to the x-axis
      y = total_cases,      # Map 'value' to the y-axis
      fill = outcome        # Map 'variable_name' to the fill aesthetic
    )) +
  # geom_area() creates an area plot
  geom_area()
```



Suppose we now want to ask what outcome contributes most to each section, or track the proportion of cured cases in each period over time. These questions also involve comparisons, but they are subtly different. Instead of comparing total values, we want to compare the relative contribution of those values to a bigger whole as proportions or percentages. In other words, we are interested in looking at the composition, or part-to-whole relationships. In the next section we will learn how to visualize these relationships.



```
# PQ 3 answer
tb_outcomes %>%
  group_by(period_date, diagnosis_type) %>%
  summarise(
    total_cases = sum(cases, na.rm = T)) %>%
  ggplot(
    aes(
      x = period_date,
      y = total_cases,
      fill = diagnosis_type)) +
  geom_area()
```



~~Visualizing~~ comparisons with normalized bar charts, pie charts, and donut charts

With compositions, we want to show how individual parts make up the whole. We could try to answer these questions using charts types discussed above, but there are many chart types devoted to compositions that do a much better job. These part-to-whole chart types immediately focus our visual attention on the relative importance of every part to the total value in the data.

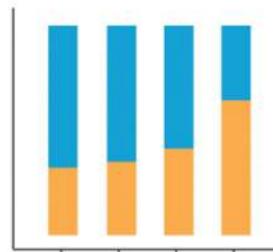
Charts for visualizing compositions



Pie chart



Donut chart



Stacked bar chart

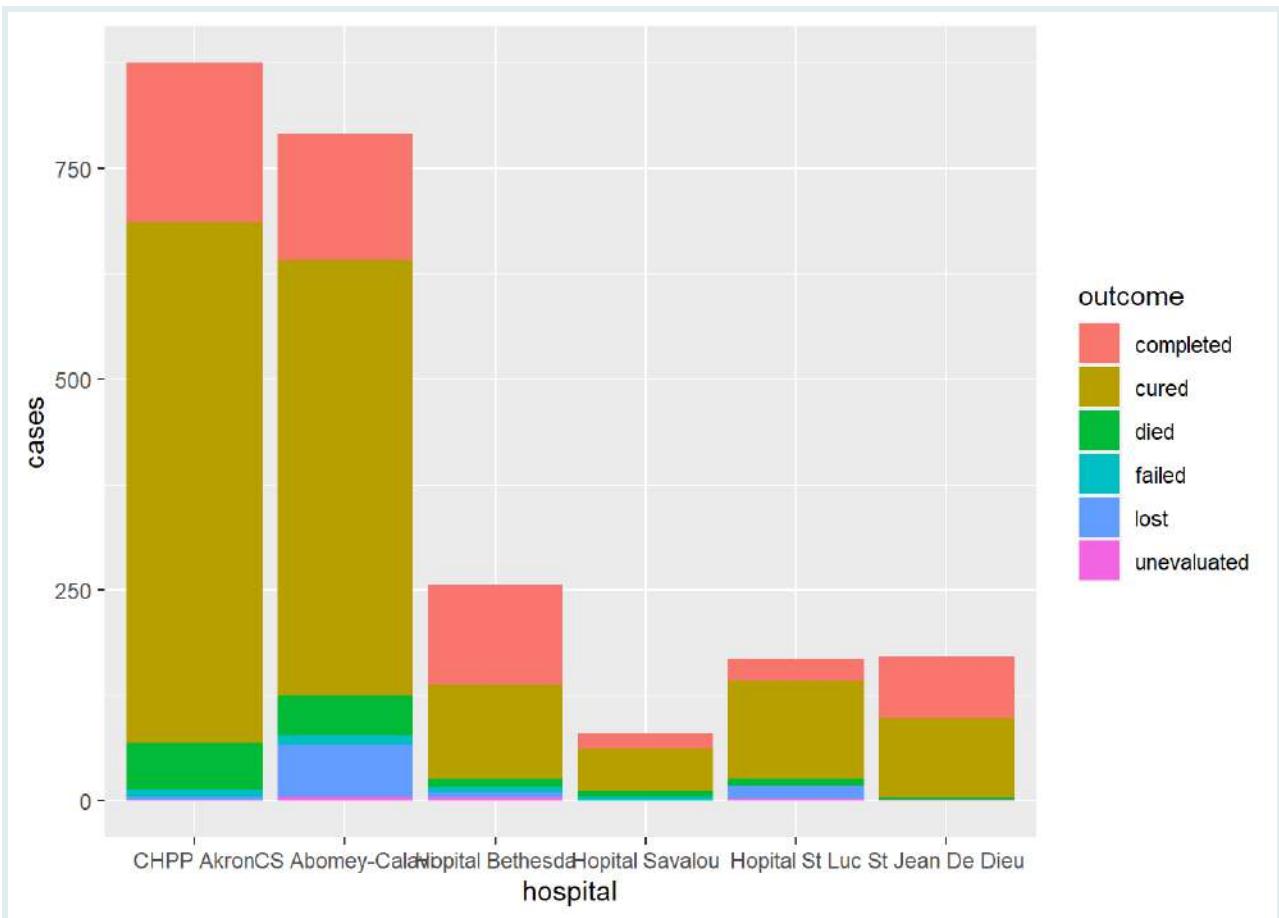
A selection of chart types for visualizing parts-to-whole relationships.

Percent-stacked bar chart

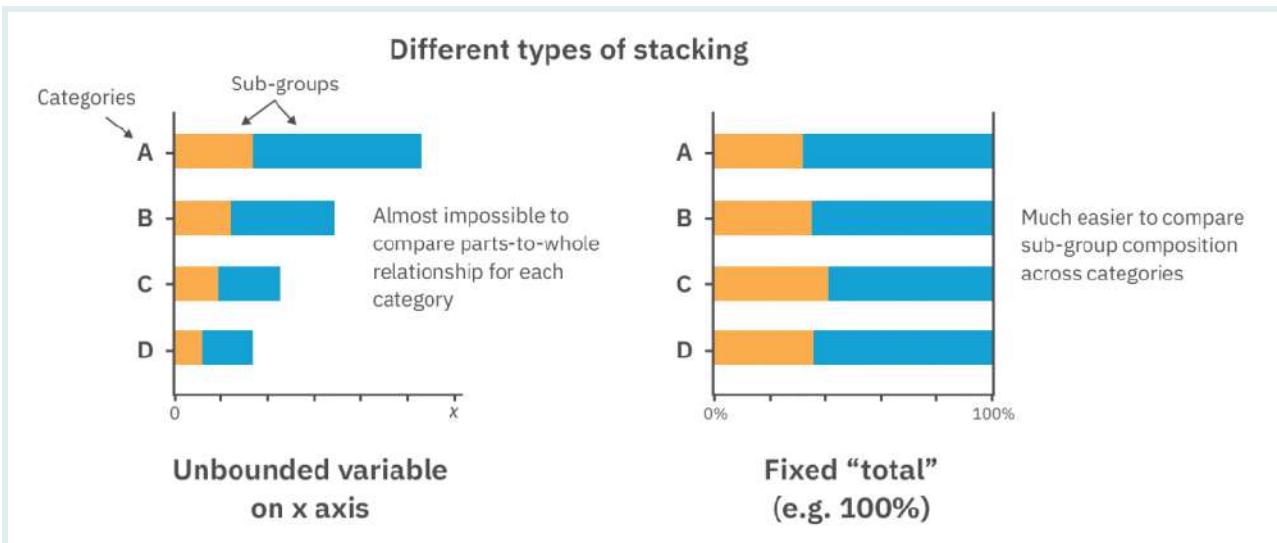
To visualize a composition, or part-to-whole relationship, we need two ingredients: the parts, and the whole.

The stacked bar charts we created earlier do a somewhat okay job of this:

```
# Regular stacked bar plot
tb_outcomes %>%
  ggplot(
    aes(x = hospital,
        y = cases,
        fill = outcome)) +
  geom_col()
```

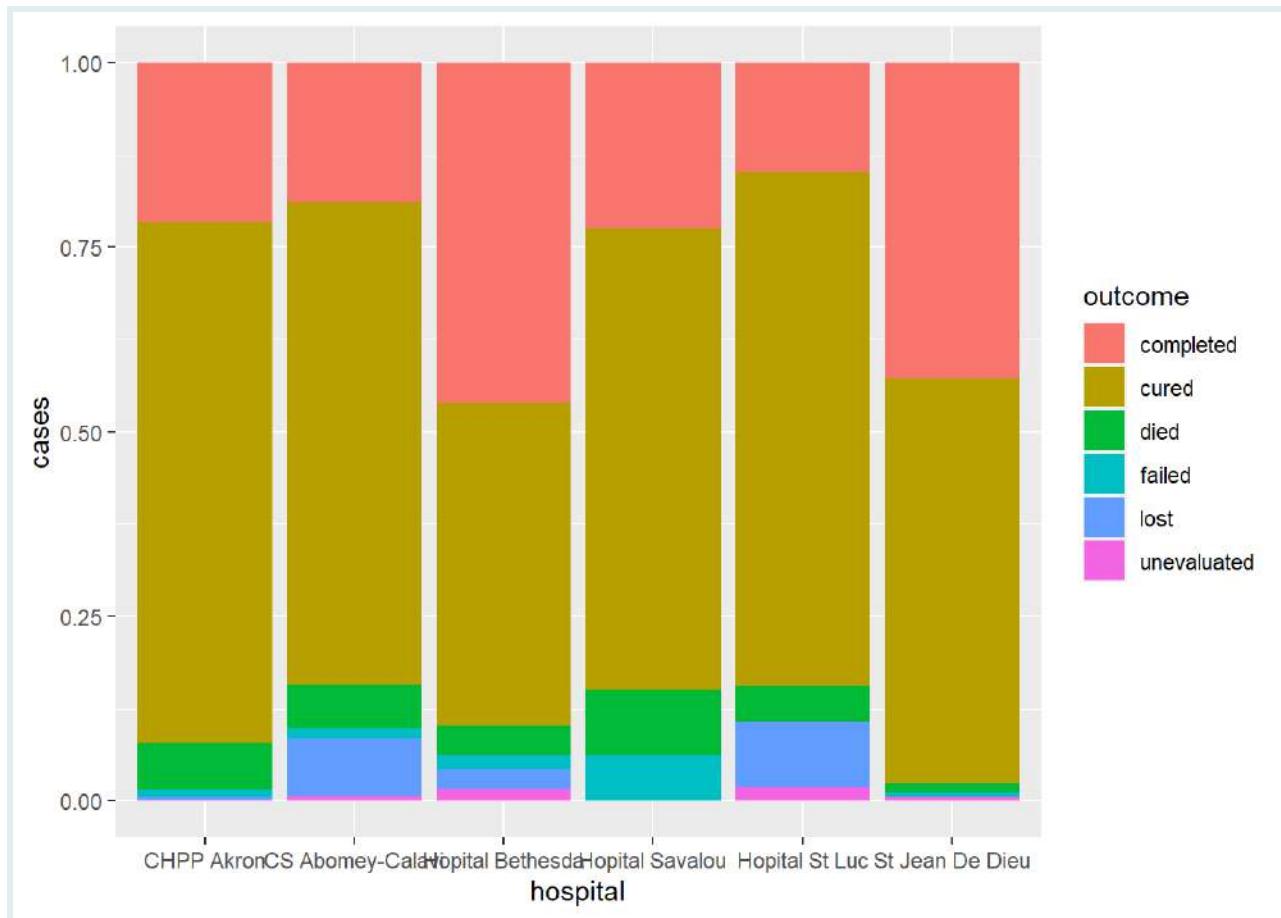


- They show us parts-of-wholes, but all the wholes are different sizes.
- The height of the bars represent the total number of cases, which is different at every location.
- Looking at the *relative* distribution of outcomes would be much easier if every bar were the same size.
- We can do this by creating a 100% stacked bar chart, where the total height of each bar is standardized to the same size, effectively showing proportions rather than counts or absolute values.



This is achieved by setting the `position` argument to "fill" in `geom_col()`.

```
# Percent-stacked bar plot
tb_outcomes %>%
  ggplot(
    aes(x = hospital,
        y = cases,
        fill = outcome)) +
  # Add position argument for normalized bars
  geom_col(position = "fill")
```



- All bars are now the same length, meaning all the wholes are now the same size. This now allows us to easily evaluate the contributions of the different parts to the whole.

Circular plots: Pie and Donut charts

In this section, we will delve into circular data visualizations, particularly pie charts and donut plots, to demonstrate categorical data distribution. These types of plots can be quite polarizing in the data visualization community due to their tendency to distort data interpretation. However, when employed judiciously, they can offer an intuitive snapshot of proportions within a dataset.



- Delve into donut and pie charts with a note of caution: they're visually enticing but can mislead.
- Recognize that bar plots often surpass these charts in delivering precise data interpretations.
- Understand that our brains prefer comparing lengths (like bars) over angles or areas (like pie slices).
- Remember, as data categories grow, pie and donut charts can get overly crowded and lose clarity.

WATCH OUT

- Acknowledge their inability to effectively display changes over time, unlike the adept bar plot.
- Use pie and donut charts sparingly, with a mindful eye on their potential to obscure data's true story.

Before we can visualize the data, we must first aggregate it to get the total counts for each treatment outcome category, ensuring we have a clear representation of each segment of our dataset.

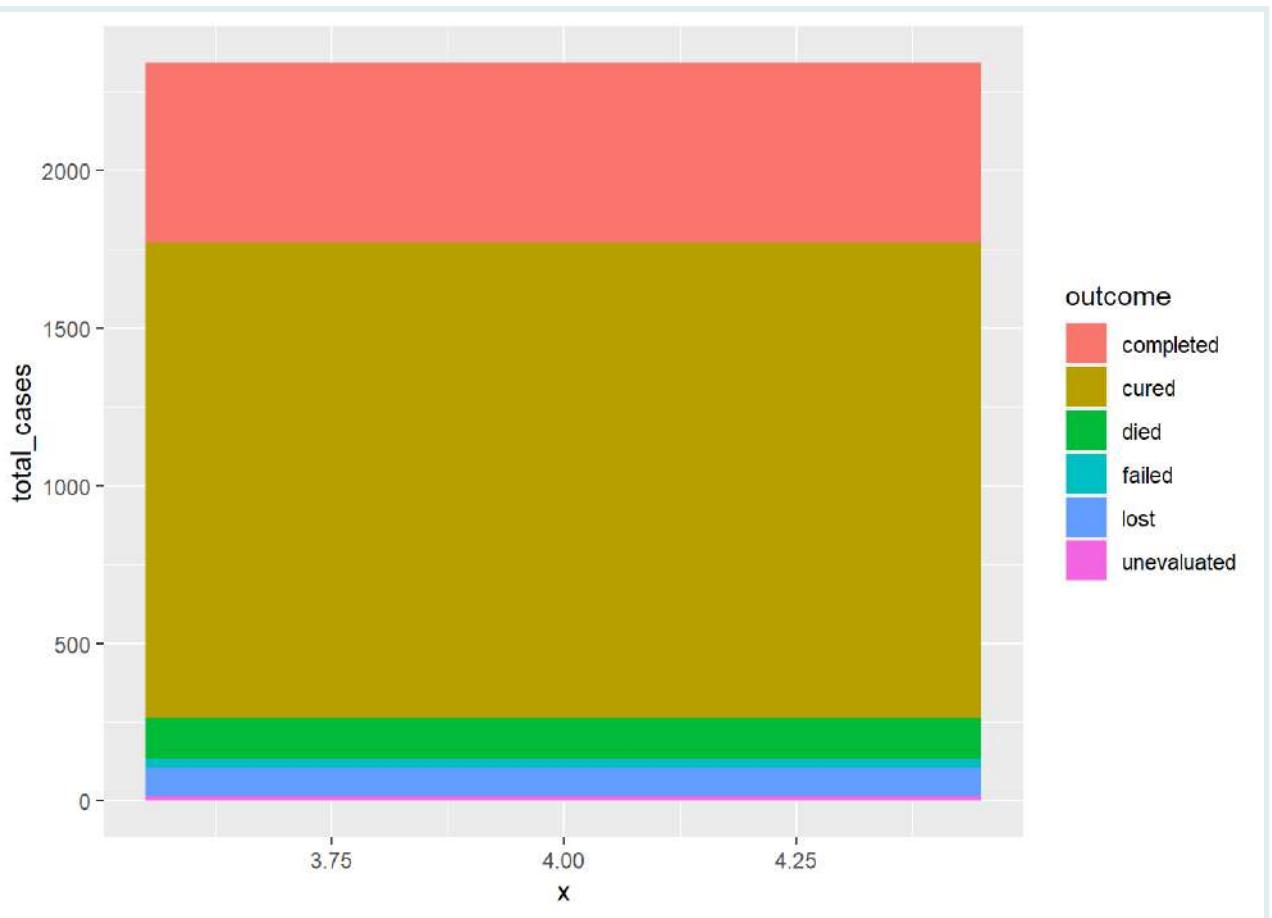
```
outcome_totals <- tb_outcomes %>%
  group_by(outcome) %>%
  summarise(
    total_cases = sum(cases, na.rm = T))

outcome_totals
```

```
## # A tibble: 6 × 2
##   outcome      total_cases
##   <chr>          <dbl>
## 1 completed      573
## 2 cured          1506
## 3 died            130
## 4 failed          30
## 5 lost             87
## 6 unevaluated     15
```

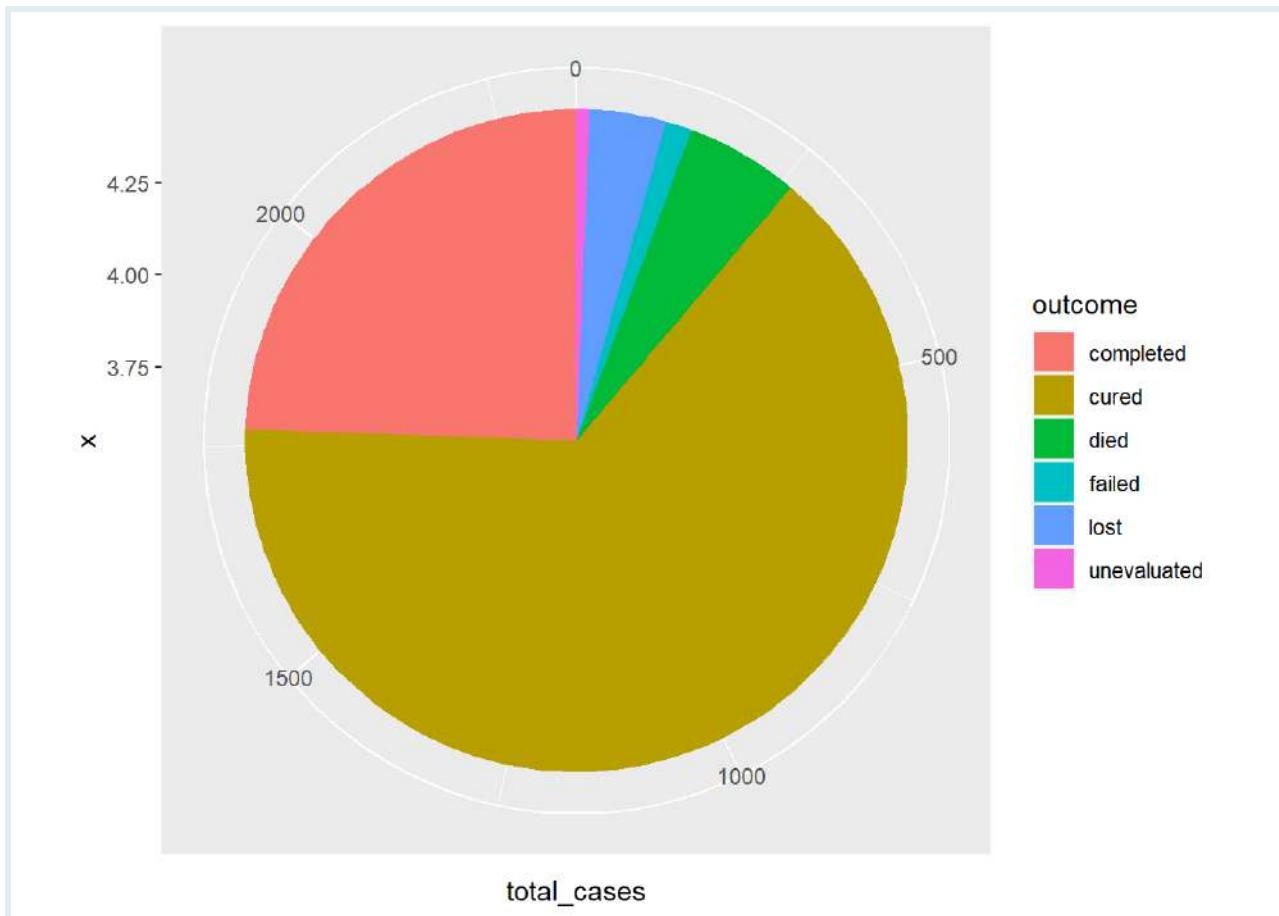
A pie chart is basically a round version of a single 100% stacked bar.

```
# Single-bar chart (precursor to pie chart)
ggplot(outcome_totals,
       aes(x=4, # Set arbitrary x value
           y=total_cases,
           fill=outcome)) +
  geom_col()
```



- In ggplot2, we'll explore how `coord_*` functions can change a plot's perspective, like tweaking aspect ratios or axis limits.
- We'll transform our plot from linear to polar coordinates using `coord_polar()`, which will shape our data into slices for a pie chart.
- By mapping the `y` aesthetic to angles (using the `theta` argument), we'll collaboratively create a visual that clearly displays the distribution of our categorical data.

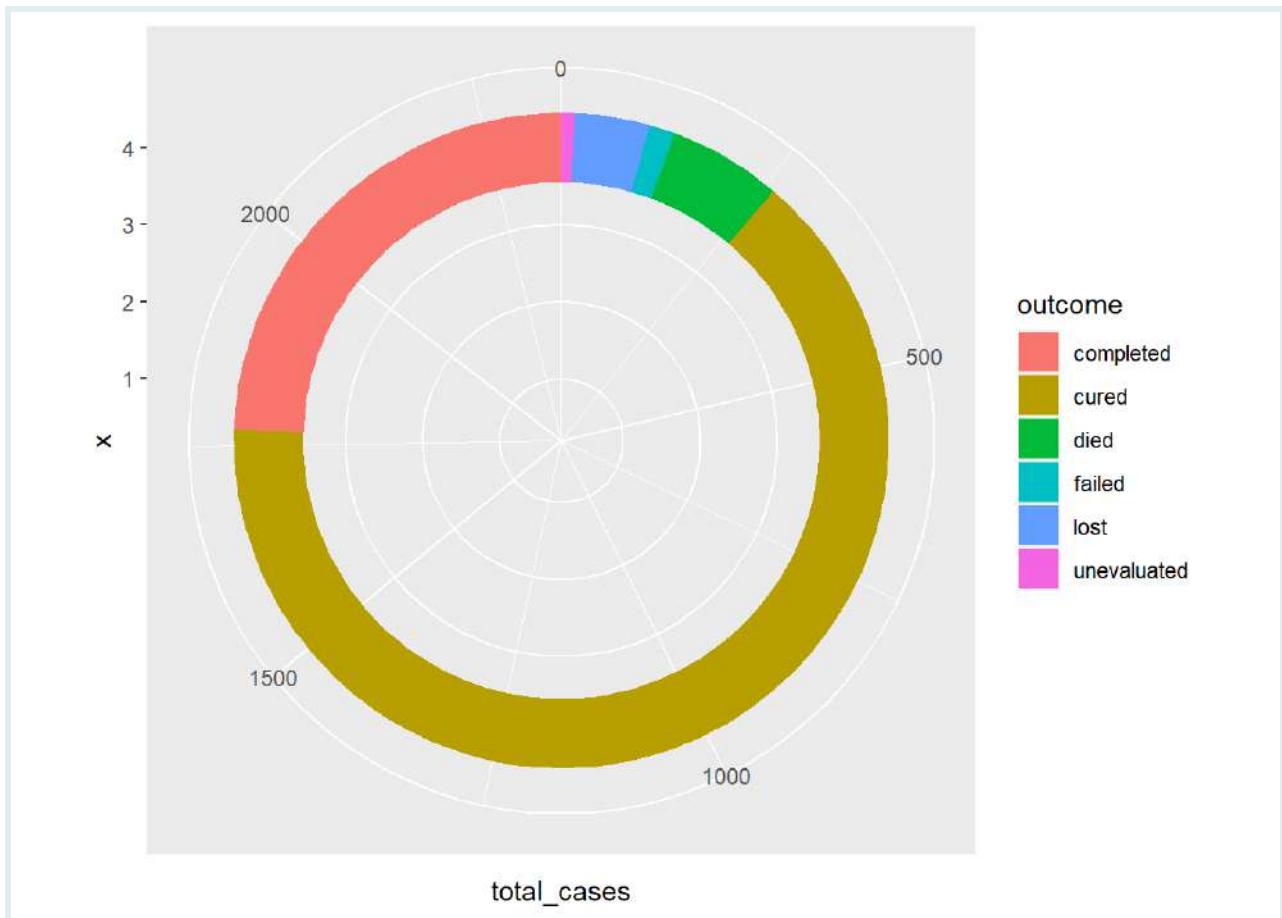
```
# Basic pie chart
ggplot(outcome_totals,
       aes(x=4,
           y=total_cases,
           fill=outcome)) +
  geom_col() +
  coord_polar(theta = "y") # Change y axis to be circular
```



- Donut charts, or ring charts, offer a compelling twist on traditional pie charts and are within our reach using ggplot2 in R.
- We build these charts with a process akin to pie charts, but we add distinctive touches.
- By employing `geom_col()` we lay the groundwork with a bar chart.
- The transformation into a circle comes next with `coord_polar(theta = "y")`.
- With `xlim(c(0.2, 4 + 0.5))`, we're setting the stage for the donut's signature feature: its central void.
- The lower limit of `0.2` carves out the space for the donut's hole, while `4.5` on the upper end ensures every category has its place.

A donut or ring chart can be created using `geom_col()`, `coord_polar(theta = "y")` and setting a x-axis limit with `xlim` like this:

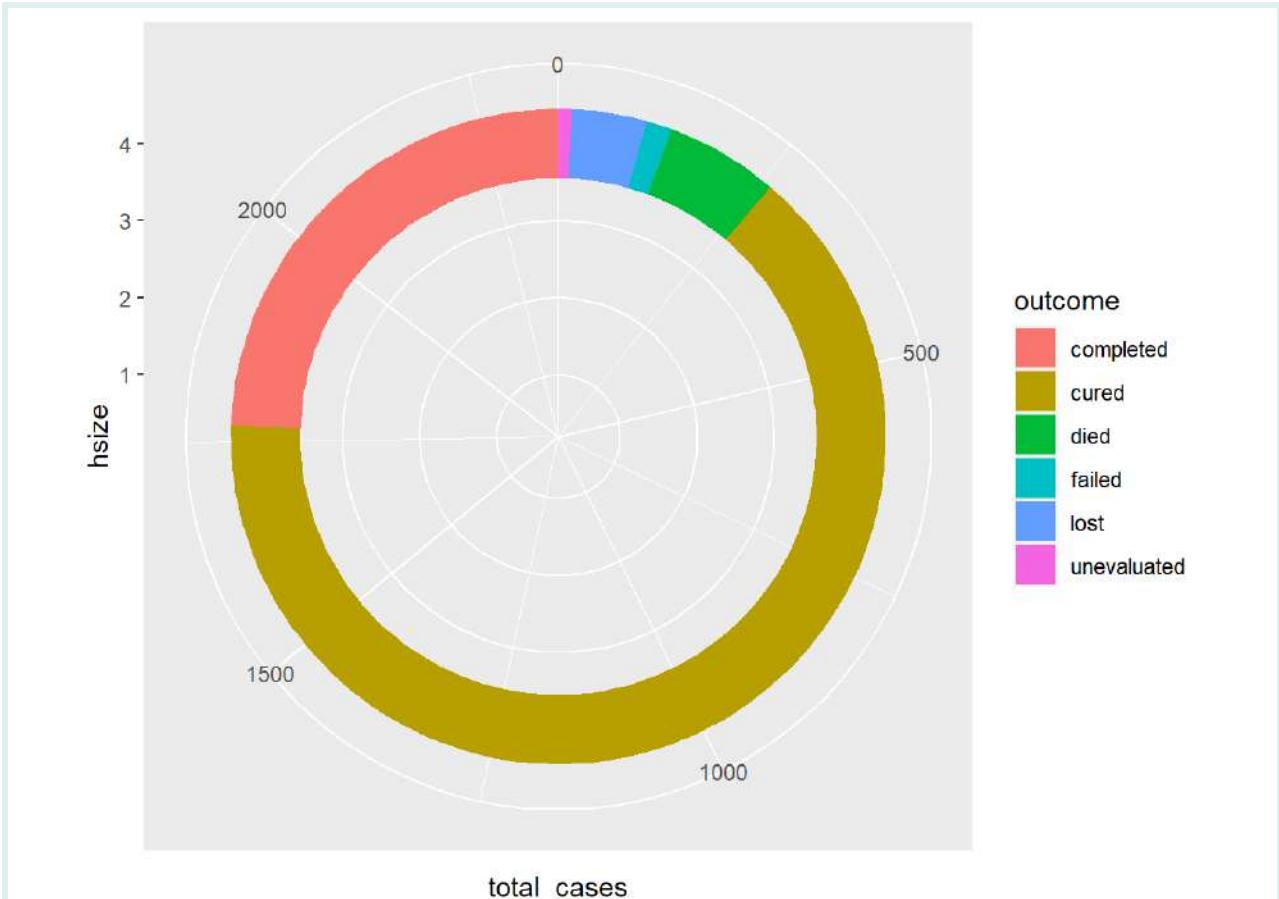
```
# Make it a donut chart
ggplot(outcome_totals,
       aes(x=4,
           y=total_cases,
           fill=outcome)) +
  geom_col() +
  coord_polar(theta = "y") +
  xlim(c(0.2, 4 + 0.5)) # Set x-axis limits
```



- The x value controls hole size.
- Replace 4 with placeholder `hszie` which can be varied.
- The bigger the value the bigger the hole size. Note that the hole size must be bigger than 0.

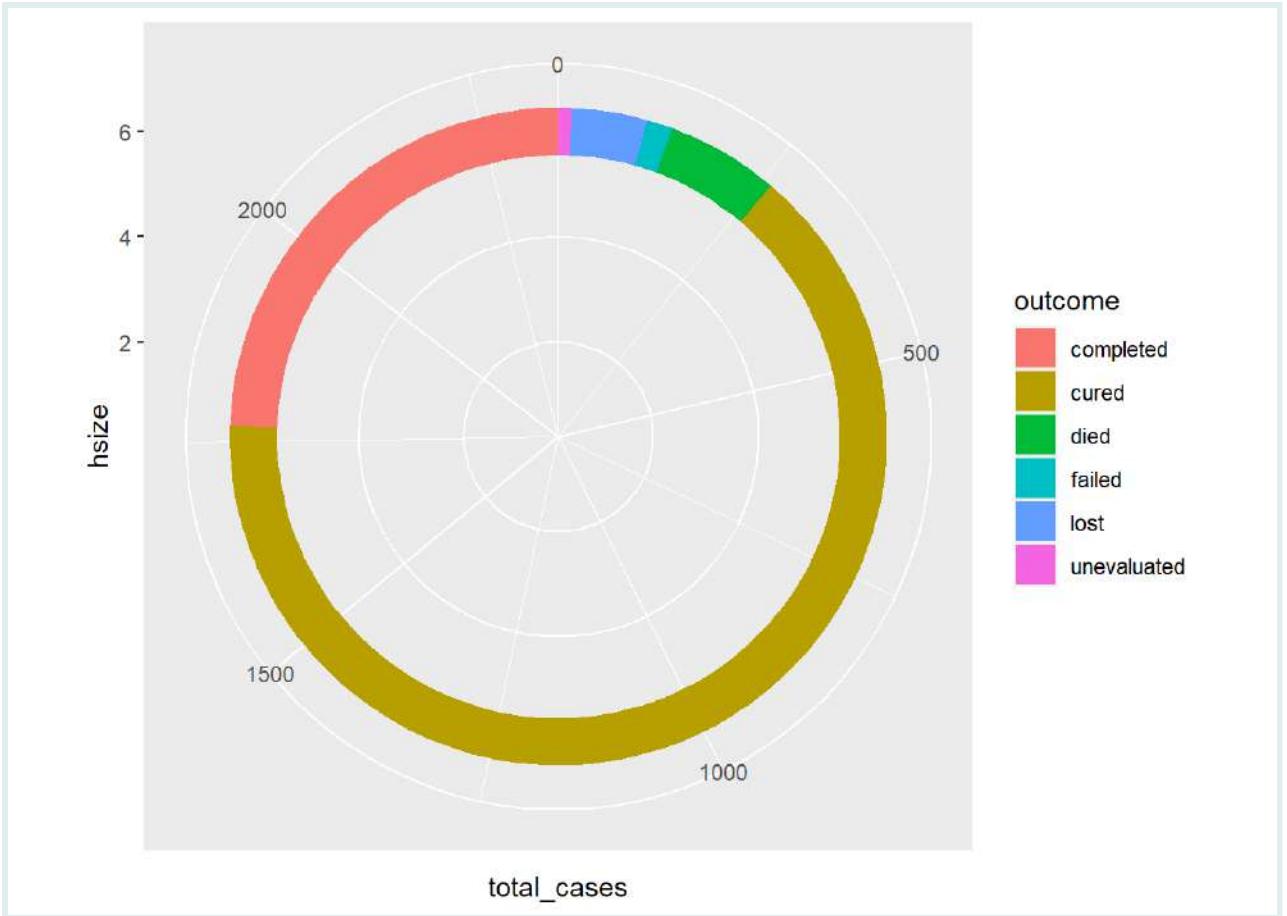
```
# Set hole width
hszie <- 4

ggplot(outcome_totals,
       aes(x=hszie, y=total_cases, fill=outcome)) +
  geom_col() +
  coord_polar(theta = "y") +
  xlim(c(0.2, hszie + 0.5))
```



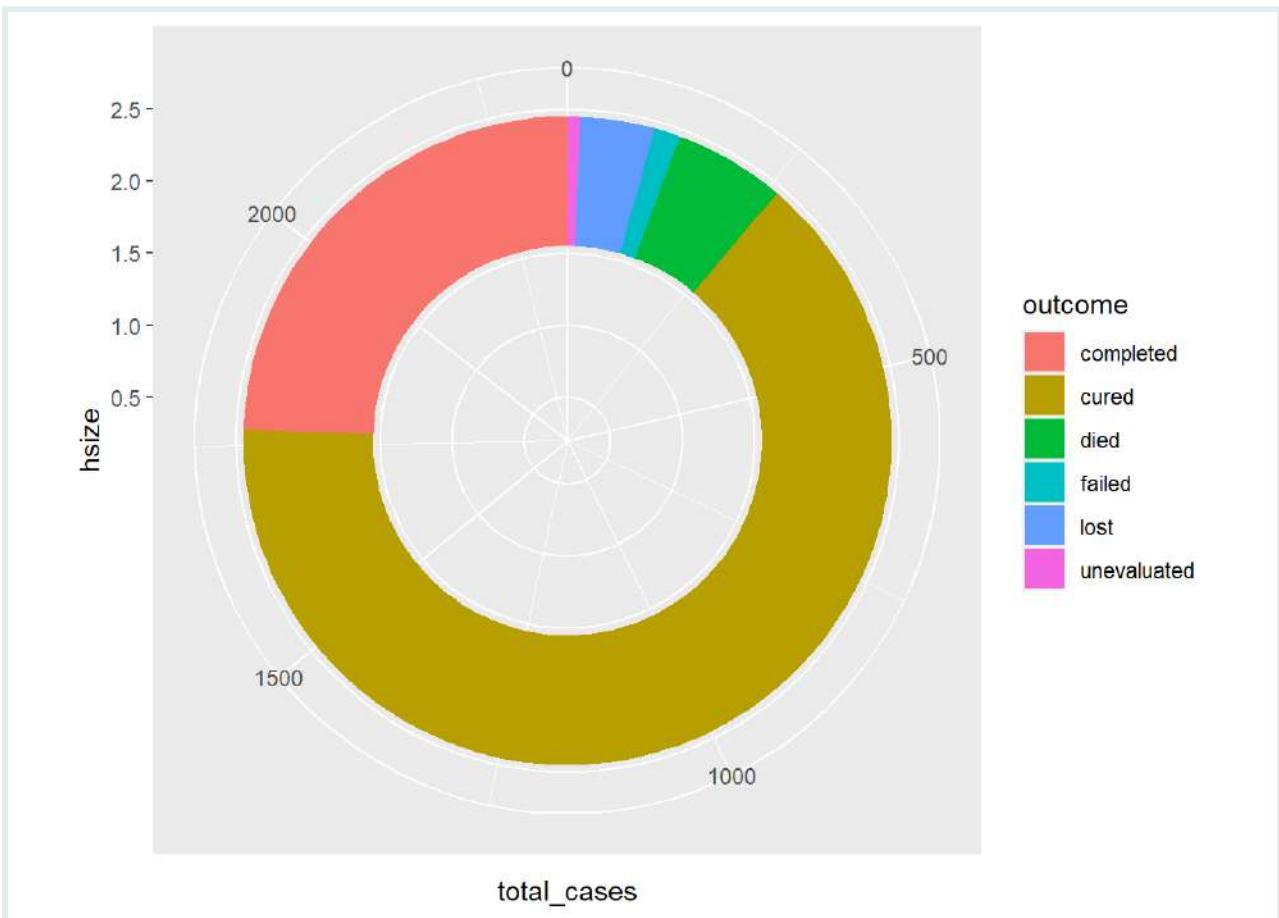
```
# Increase the value to make the hole bigger
hsizes <- 6

ggplot(outcome_totals, aes(x=hsizes, y=total_cases, fill=outcome)) +
  geom_col() +
  coord_polar(theta = "y") +
  xlim(c(0.2, hsizes + 0.5))
```



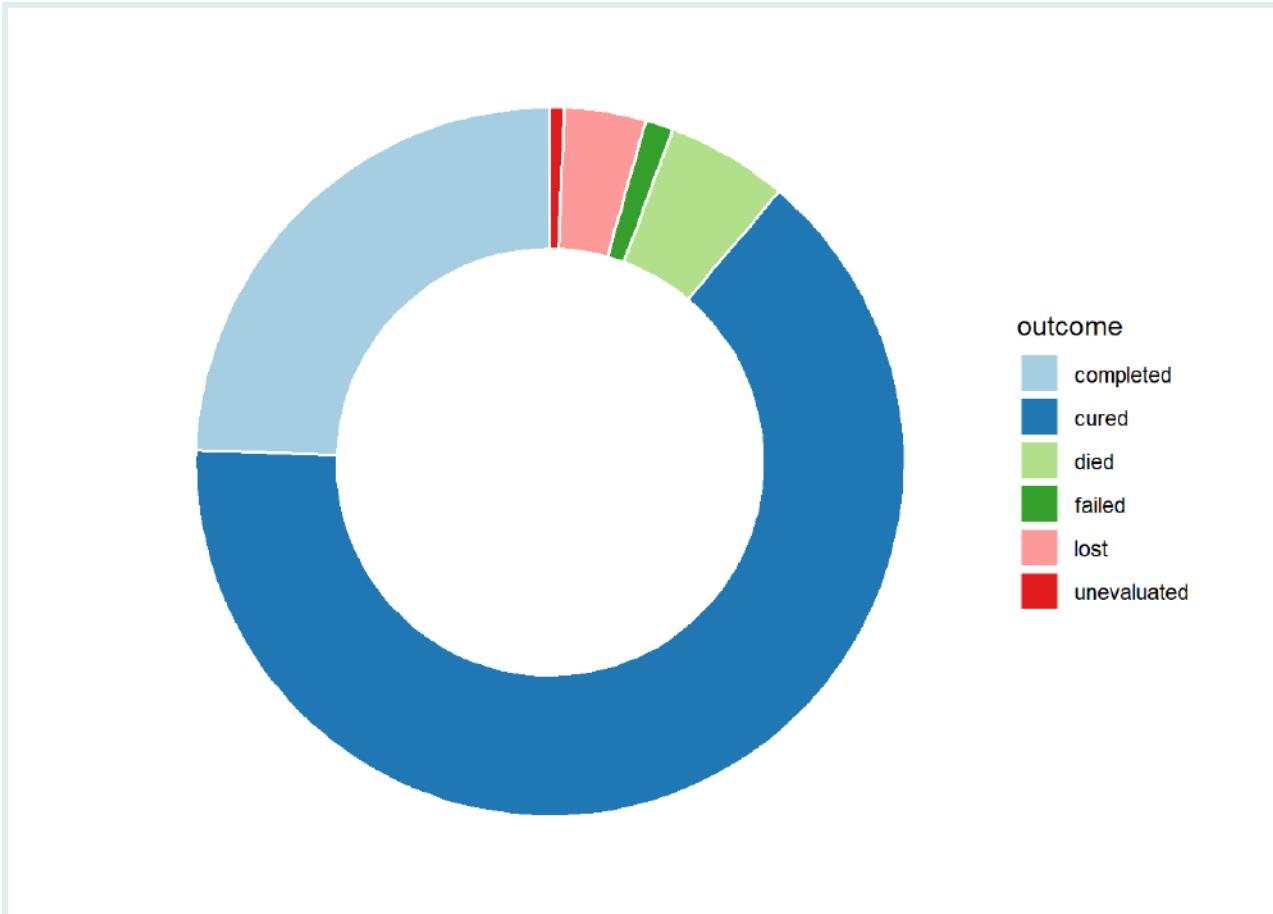
```
# Decrease the value to make the hole smaller
hsizes <- 2

ggplot(outcome_totals, aes(x=hsizes, y=total_cases, fill=outcome)) +
  geom_col() +
  coord_polar(theta = "y") +
  xlim(c(0.2, hsizes + 0.5))
```



The border, fill colors and the theme of the donut plot can be customized in several ways. The code below illustrates some customization options:

```
# Improved aesthetics
ggplot(outcome_totals,
       aes(x=hsizes, y=total_cases, fill=outcome)) +
  geom_col(color = "white", linewidth = 0.7) + # Add white borders
  coord_polar(theta = "y") +
  xlim(c(0.2, hsizes + 0.5)) +
  scale_fill_brewer(palette = "Paired") + # Set color palette
  theme_void() # Remove background, grid, numeric labels
```



RECAP



What do stacked bars, pies, and donuts have in common? They are all types of parts-to-whole charts! Along with making comparisons, visualizing what things are composed of is one of the most important applications of data visualization. These visualizations work for data that is grouped into multiple subcategories, where our aim is to see how much of the total each category makes up. Examining these “parts-to-whole” relationships can provide insight into demographics, budget allocation, levels of agreement, and more.

PRACTICE



Practice Question: The whole picture

Using the `tb_outcomes` dataset in R, create a visual representation that compares the proportion of tuberculosis cases by outcome across different types of diagnoses. Your visualization should:

1. Group the data by `outcome` and `diagnosis_type`.

2. Summarize the total number of `cases` for each group, accounting for possible missing values (`NAs`).
3. Display this information in a series of donut charts, one for each diagnosis type, showing the proportion of each outcome within the diagnosis type.
4. Ensure the charts have a clean look with `white` borders around the segments and no background or gridlines.
5. Arrange the individual donut charts for each diagnosis type using `facet_wrap`.

Write the R code to create this visualization.



```
# PQ4 answer
tb_outcomes %>%
  group_by(-----, -----) %>%
  summarise(
    total_cases = sum(-----, -----) ) %>%
ggplot(aes(x=-----, y=-----, fill=-----)) +
  geom_col(color = -----, position = -----) +
  coord_polar(theta = "y") +
  xlim(c(0.2, ----- + 0.5)) +
  theme_void() +
  -----  
  

## Error: <text>:3:18: unexpected ',',  

## 2: tb_outcomes %>%  

## 3:   group_by(-----,  

##               ^
```

Wrap Up!

Solutions

Contributors

References

Some material in this lesson was adapted from the following sources:

- Horst, Allison. "Allisonhorst/Dplyr-Learnr: A Colorful Introduction to Some Common Functions in Dplyr, Part of the Tidyverse." GitHub. Accessed April 6, 2022. <https://github.com/allisonhorst/dplyr-learnr>.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.

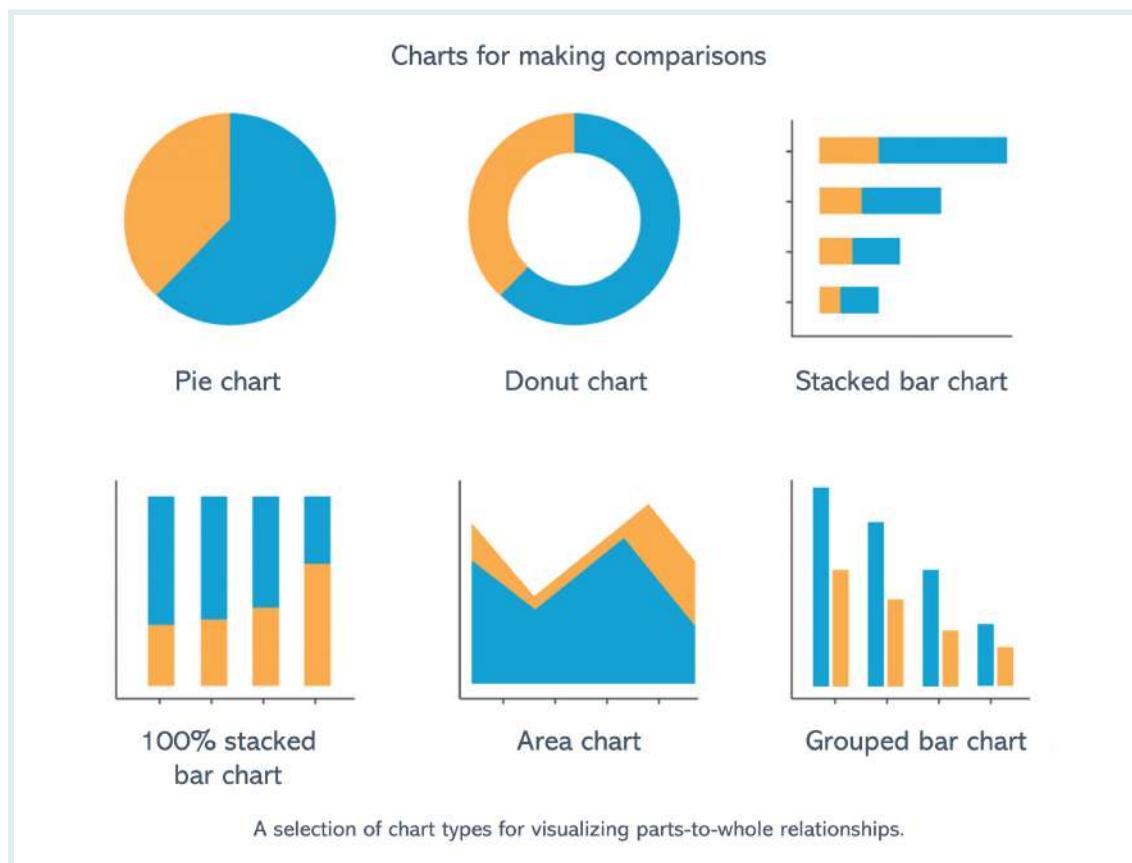


Plot Labels with ggplot2

Introduction
Learning Objectives
Packages
Introduction to text geoms in {ggplot2}
The <code>vjust</code> and <code>hjust</code> arguments
Understanding <code>hjust</code> (horizontal justification)
Understanding <code>vjust</code> (vertical justification)
Data Example: TB treatment outcomes in Benin
Labeling stacked bar plots
Labeling dodged bar plots
Labeling percent-stacked bar plots
Labeling circular plots
Wrap Up!
Solutions

Introduction

Bar plots are one the most common chart type out there and come in several varieties. In the previous lesson, we learned how to make bar plots and their circular counterparts with `{ggplot2}`.



In this lesson, we'll delve into the intricacies of labeling in `ggplot2`, focusing on `geom_label()` and `geom_text()` functions from `{ggplot2}`.

Learning Objectives

After this lesson, you will be able to:

1. **Use two different text geoms to label ggplots:**
 - `geom_text()` for simple labels
 - `geom_label()` for emphasized labels
 2. Appropriately transform and summarize data in the appropriate format for different chart types.
 3. Adjust text placement to position labels on stacked, Dodged, and percent-stacked bar plots.
 4. Adjust text placement to position labels on pie charts and donut plots.
-

Packages

Run the code below to load the packages for the lesson.

```
pacman::p_load(tidyverse, here, patchwork, medicaldata)
```

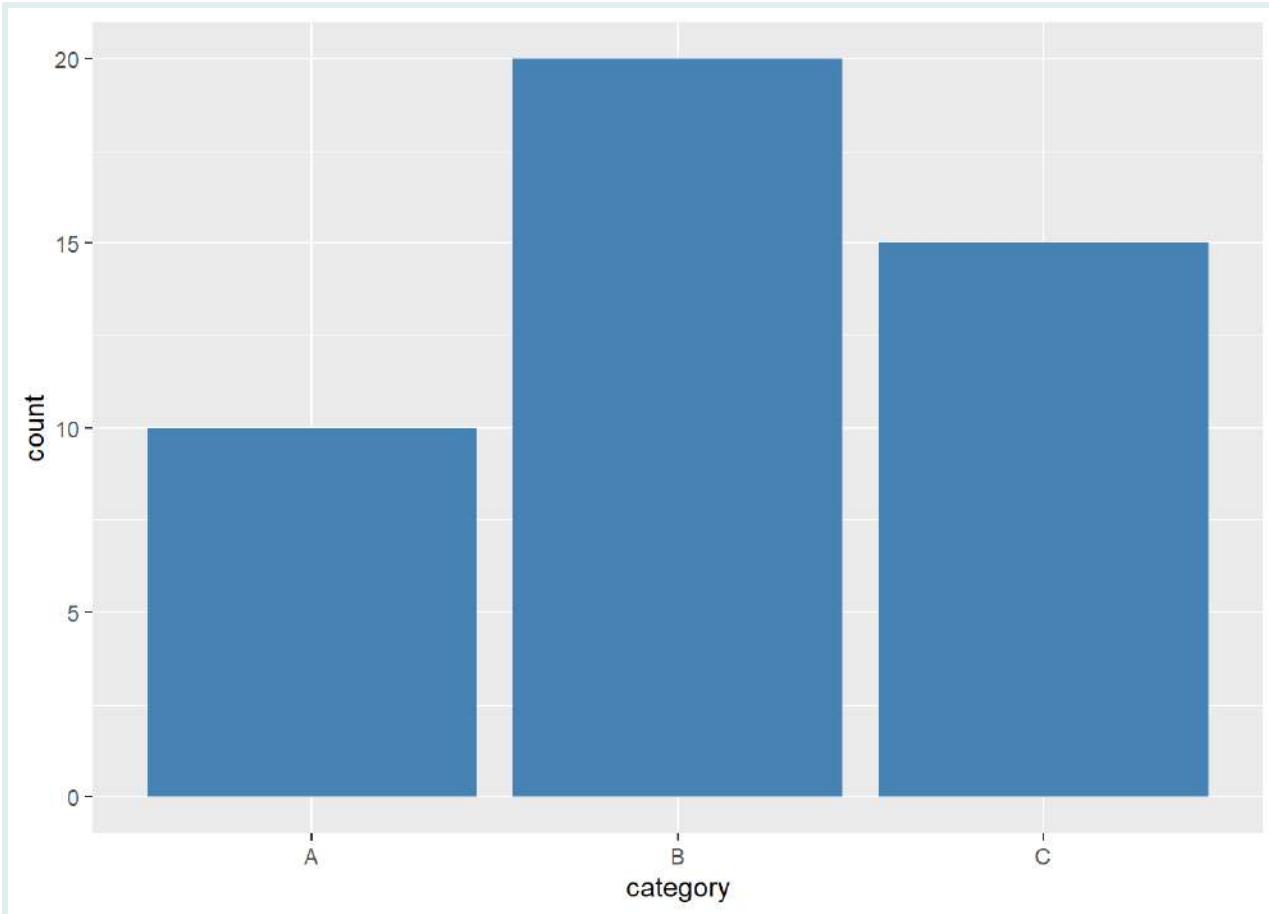
Introduction to text geoms in {ggplot2}

We'll start with `geom_text()` for simple labeling and then move to `geom_label()` for labels with more emphasis. We will show how to use these geoms on simple bar plots, then we will get into more details on how to leverage them for stacked bars, Dodged bars, normalized stacked bars, and circular plots.

First let's practice using these functions on a simple bar plot made with fake data. Once we cover the fundamentals of the labeling syntax, we will apply these to real epidemiology data.

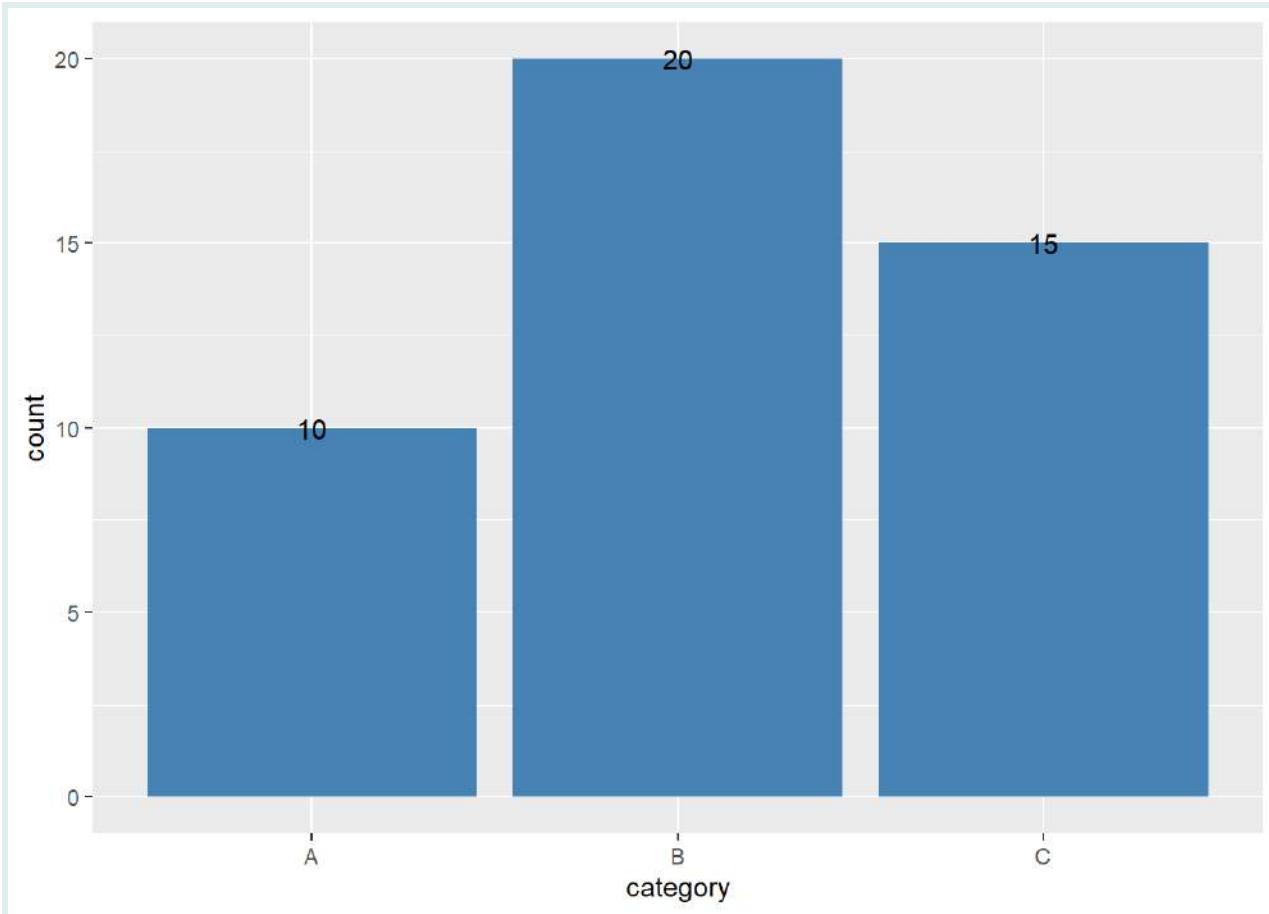
```
# Create example data frame
data <- data.frame(
  category = c("A", "B", "C"),
  count = c(10, 20, 15)
)

# Create the bar plot
ggplot(data, aes(x = category, y = count)) +
  geom_col(fill = "steelblue")
```



We can easily add labels to our bars with the `geom_text()` function and telling the `aes()` function which column to extract `label` text from:

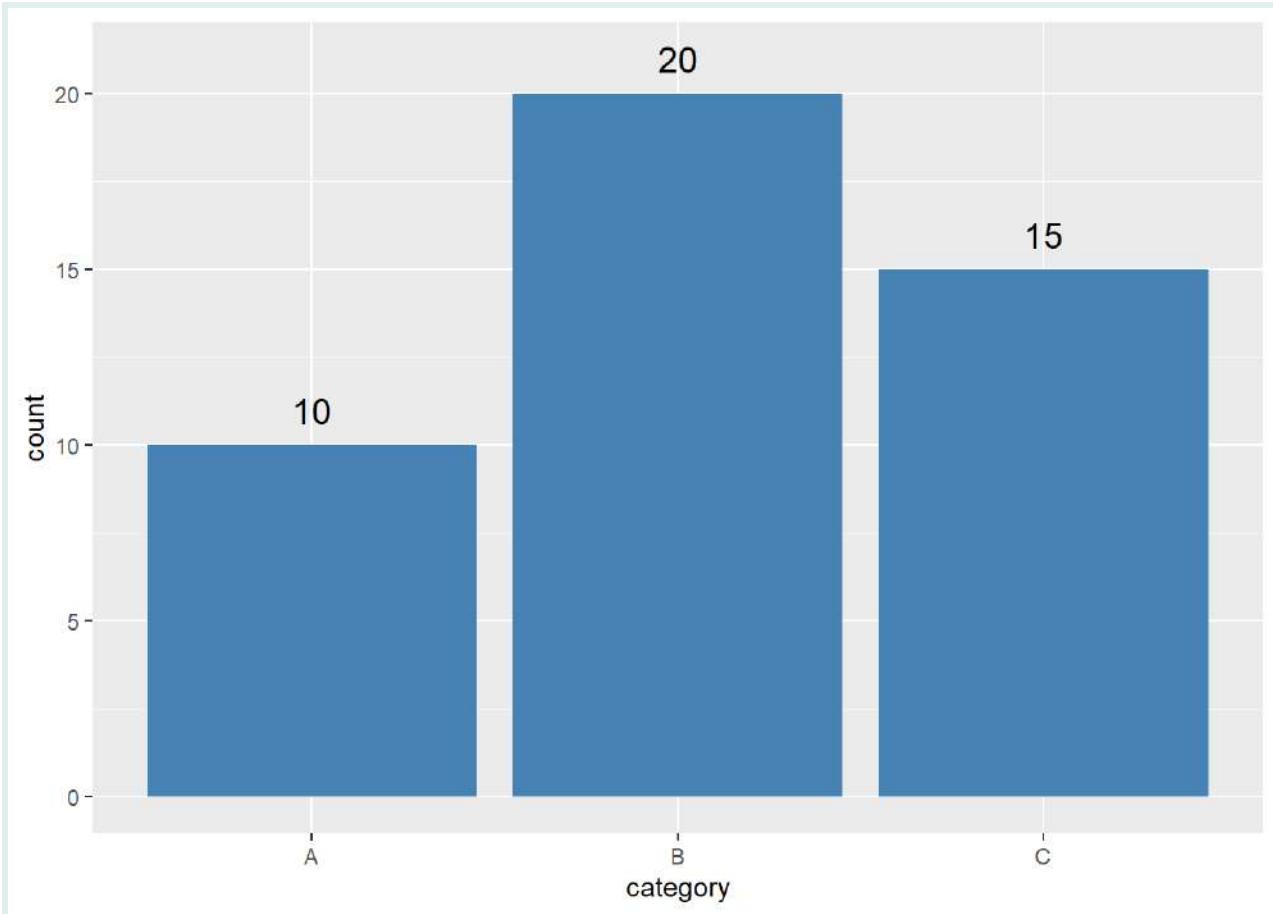
```
ggplot(data, aes(x = category, y = count)) +  
  geom_col(fill = "steelblue") +  
  geom_text(aes(label = count)) # provide variable to `label` argument
```



As you can see however, the placement of our text is odd – neither on the bar, nor under the bar. Additionally, they are quite small and difficult to make out. We can address this by making them bigger, and vertically adjusting their placement.

To do this, we will nudge the text upwards using the `y_nudge` argument. We will also increase the size of the text using the `size` argument.

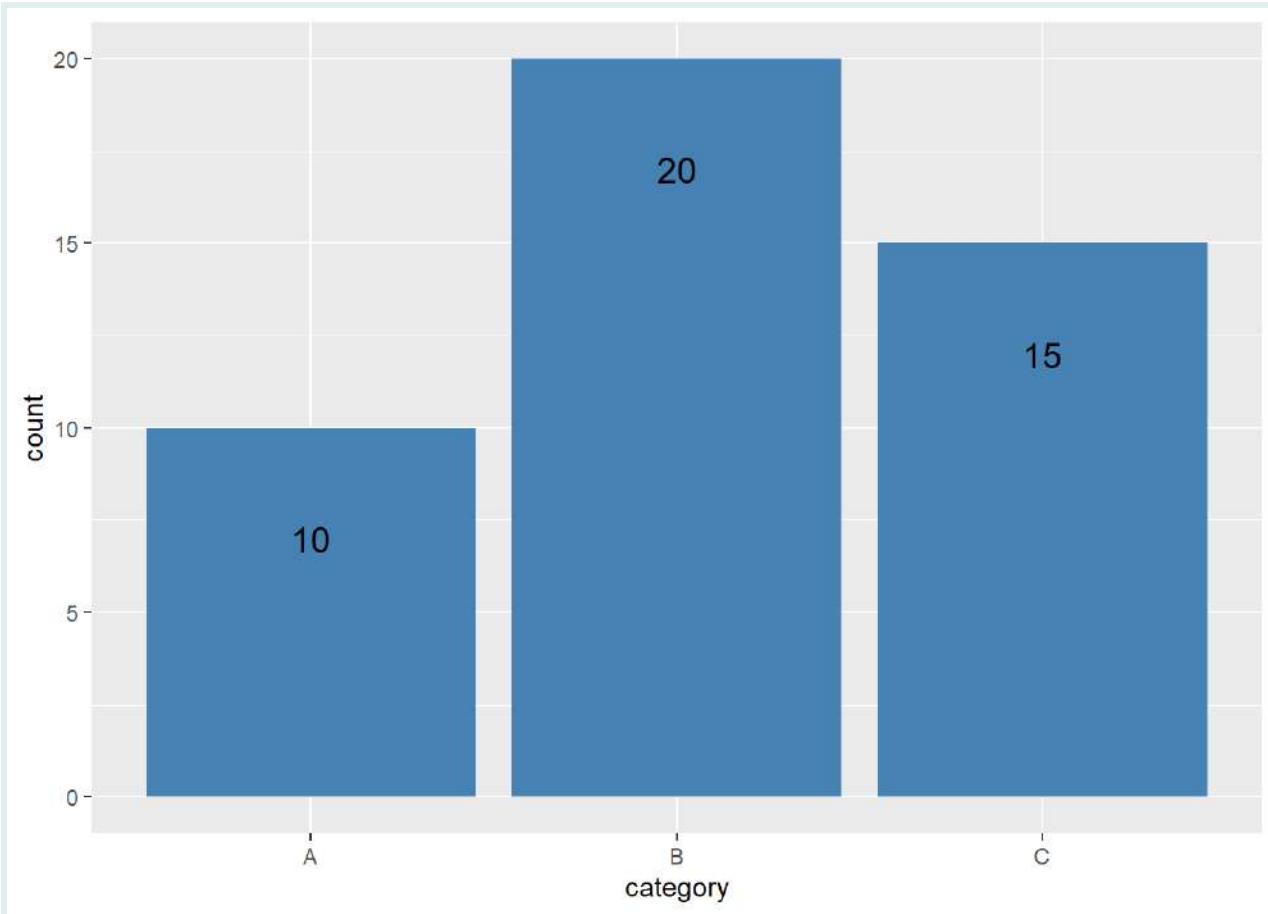
```
ggplot(data, aes(x = category, y = count)) +  
  geom_col(fill = "steelblue") +  
  geom_text(aes(label = count),  
            nudge_y = 1,  
            size = 5) # move text up
```



Note that the value of `nudge_y` is in the same units as the y-axis.

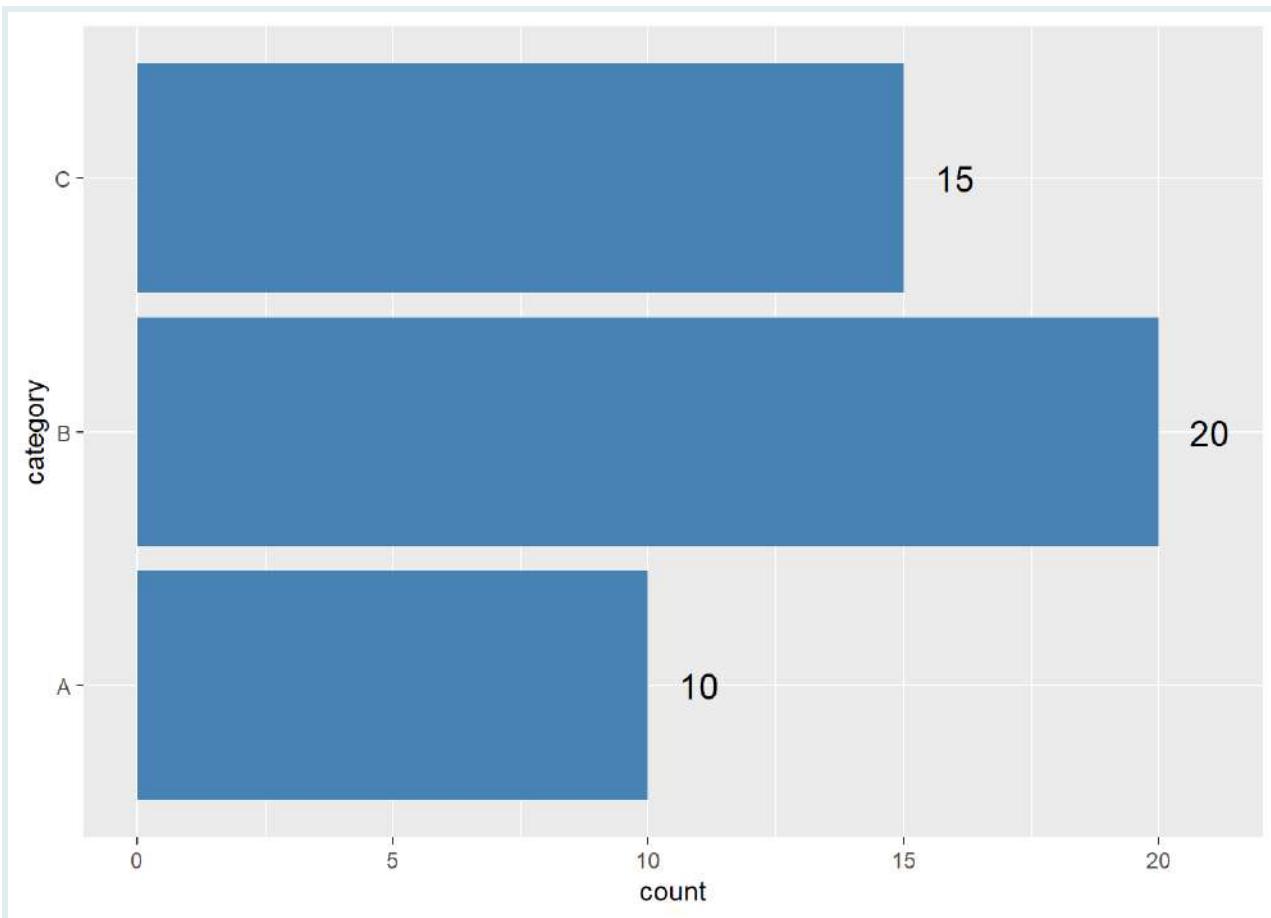
Let's try nudging the text down by setting `nudge_y` to a negative value:

```
ggplot(data, aes(x = category, y = count)) +  
  geom_col(fill = "steelblue") +  
  geom_text(aes(label = count),  
            nudge_y = -3,  
            size = 5) # move text down
```



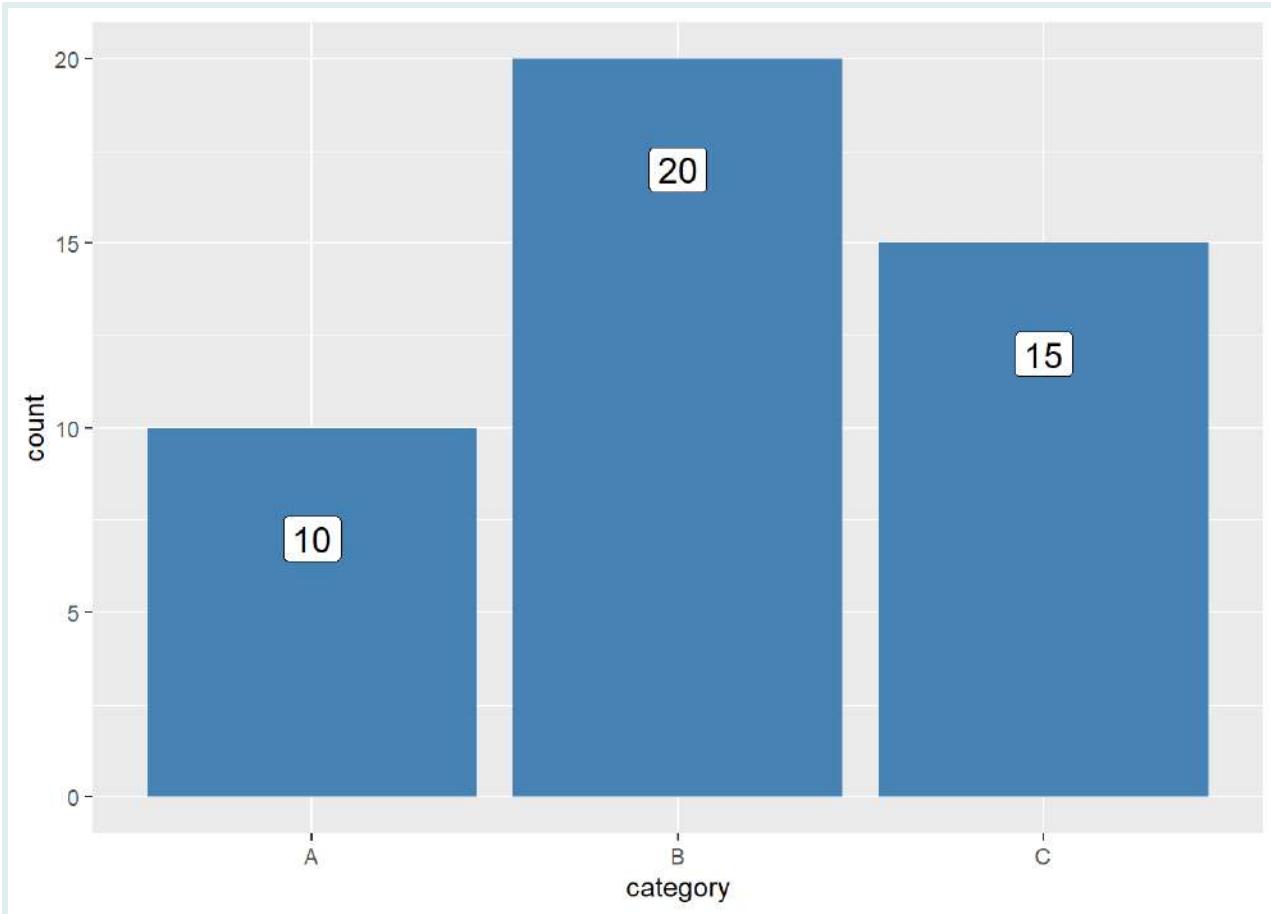
If we made a horizontal bar plot, we would need to nudge the text to the right or left using the `nudge_x` argument instead of `nudge_y`:

```
ggplot(data, aes(x = count, y = category)) +  
  geom_col(fill = "steelblue") +  
  geom_text(aes(label = count),  
            nudge_x = 1,  
            size = 5) # move text to the right
```



Now let's see how the `geom_label()` function works. We can use the same code as above, but replace `geom_text()` with `geom_label()`:

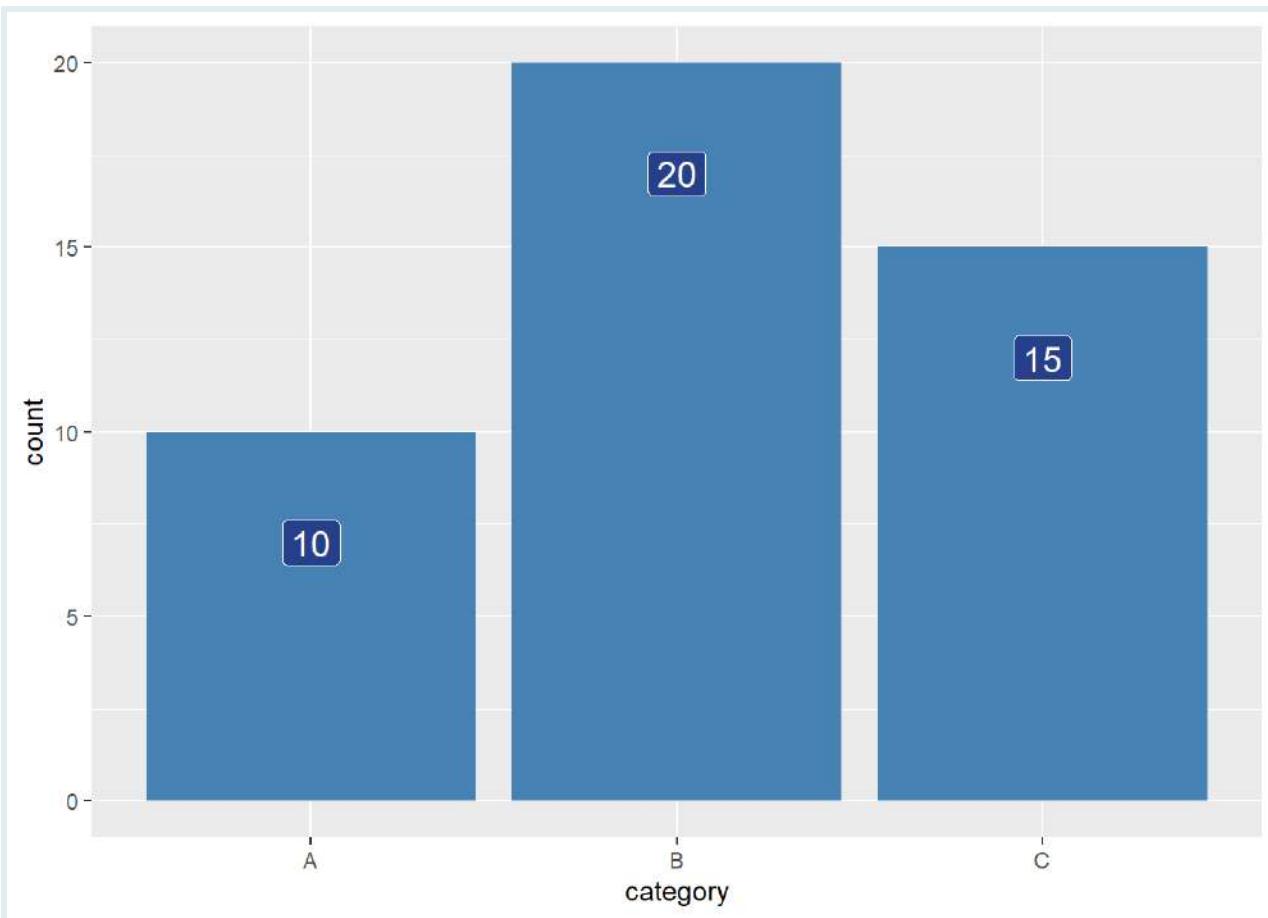
```
ggplot(data, aes(x = category, y = count)) +  
  geom_col(fill = "steelblue") +  
  geom_label(aes(label = count),  
            nudge_y = -3,  
            size = 5)
```



As you can see, `geom_label()` draws a rectangle behind the text, making it easier to read.

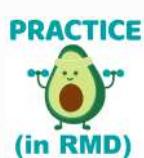
In this code, the `fill` aesthetic in `geom_label()` can be adjusted to control the background fill color of the labels. For example, let's make the background dark blue, and the text white:

```
ggplot(data, aes(x = category, y = count)) +  
  geom_col(fill = "steelblue") +  
  geom_label(aes(label = count),  
            nudge_y = -3,  
            fill = "royalblue4",  
            color = "white",  
            size = 5)
```



Q: Simple labeling

Consider the following sample data frame:



```
# Create example data frame
district_cases <- data.frame(
  district = c("A", "B", "C"),
  cases = c(10, 20, 15)
)

district_cases
```

```
##   district cases
## 1         A    10
## 2         B    20
## 3         C    15
```



Create a labeled bar plot of the data frame above, where the x-axis is the district and the y-axis is the number of cases. The labels should be the number of cases, and should be placed above the bars. The labels should have “darkblue” text with a “lightblue” background. The bar color should be “steelblue”



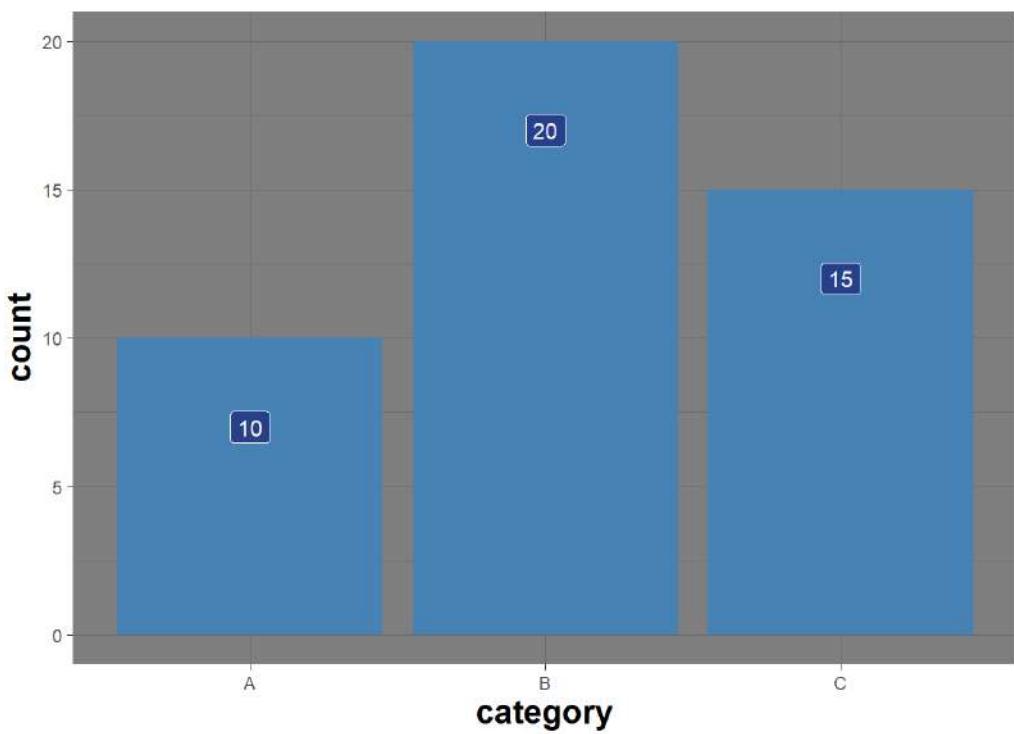
Setting a custom {ggplot2} theme So far, we’ve added a theme function to each of our bar plots. Let’s learn how to create our own custom theme functions, and how to use `theme_set()` function to set a global theme for all plots.

We’ll define a custom theme that is a combination of `theme_dark` and large bold axis labels:

```
theme_dark_custom <-
  theme_dark() +
  theme(
    axis.title = element_text(size = 16, face = "bold")
  )
```

Now we can set use this these for specific plot like this:

```
ggplot(data, aes(x = category, y = count)) +
  geom_col(fill = "steelblue") +
  geom_label(aes(label = count),
             nudge_y = -3,
             fill = "royalblue4",
             color = "white") +
  theme_dark_custom
```



PRO TIP



Note the lack of parentheses after `theme_dark_custom`.

We can set this theme as the default for all plots:

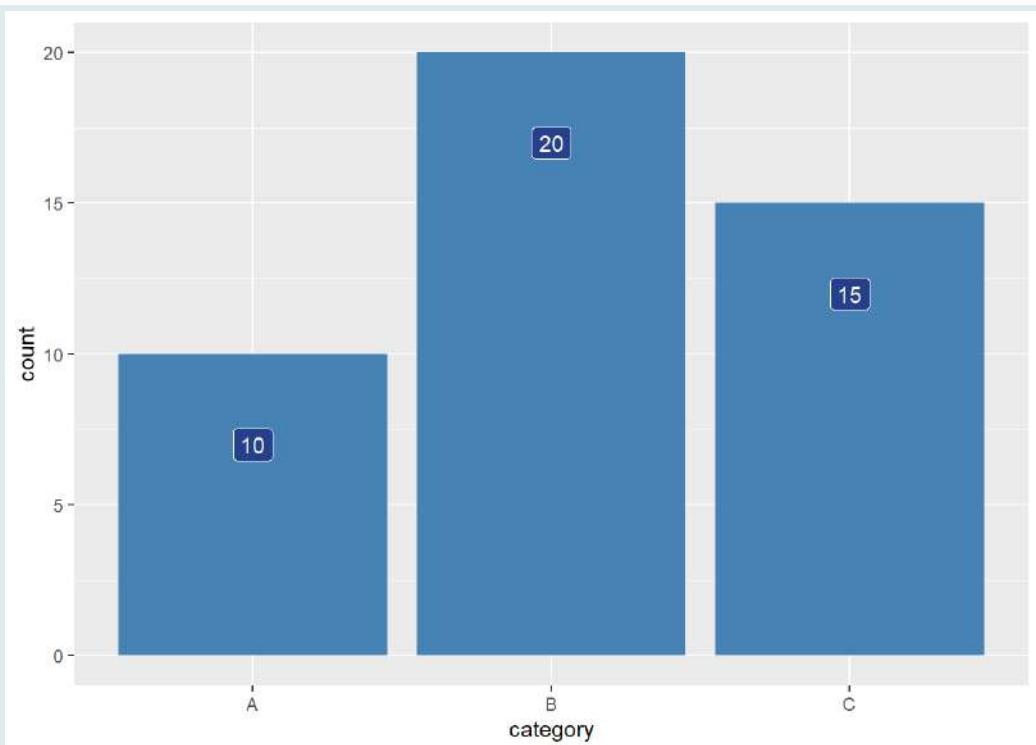
```
theme_set(theme_light_custom)
```

```
## Error in eval(expr, envir, enclos): object
'theme_light_custom' not found
```

Now `theme_light_custom()` will be automatically applied to every plot you draw.

For example, let's redraw the plot we made earlier:

```
ggplot(data, aes(x = category, y = count)) +
  geom_col(fill = "steelblue") +
  geom_label(aes(label = count),
             nudge_y = -3,
             fill = "royalblue4",
             color = "white")
```



This is a great way to ensure that all of your plots have a consistent look and feel.

To set the default theme back to the original, use
`theme_set(theme_gray())`.

```
theme_set(theme_gray())
```

The `vjust` and `hjust` arguments

Rather than use `nudge_x` and `nudge_y`, to adjust the position of text, we can use the `vjust` and `hjust` arguments. These arguments adjust the vertical and horizontal justification of the text, respectively. It is notoriously difficult to understand exactly how these work, but we will introduce their basic functionality here.

Understanding `hjust` (horizontal justification)

The `hjust` argument in `ggplot2` adjusts the horizontal position of text labels relative to their anchor points (the actual data points). `hjust` values range from 0 to 1, where:

- `hjust = 0` aligns the text label's left edge with the anchor point.
- `hjust = 0.5` centers the text label on the anchor point.

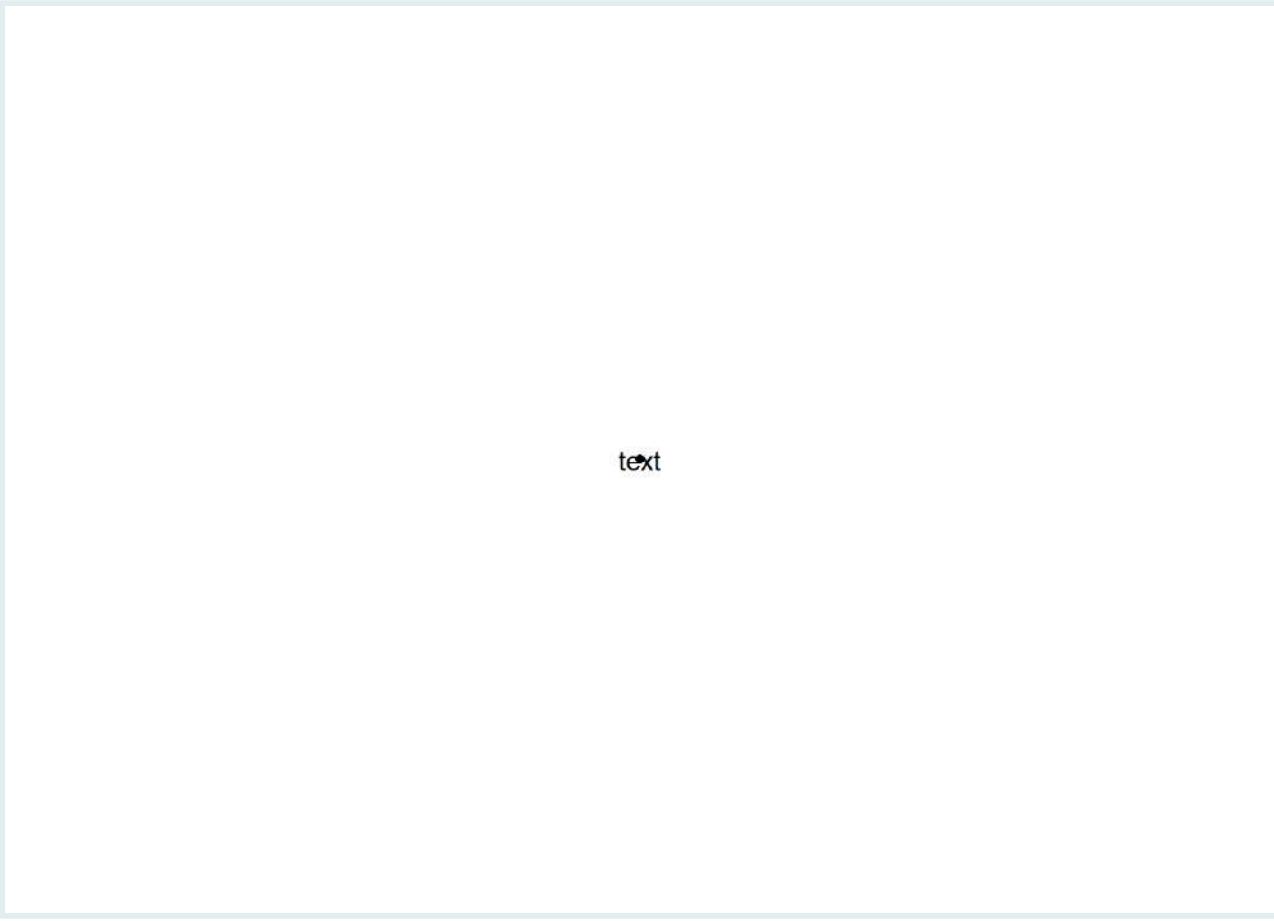
- `hjust = 1` aligns the text label's right edge with the anchor point.

Here's a simple example to illustrate this. First, let's make a plot with a single point and text with no `hjust` argument:

```
# Example data
df <- data.frame(x = 1, y = 1)

# Base plot with a point
base_p <- ggplot(df, aes(x, y)) + geom_point() + theme_void()

base_p + geom_text(aes(label = "text"))
```



text

With no `hjust` argument, the text is centered on the point, which means that the default value of `hjust` is 0.5.

Now let's try setting `hjust` to a variety of values:

```
p_hjust_0 <- base_p + geom_text(aes(label = "hjust=0"), hjust = 0)
p_hjust_0.25 <- base_p + geom_text(aes(label = "hjust=0.25"), hjust = 0.25)
p_hjust_0.5 <- base_p + geom_text(aes(label = "hjust=0.5"), hjust = 0.5)
p_hjust_0.75 <- base_p + geom_text(aes(label = "hjust=0.75"), hjust = 0.75)
p_hjust_1 <- base_p + geom_text(aes(label = "hjust=1"), hjust = 1)

# Combine plots with patchwork
p_hjust_0 / p_hjust_0.25 / p_hjust_0.5 / p_hjust_0.75 / p_hjust_1
```

`hjust=0`

`hjust=0.25`

`hjust=0.5`

`hjust=0.75`

`hjust=1`

As you can see, the text is aligned to the left edge of the point when `hjust = 0`, to the right edge of the point when `hjust = 1`, and moves closer to the center as `hjust` approaches 0.5.

While `hjust` was originally meant to be used between 0 and 1, you can actually use any value for `hjust`, above or below 0 and 1. For example, if you set `hjust = -0.2`, the text will be left-aligned, but with an additional 20% of the text width added to the left of the anchor point, and if you set `hjust = 1.2`, the text will be right-aligned, but with an additional 20% of the text width added to the right of the anchor point:

```
p_hjust_neg0.5 <- base_p + geom_text(aes(label = "hjust=-0.5"), hjust = -0.5)
p_hjust_neg0.2 <- base_p + geom_text(aes(label = "hjust=-0.2"), hjust = -0.2)
p_hjust_1.2 <- base_p + geom_text(aes(label = "hjust=1.2"), hjust = 1.2)
p_hjust_1.5 <- base_p + geom_text(aes(label = "hjust=1.5"), hjust = 1.5)

# Combine plots with patchwork
p_hjust_neg0.5 / p_hjust_neg0.2 / p_hjust_0 / p_hjust_0.25 / p_hjust_0.5 /
  p_hjust_0.75 / p_hjust_1 / p_hjust_1.2 / p_hjust_1.5
```

• `hjust=-0.5`

• `hjust=-0.2`

`hjust=0`

`hjust=0.25`

`hjust=0.5`

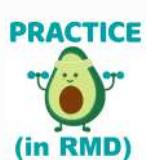
`hjust=0.75`

`hjust=1`

`hjust=1.2` •

`hjust=1.5` •

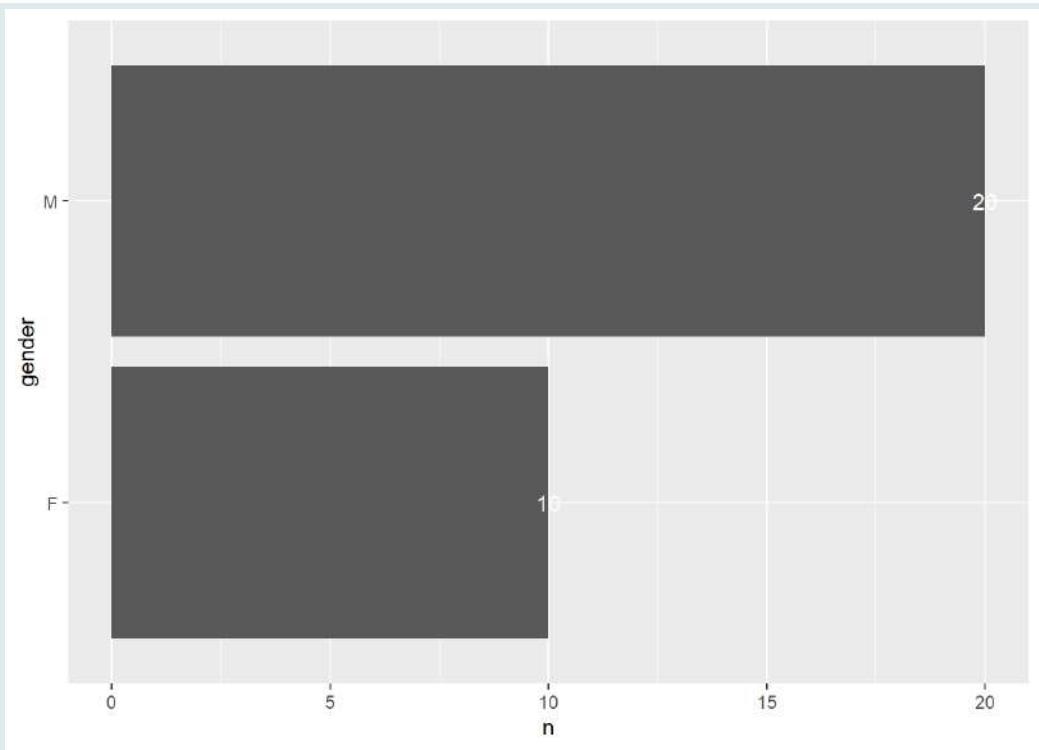
Q: Horizontal adjusment practice



Consider the following horizontal bar plot with text labels added:

```
# sample data
sample_gender <-
  data.frame(gender = c("F", "M"),
             n = c(10, 20))

ggplot(sample_gender, aes(x = n, y = gender)) +
  geom_col() +
  geom_text(aes(label = n), color = "white")
```



Use the `hjust` or `vjust` arguments to adjust the position of the text label so that it is inside the bar, with some padding on the right side.

Using `hjust` and `vjust` values outside the 0-1 range can be problematic when your labels are not the same length. For example, if you have labels of different lengths, setting `hjust = 1.2` will cause the longer labels to extend further to the right than the shorter labels.

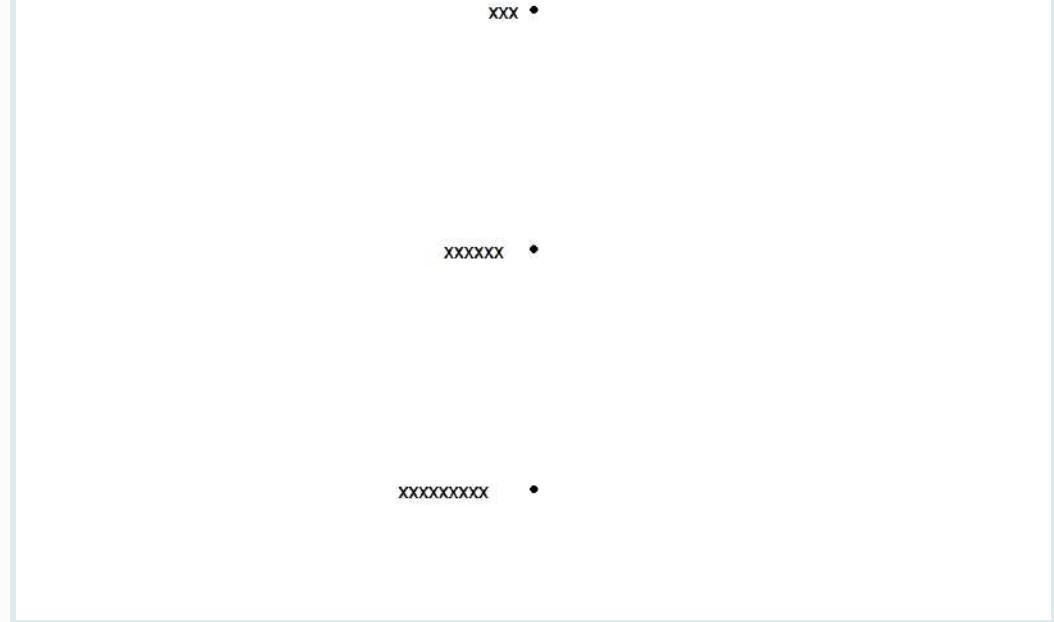
PRO TIP



For example:

```
# Different text labels with varying lengths
p_xx <- base_p + geom_text(aes(label = "xxx"), hjust = 1.5)
p_xxxxx <- base_p + geom_text(aes(label = "xxxxxx"), hjust = 1.5)
p_xxxxxxxxx <- base_p + geom_text(aes(label = "xxxxxxxxx"), hjust =
    1.5)

# Combine plots with patchwork
p_xx / p_xxxxx / p_xxxxxxxxx
```



As you can see, the longer labels have more extra space added to the right of the anchor point than the shorter labels. This is because `hjust` is adding 50% of the text width to the right of the anchor point, so longer labels get more padding.

If this is a problem for you, you can use the `nudge_x` argument to adjust the position of the labels instead. There *are* certain times when using nudges can be problematic though, which is why `hjust` and `vjust` are still useful.

Understanding `vjust` (vertical justification)

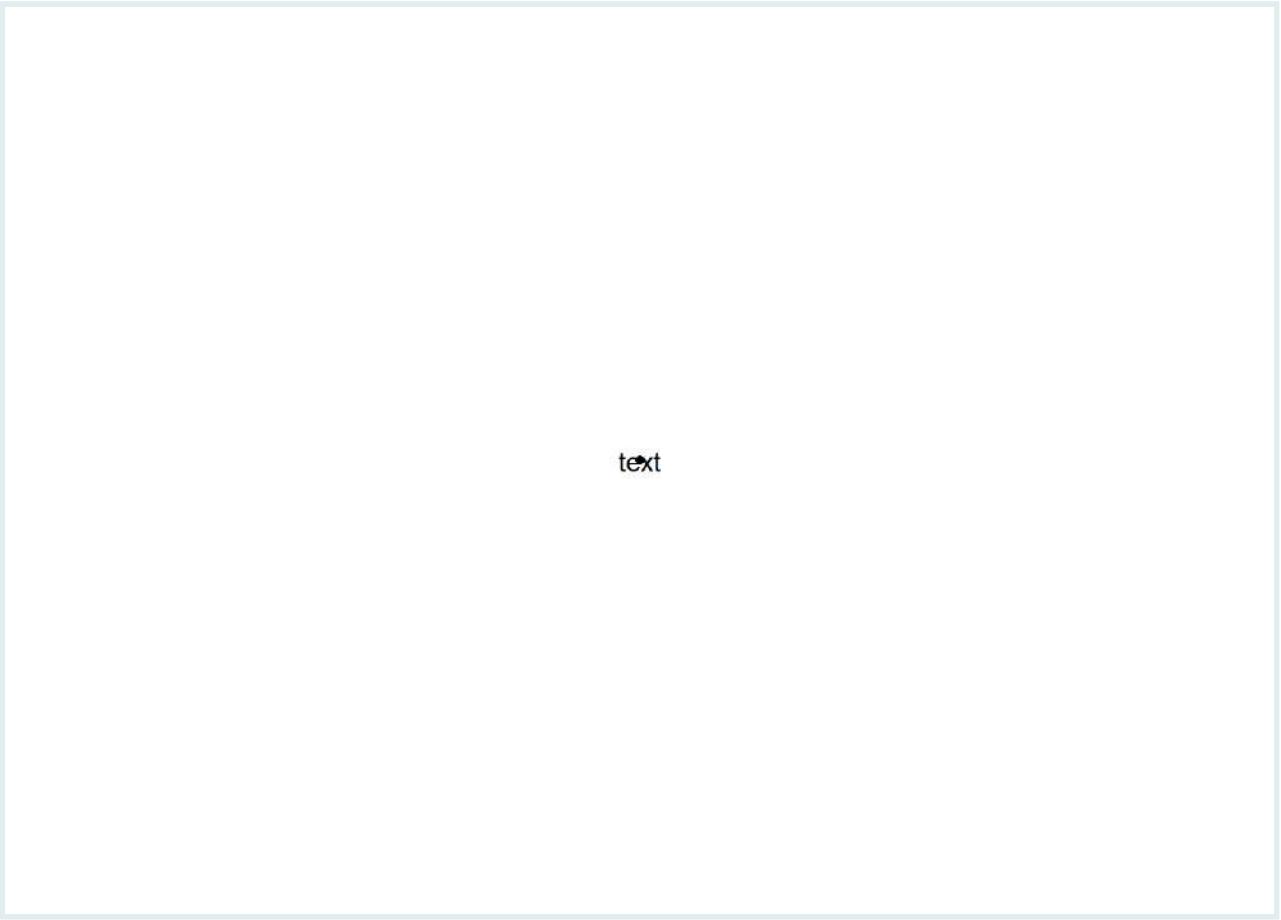
Similarly, the `vjust` argument in `ggplot2` adjusts the vertical position of text labels in relation to their anchor points. `vjust` values also range from 0 to 1, where:

- `vjust = 0` aligns the bottom edge of the text label with the anchor point.
- `vjust = 0.5` centers the text label vertically on the anchor point.
- `vjust = 1` aligns the top edge of the text label with the anchor point.

Here's an example to illustrate `vjust`. We'll start with the same base plot and add text with no `vjust` argument:

```
# Base plot with a point
p <- ggplot(df, aes(x, y)) + geom_point() + theme_void()

p + geom_text(aes(label = "text"))
```



text

By default, with no `vjust` specified, the text is vertically centered on the point, indicating the default value of `vjust` is 0.5.

Now, let's experiment with different `vjust` values:

```
p_vjust_0 <- p + geom_text(aes(label = "vjust=0"), vjust = 0)
p_vjust_0.25 <- p + geom_text(aes(label = "vjust=0.25"), vjust = 0.25)
p_vjust_0.5 <- p + geom_text(aes(label = "vjust=0.5"), vjust = 0.5)
p_vjust_0.75 <- p + geom_text(aes(label = "vjust=0.75"), vjust = 0.75)
p_vjust_1 <- p + geom_text(aes(label = "vjust=1"), vjust = 1)

# Combine plots with patchwork
p_vjust_0 / p_vjust_0.25 / p_vjust_0.5 / p_vjust_0.75 / p_vjust_1
```

vjust=0

vjust=0.25

vjust=0.5

vjust=0.75

vjust=1

Here, `vjust = 0` aligns the text to the bottom of the point, `vjust = 1` aligns it to the top, and as `vjust` approaches 0.5, the text moves closer to the vertical center.

Like `hjust`, `vjust` can also take values outside the 0 to 1 range. For example, `vjust = -0.2` would place the text slightly below the anchor point, and `vjust = 1.2` would place it slightly above. Let's see how these values affect text positioning:

```
p_vjust_neg0.5 <- p + geom_text(aes(label = "vjust=-0.5"), vjust = -0.5)
p_vjust_1.5 <- p + geom_text(aes(label = "vjust=1.5"), vjust = 1.5)

# Combine plots with patchwork
p_vjust_neg0.5 / p_vjust_0 / p_vjust_0.25 / p_vjust_0.5 / p_vjust_0.75 /
    p_vjust_1 / p_vjust_1.5
```

vjust=0.5

vjust=0

vjust=0.25

vjust=0.5

vjust=0.75

vjust=1

vjust=1.5

As with `hjust`, using `vjust` values beyond the typical 0 to 1 range can be useful for fine-tuning the placement of your text labels, allowing them to extend slightly above or below the anchor point.

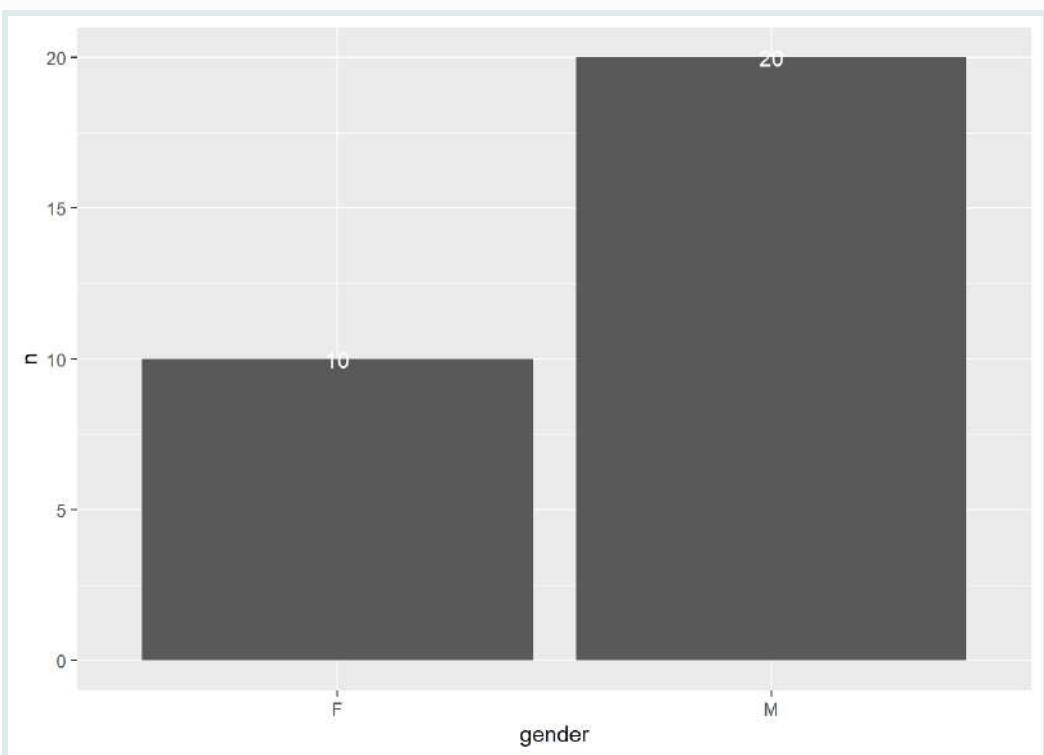
Q: Vertical adjustment practice

Consider the following bar plot with text labels added:



```
# sample data
sample_gender <-
  data.frame(gender = c("F", "M"),
             n = c(10, 20))

ggplot(sample_gender,
       aes(x = gender, y = n)) +
  geom_col() +
  geom_text(aes(label = n), color = "white")
```



Use the `hjust` or `vjust` arguments to adjust the position of the text label so that it is inside the bar, with some padding on the top.

Data Example: TB treatment outcomes in Benin

Let's apply what we've learned to a real dataset.

The `tb_outcomes` dataset, which we used in the previous lesson, will serve as the foundation for our examples.

```
tb_outcomes <- read_csv(here::here('data/benin_tb.csv'))  
tb_outcomes
```

We'll be trying to plot the number of cases per hospital.

Unlike with our initial practice dataset, we do not already have the total number of cases per hospital; this information is stored in the `cases` column, but we need to summarize it first.

Let's calculate the total number of cases per hospital using the `group_by()` and `summarize()` function:

```

hospital_sums <-
  tb_outcomes %>%
  group_by(hospital) %>%
  summarize(cases = sum(cases))

hospital_sums

```

```

## # A tibble: 6 × 2
##   hospital      cases
##   <chr>        <dbl>
## 1 CHPP Akron    875
## 2 CS Abomey-Calavi 791
## 3 Hopital Bethesda 256
## 4 Hopital Savalou    80
## 5 Hopital St Luc    168
## 6 St Jean De Dieu   171

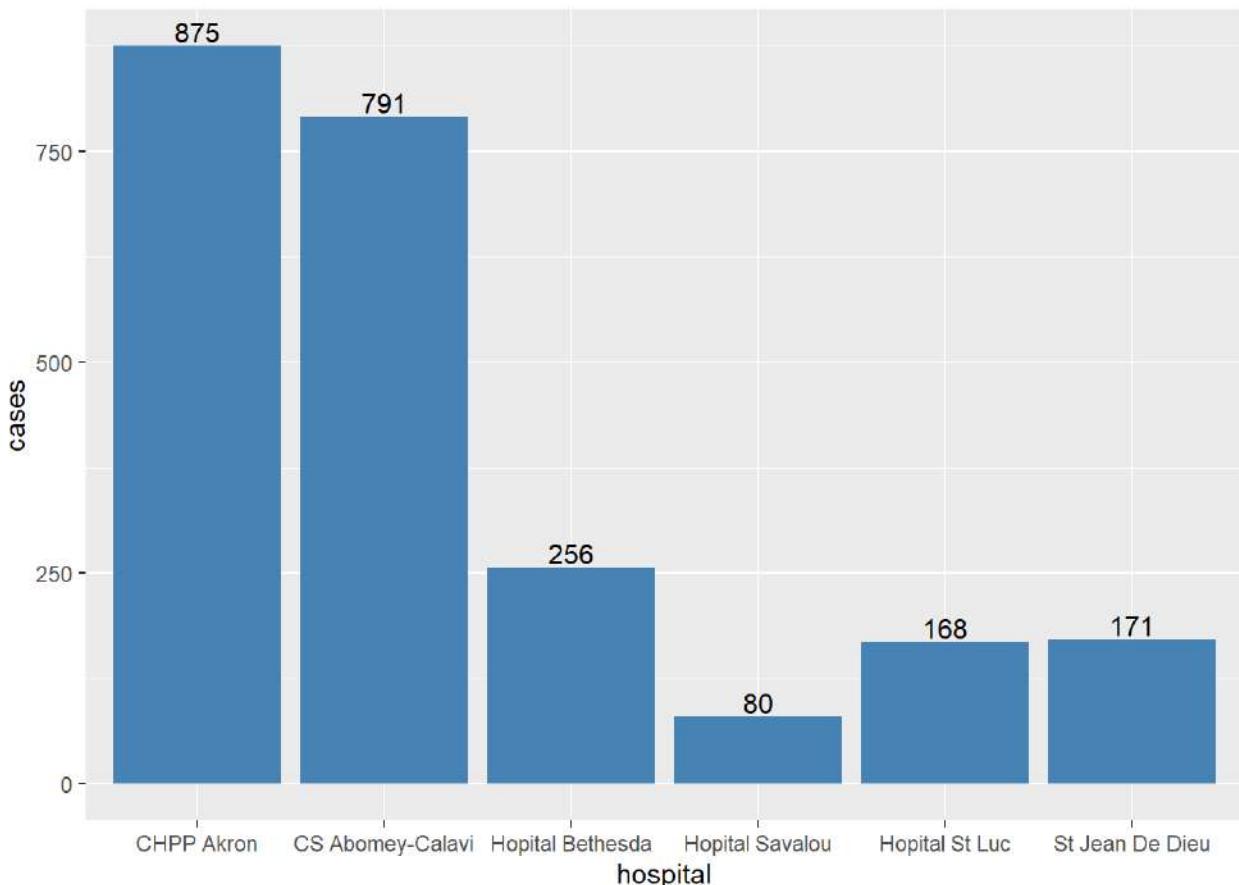
```

Now let's use `hospital_sums` to visualize each hospital's total number of cases and use `geom_text()` to annotate the bars:

```

ggplot(hospital_sums, aes(x = hospital, y = cases)) +
  geom_col(fill = "steelblue") +
  geom_text(aes(label = cases),
            vjust = -0.2)

```



Great, now you see how to use the `summarize()` function to calculate group totals, and how to use `geom_text()` to annotate your plots.

Q: Summarize then plot

Consider the `aus_tb_notifs` dataset imported below, which shows the number of TB cases in urban and rural areas per quarter:

```
aus_tb_notifs <-  
  read_csv(here::here('data/aus_tb_notifs_modified.csv'))  
aus_tb_notifs
```

```
## # A tibble: 52 × 4  
##   year quarter rural urban  
##   <dbl> <chr>    <dbl> <dbl>  
## 1 2010 Q1        4     87  
## 2 2010 Q2        4     98  
## 3 2010 Q3        5    101  
## 4 2010 Q4       10    124  
## 5 2011 Q1        5     81  
## 6 2011 Q2        4     52  
## 7 2011 Q3        9    102  
## 8 2011 Q4        5    100  
## 9 2012 Q1        9     80  
## 10 2012 Q2       4     63  
## # ... i 42 more rows
```



Create a simple bar plot to visualize the total number of TB cases in urban areas for **each year**. Label each bar with the total number of cases using `geom_text()` just below the bar.

Hint: First, aggregate the data by year and sum up the urban cases. Then use `ggplot()` with `geom_col()` for the bar plot and `geom_text()` for the labels.



Further Aesthetic modifications

So far we have only used some of the possible aesthetics for `geom_text()`. The minimum three aesthetics are `x`, `y`, and `label`. These must be mapped to a variable defined inside `aes()`.

Additional aesthetics include:

- `size`: the size of the text, in mm
- `angle`: the angle of the text, from 0 to 360
- `alpha`: the transparency of the text, from 0 to 1
- `color`: the color of the text
- `family`: the font family of the text, such as “sans”, “serif”, “mono”
- `fontface`: the font face of the text, including “plain”, “bold”, “italic”, “bold.italic”
- `group`: a grouping variable for the text
- `hjust`: horizontal justification of the text
- `vjust`: vertical justification of the text
- `lineheight`: the line height of the text

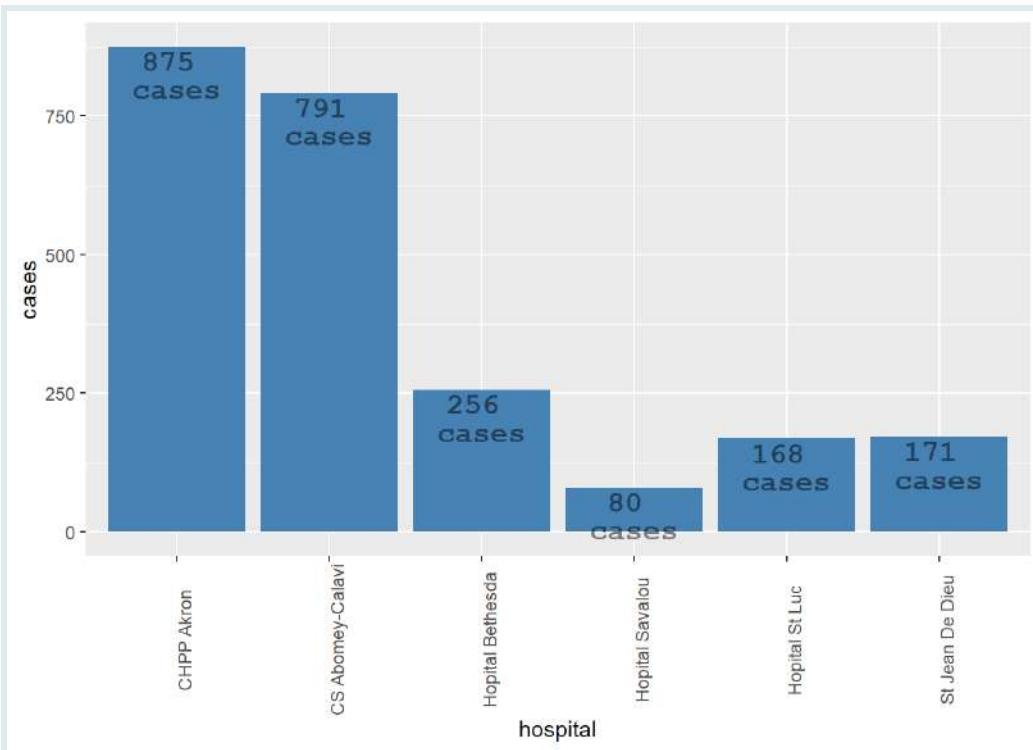


`nudge_y` and `nudge_x` are also available, but are not formally considered aesthetics, as they cannot be mapped to a variable inside `aes()`, and must be set outside of it.

Here is an example plot with most of these aesthetics set. It's not a very beautiful plot. Try modifying the code to see how each aesthetic changes the plot:

```
ggplot(hospital_sums, aes(x = hospital, y = cases)) +  
  geom_col(fill = "steelblue") +  
  geom_text(aes(label = paste(cases, "\ncases")),  
            size = 5,  
            angle = 0,  
            alpha = 0.5,  
            color = "black",  
            family = "mono",  
            fontface = "bold",  
            hjust = 0.5,  
            vjust = 1,  
            nudge_y = -10,  
            lineheight = 0.8) +  
  theme(axis.text.x = element_text(angle = 90))
```

PRO TIP



Labeling stacked bar plots

So far, we've only looked at bar plots with a single categorical variable. Let's build plots with two categorical variables and add labels to each subgroup. We'll start with stacked bar plots.

We summarize the `tb_outcomes` dataset by `period_date` and `diagnosis_type`, calculating the sum of cases (`cases`) for each group.

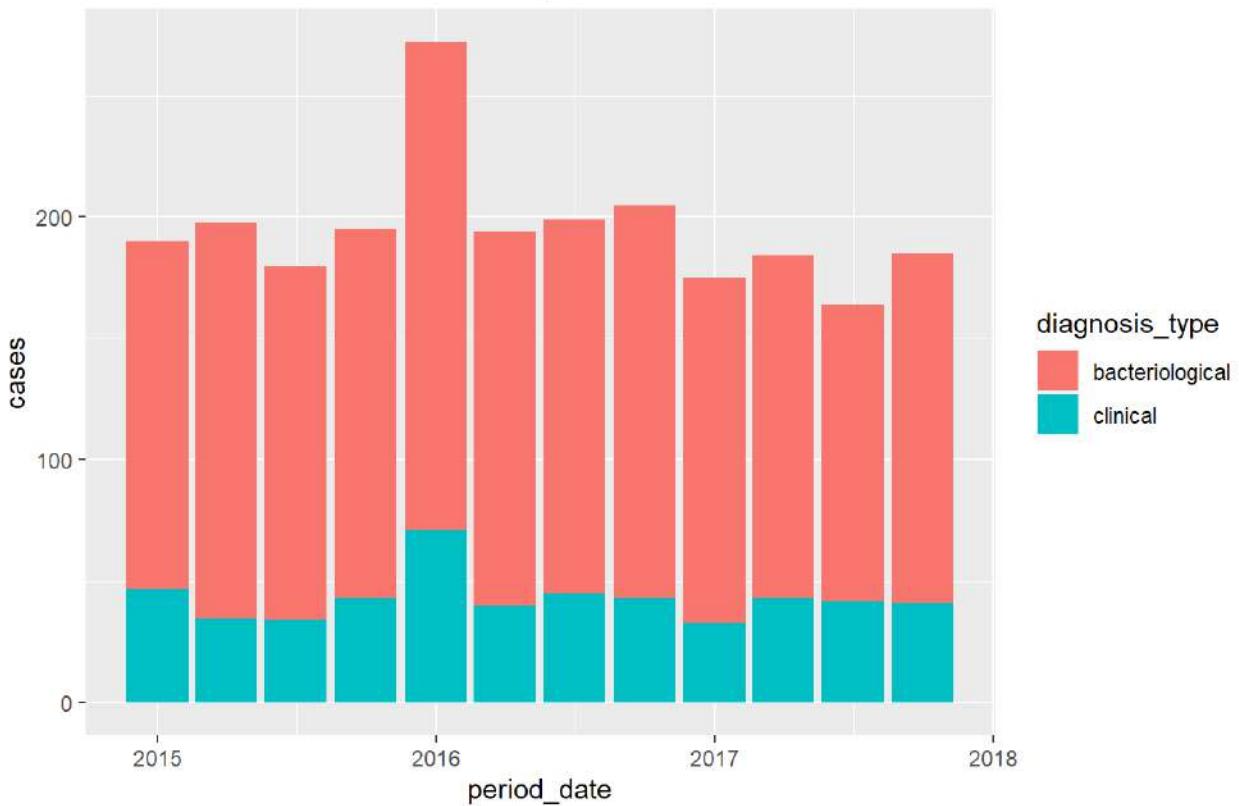
```
# Summarize the data by period and diagnosis type
tb_sum <- tb_outcomes %>%
  group_by(period_date, diagnosis_type) %>%
  summarise(cases = sum(cases))

tb_sum
```

Now, let's create a **simple stacked bar plot** and see how to add labels to it:

```
# Create a basic bar plot using the summarized data
quarter_dx_bar <- tb_sum %>%
  ggplot(aes(x = period_date, y = cases, fill = diagnosis_type)) +
  geom_col() +
  labs(title = "New and relapse TB cases per quarter",
       subtitle = "Data from six health facilities in Benin, 2015-2017")
quarter_dx_bar
```

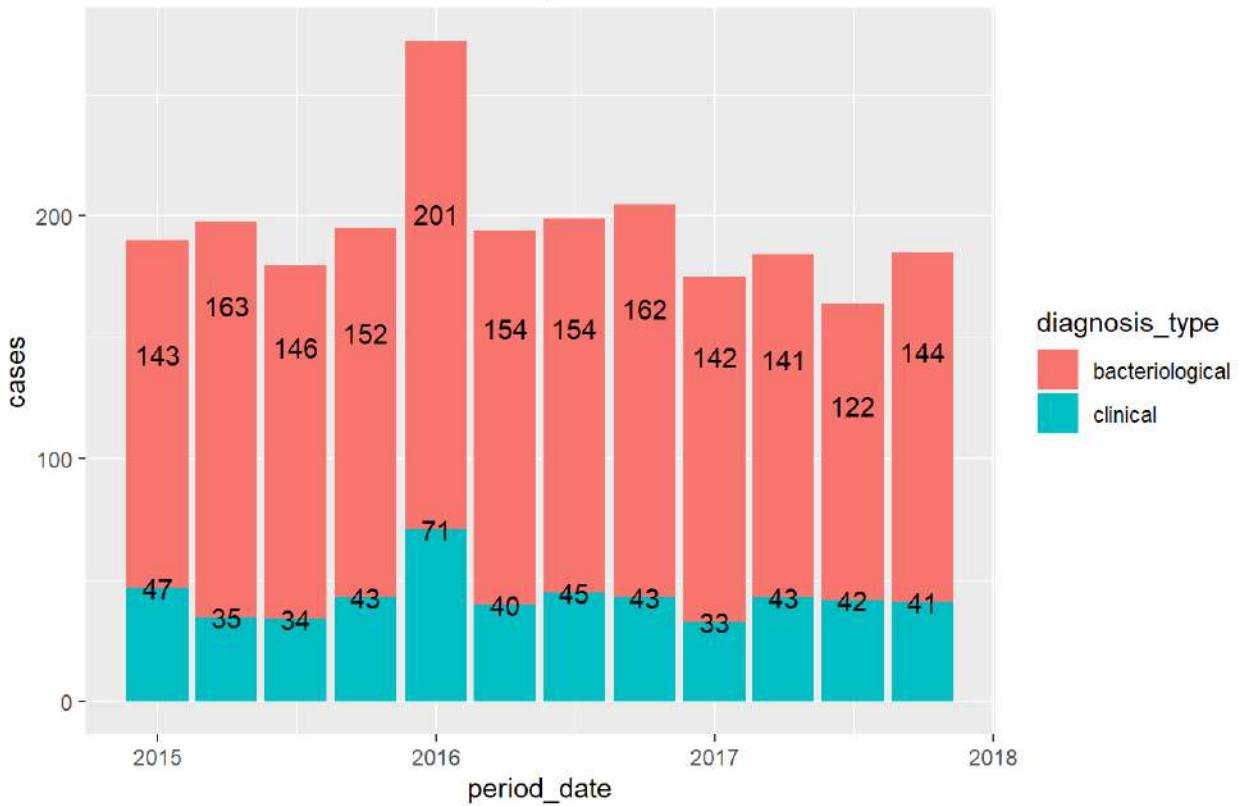
New and relapse TB cases per quarter
Data from six health facilities in Benin, 2015-2017



We'll use the `cases` column for labeling each bar:

```
# Add text labels to the bar plot
quarter_dx_bar +
  geom_text(aes(label = cases))
```

New and relapse TB cases per quarter
Data from six health facilities in Benin, 2015-2017

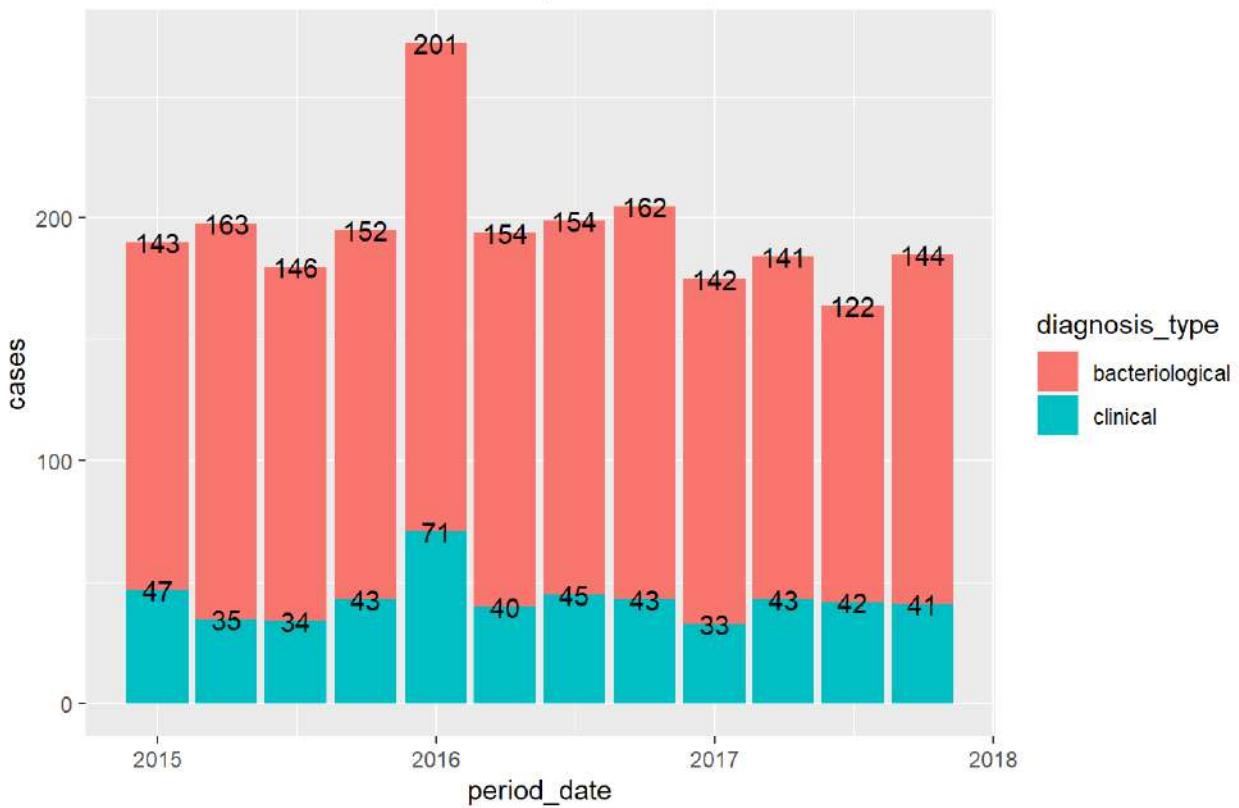


Oops, the labels are not in the right place! They don't align with the height of the bars in our plot.

The issue is that `geom_text()` does not stack positions by default like `geom_col()`. We must explicitly set `position = "stack"` in `geom_text()`:

```
# Place text at the top of each bar segment
quarter_dx_bar +
  geom_text(aes(label = cases),
            position = "stack") # Set position to stack
```

New and relapse TB cases per quarter
Data from six health facilities in Benin, 2015-2017

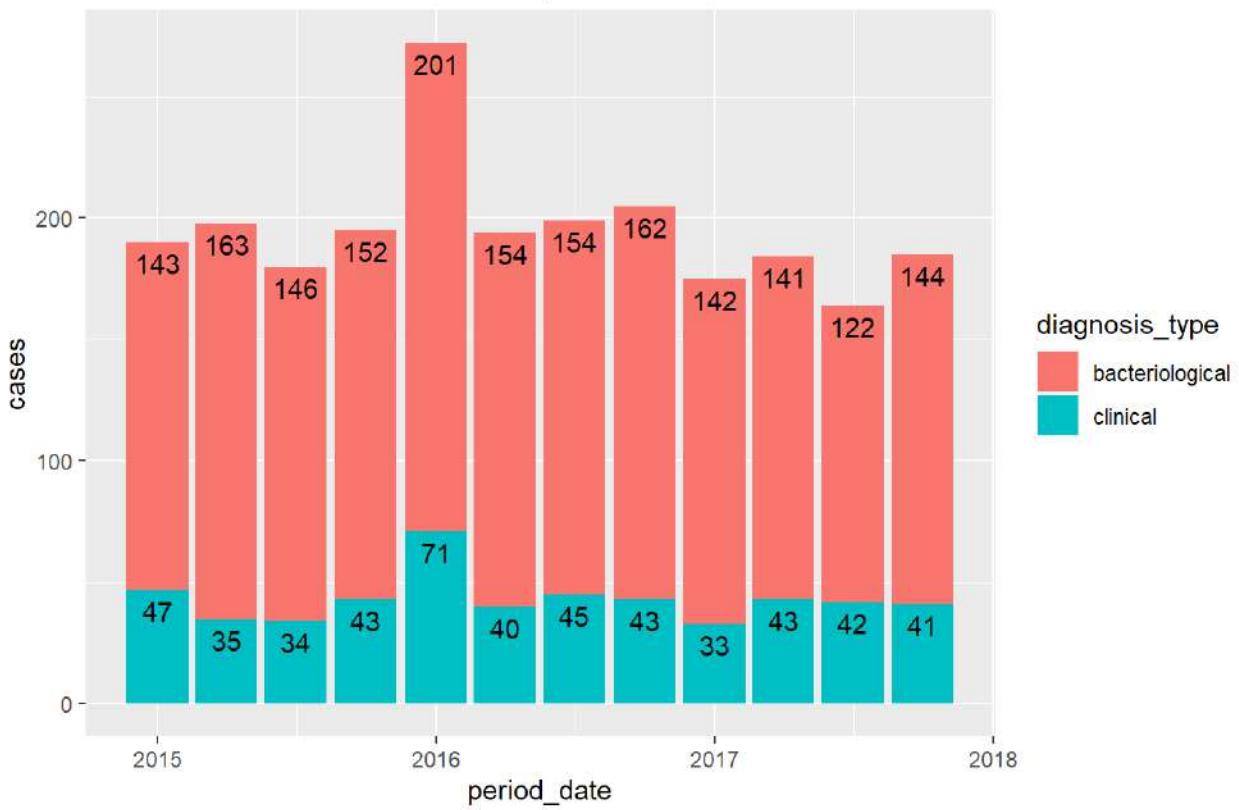


Great!

To vertically align the text inside the bars, we can add `vjust` to `geom_text()`:

```
# Reposition labels inside the stacks for clarity and change the font style
quarter_dx_bar +
  geom_text(aes(label = cases),
            position = "stack",
            vjust = 1.5)
```

New and relapse TB cases per quarter
Data from six health facilities in Benin, 2015-2017

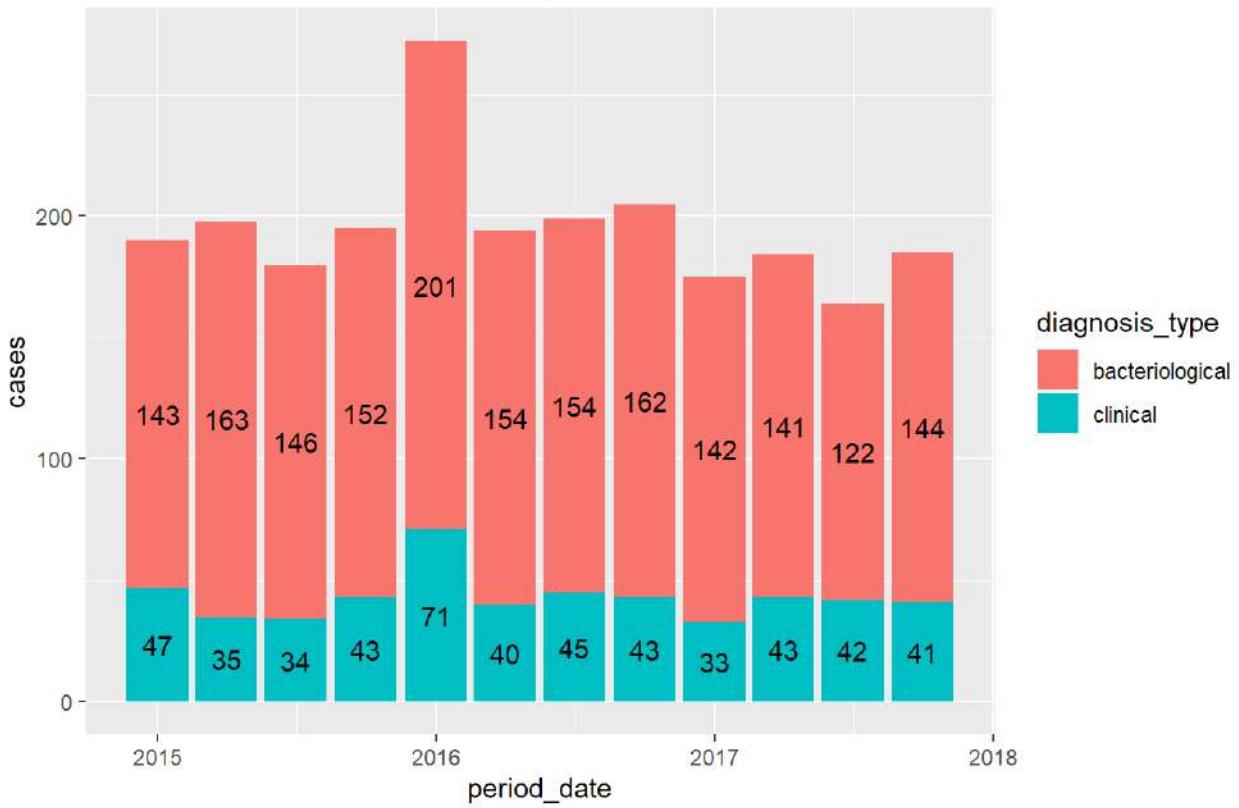


This works well, the labels are now inside the bars, and setting `vjust = 1.5` adds an extra 50% of label height as padding between the label and the top of the bar.

But what if we want to center the labels vertically within each bar segment? To do this, we switch from `position = "stack"` to the more customizable `position_stack()` function, and set `vjust = 0.5` *within* `position_stack()`:

```
quarter_dx_bar +
  geom_text(aes(label = cases),
            position = position_stack(vjust = 0.5))
```

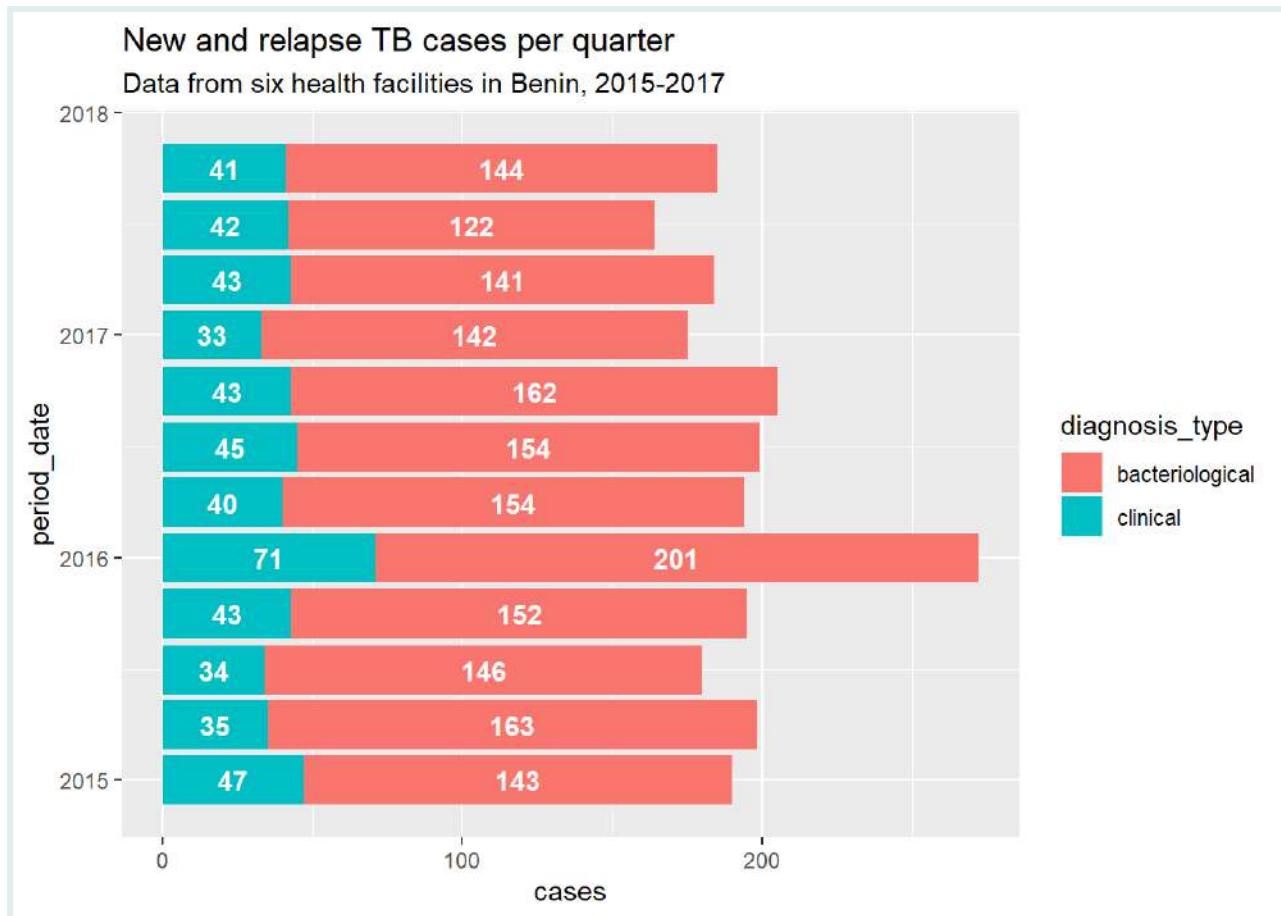
New and relapse TB cases per quarter
Data from six health facilities in Benin, 2015-2017



Now the labels are vertically centered within each bar segment.

This label placement is especially nice for horizontal bar plots. Below we flip the axes of our plot using `coord_flip()` to create a horizontal bar plot, and add some extra aesthetic modifications to make the plot more readable:

```
quarter_dx_bar +
  geom_text(aes(label = cases),
            position = position_stack(vjust = 0.5),
            # some extra adjustments
            color = "white",
            fontface = "bold") +
  coord_flip()
```



That looks great! Let's move on to dodged bar charts now.

Q: Practice with labeling stacked plots

Create a stacked bar plot showing the distribution per year of TB cases in rural and urban areas using the `aus_tb_notifs` dataset. Use `geom_text()` and adjust the position of the labels for clarity.



Hint: Pivot the data so that `area_type` is a column, then summarize the data by `year` and `area_type`, calculating the sum of cases (`cases`) for each group. The pivoting is done for you in the code below.

```
# Pivot the data
aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases")
```



```
## # A tibble: 104 × 4
##   year quarter area_type cases
##   <dbl> <chr>   <chr>     <dbl>
## 1 2010 Q1    rural      4
## 2 2010 Q1    urban     87
## 3 2010 Q2    rural      4
## 4 2010 Q2    urban     98
## 5 2010 Q3    rural      5
## 6 2010 Q3    urban    101
## 7 2010 Q4    rural     10
## 8 2010 Q4    urban    124
## 9 2011 Q1    rural      5
## 10 2011 Q1   urban     81
## # ... i 94 more rows
```

```
# Summarize the data by year and area type
```

```
# Create the stacked bar plot
```

Labeling dodged bar plots

Dodged bar charts display multiple categories side by side. Let's explore how to group the data and properly position labels for clear interpretation.

To begin, we'll group our dataset `tb_outcomes` by `hospital` and `diagnosis_type`, calculating the sum of cases (`cases`) for each group.

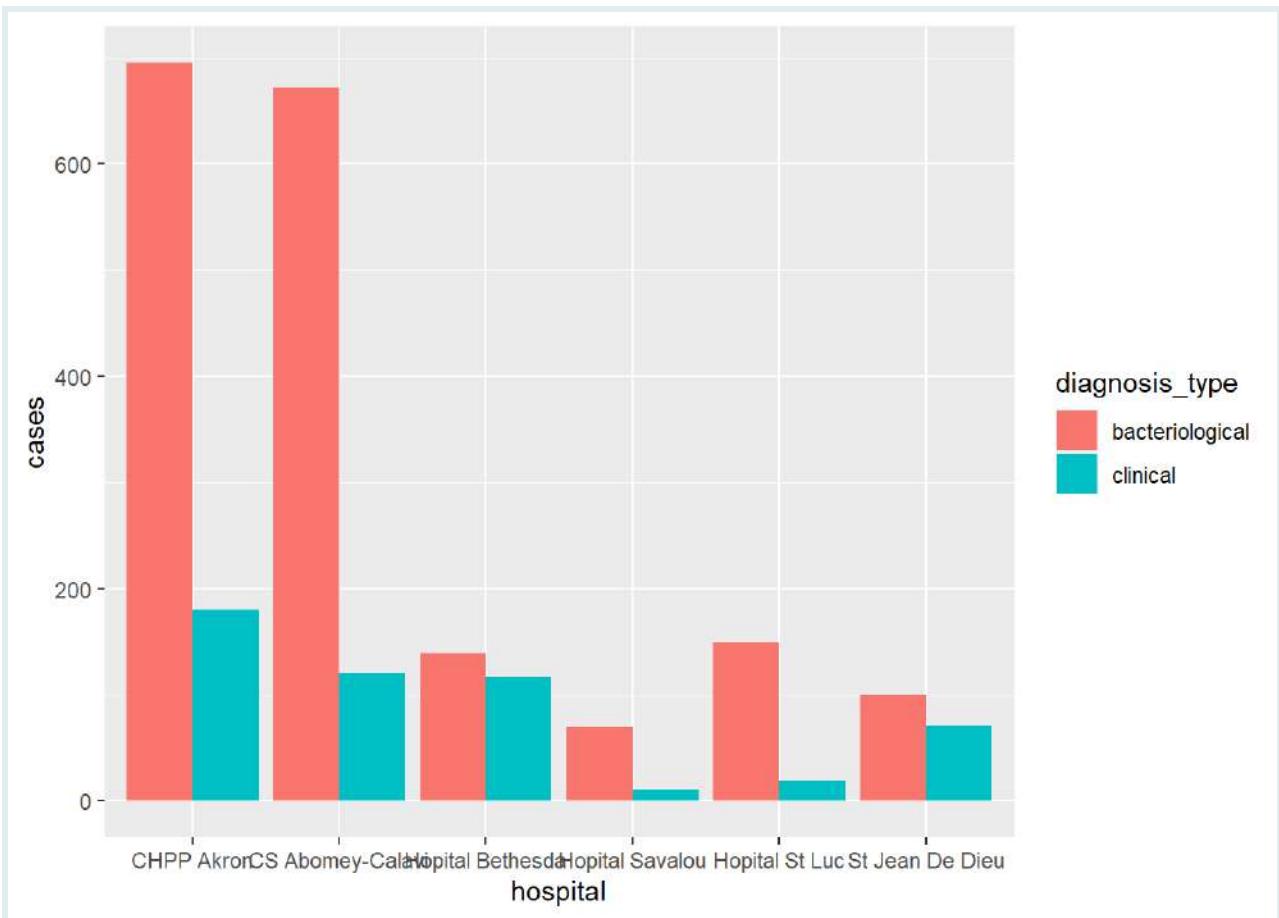
```
hospital_dx_cases <- tb_outcomes %>%
  group_by(hospital, diagnosis_type) %>%
  summarise(cases = sum(cases))

hospital_dx_cases
```

Next, let's create a simple **dodged bar chart**, where the height of each bar signifies the total number of cases for a specific diagnosis in each hospital. Since the default parameter for `geom_col` is `stack`, we must explicitly set `position = "dodge"` to create a dodged bar chart.

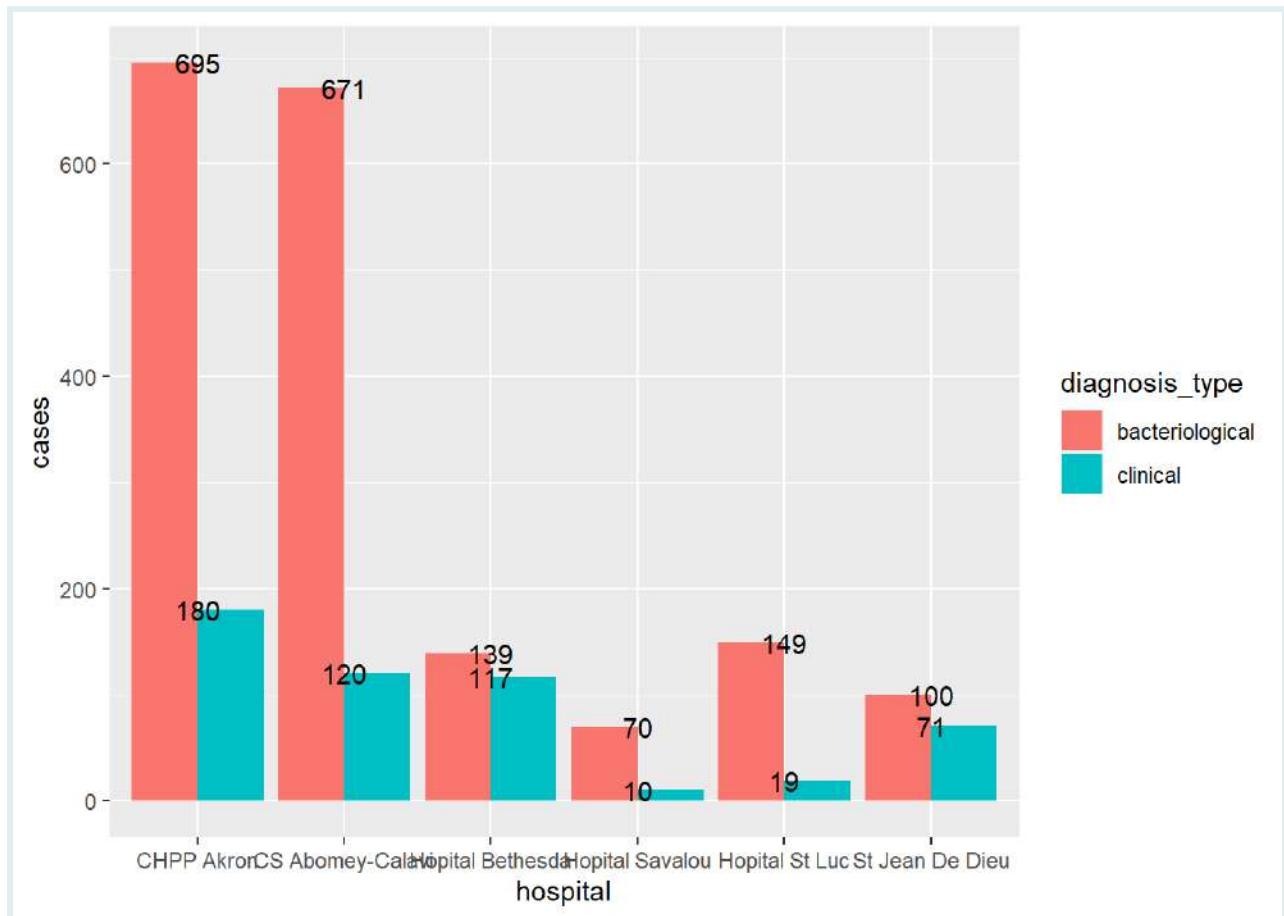
```
hospital_dx_bar <- hospital_dx_cases %>%
  ggplot(aes(x = hospital, y = cases, fill = diagnosis_type)) +
  geom_col(position = "dodge")

hospital_dx_bar
```



Now, we can annotate the chart with `geom_text()` to display the labels, just as we've done before.

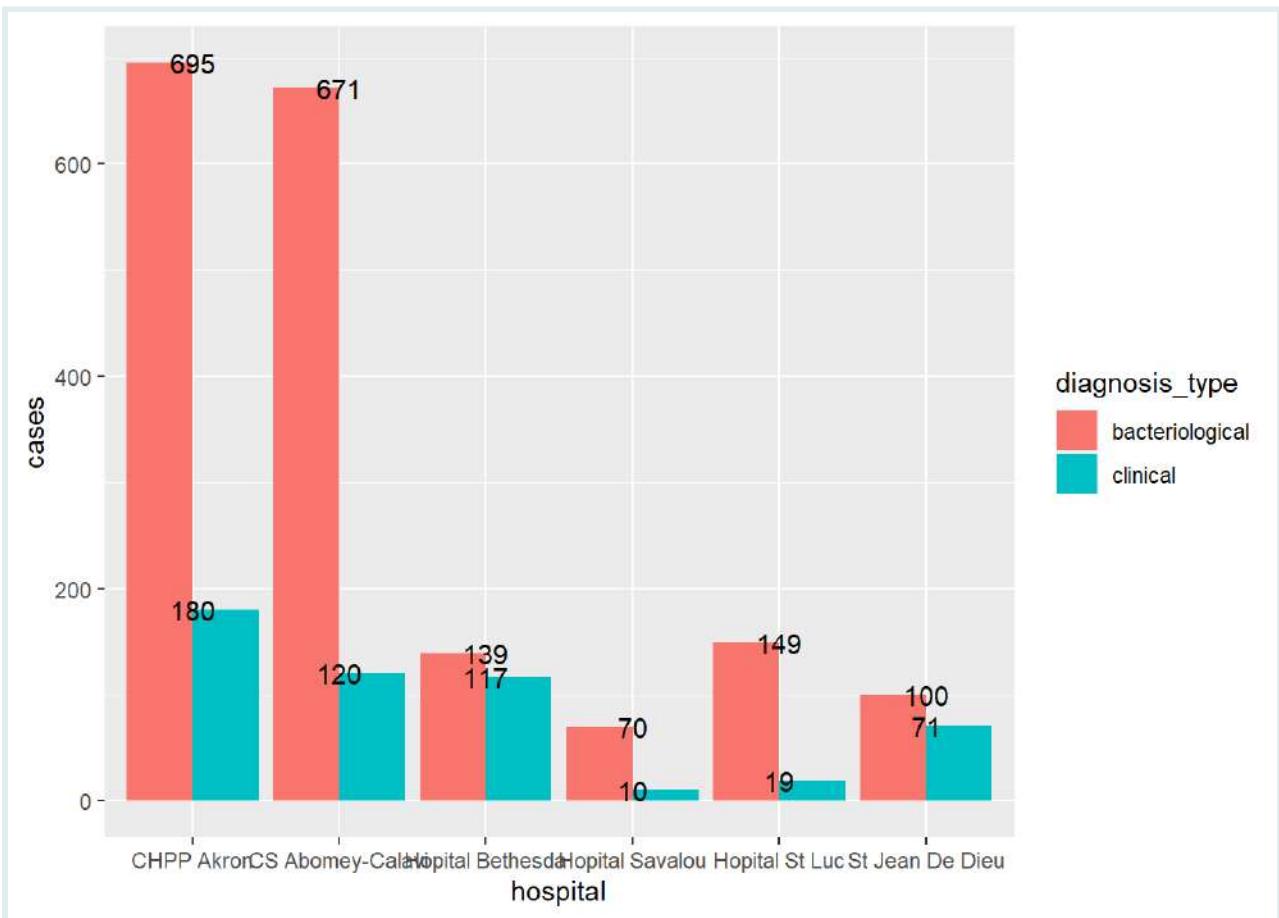
```
hospital_dx_bar +  
  geom_text(aes(label = cases))
```



Oops, that's not quite right! The labels are vertically centered in a straight line, and they're not aligned with the bars. Let's take a look at how we can fix that.

Just as with our stacked bar chart in the previous section, we need to add the position adjustment to `geom_text()`. This time we're going to specify `position = position_dodge()`.

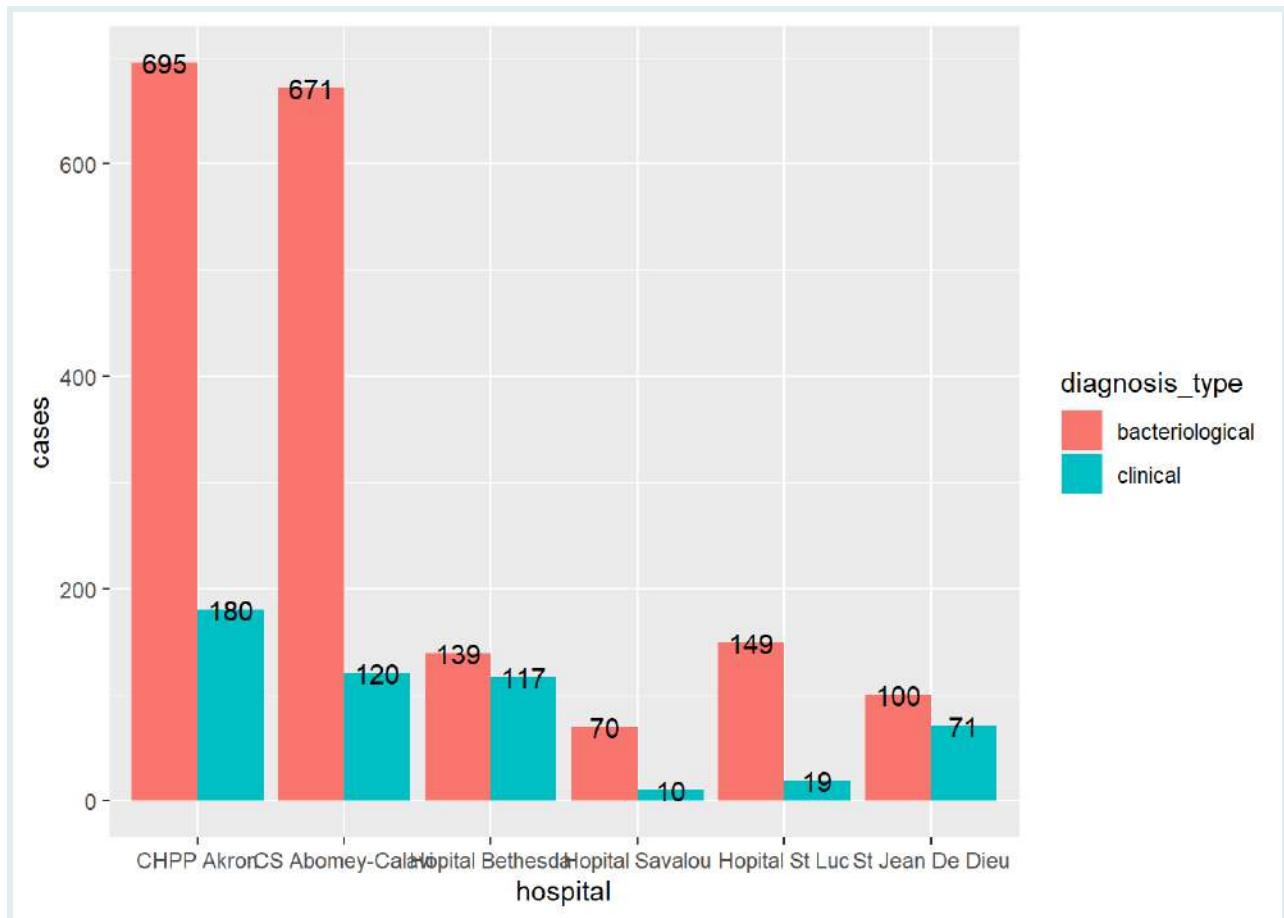
```
hospital_dx_bar +
  geom_text(aes(label = cases),
            position = position_dodge())
```



Oh no. We get the same chart as before. This is because a width argument is required for `position_dodge()`.

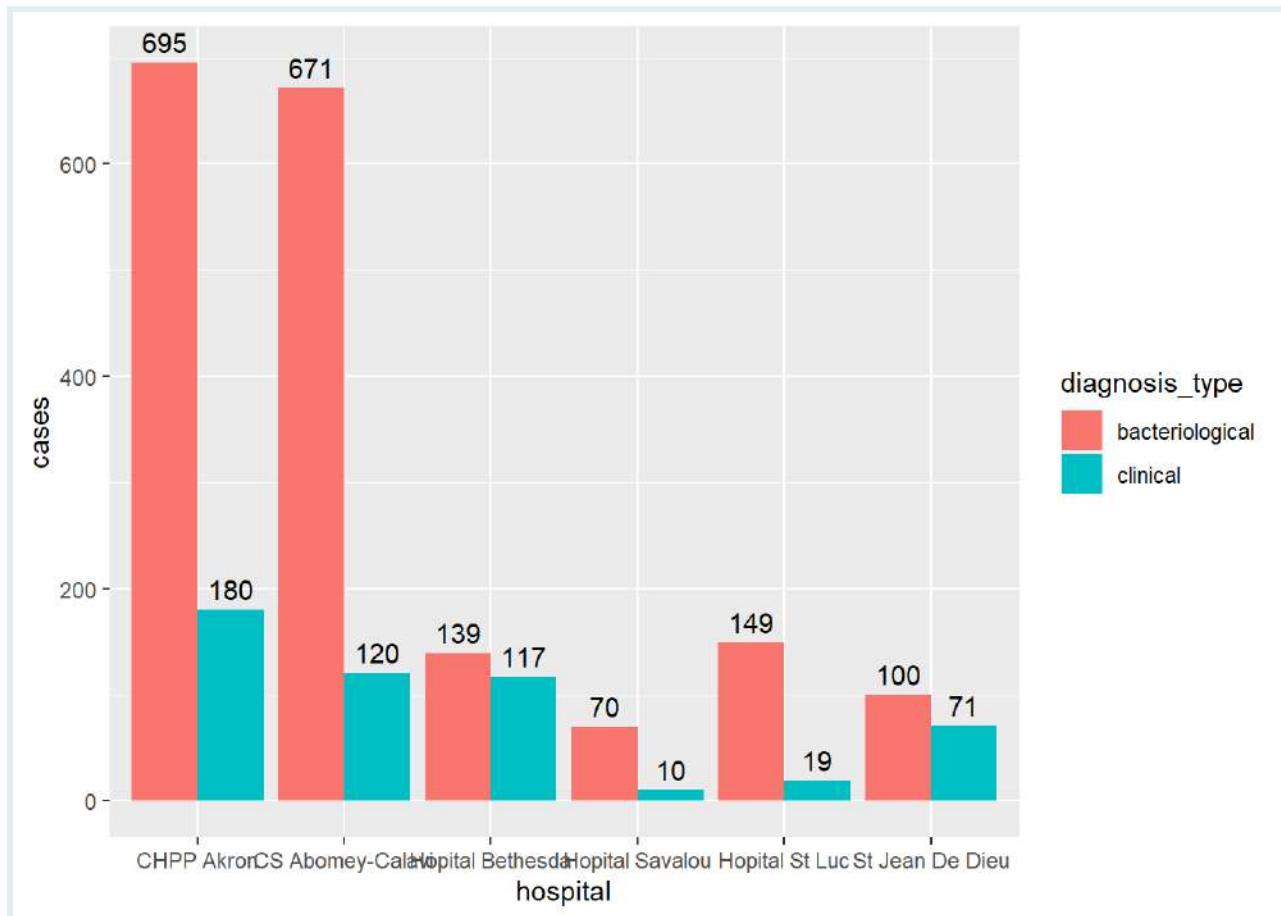
For `geom_col()`, the default value of `width` is 0.9. We'll also use 0.9 for `geom_text()` to ensure the bars and labels are aligned:

```
hospital_dx_bar +
  geom_text(aes(label = cases),
            position = position_dodge(width = 0.9))
```



Now all that's left to do is shift the labels up a bit with vjust.

```
hospital_dx_bar +
  geom_text(aes(label = cases),
            position = position_dodge(width = 0.9),
            vjust = -0.5)
```



That looks great! Next, we'll move on to percent-stacked bar plots.

Q: Practice with labeling dodged bar plots

Generate a dodged bar plot that displays rural and urban TB cases side by side for each year using the `aus_tb_notifs` dataset. Label each bar using `geom_text()`, ensuring the labels are correctly aligned.



You can use the code and comments below as a guide:

```
# Pivot the data
aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases")
```

```
## # A tibble: 104 × 4
##       year quarter area_type cases
```



```
## <dbl> <chr> <chr> <dbl>
## 1 2010 Q1    rural     4
## 2 2010 Q1    urban    87
## 3 2010 Q2    rural     4
## 4 2010 Q2    urban    98
## 5 2010 Q3    rural     5
## 6 2010 Q3    urban   101
## 7 2010 Q4    rural    10
## 8 2010 Q4    urban   124
## 9 2011 Q1    rural     5
## 10 2011 Q1   urban   81
## # i 94 more rows
```

```
# then summarize the data by year and area type

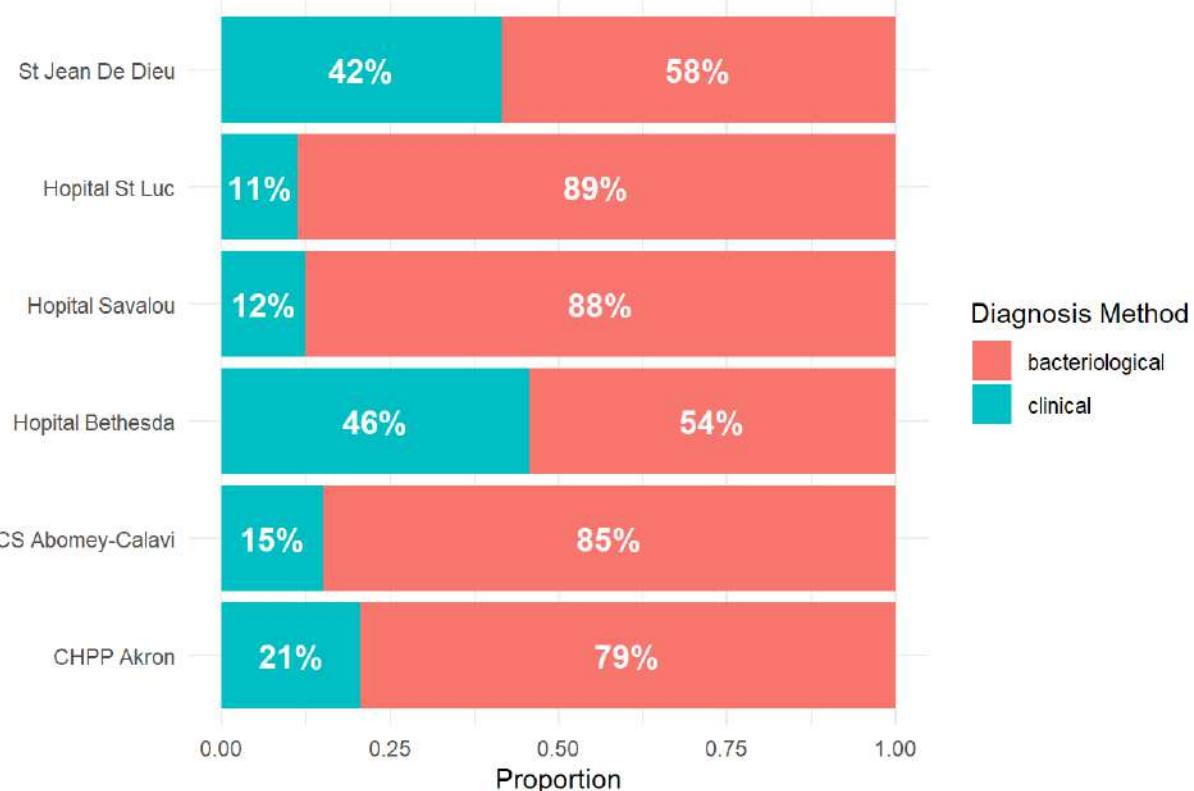
# then create the dodged bar plot
# for the text, use position = position_dodge(width = 0.9)
```

Labeling percent-stacked bar plots

When labeling percent-stacked bar plots, the labels should reflect the percentages of each category. This means we need to format the labels into percentages to ensure they match the segments on the chart. By the end of this section, you'll know how to create a graph like the one below!

New and Relapse Tuberculosis Cases Diagnosis

Data from six health facilities in Benin, 2015-2017



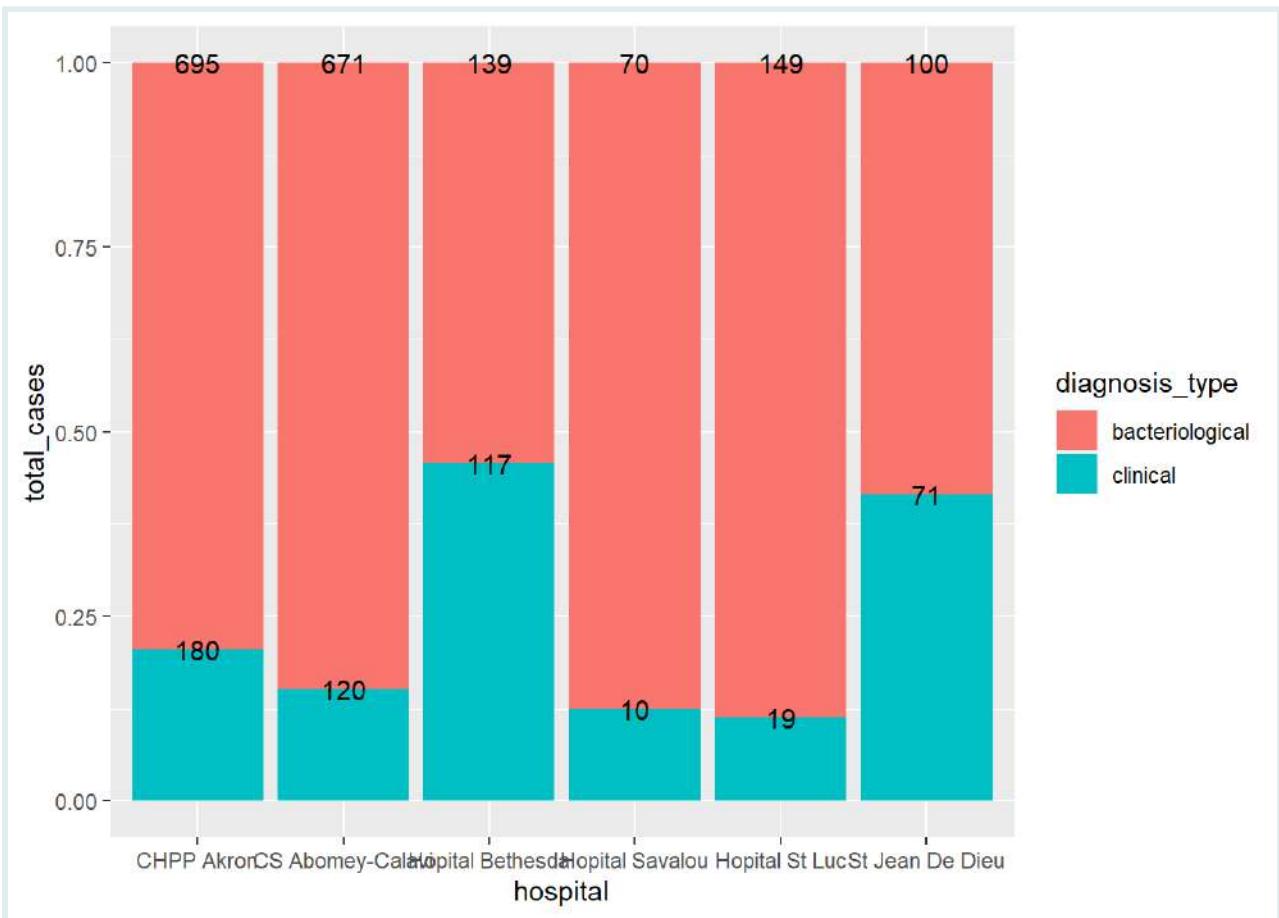
To get started, let's calculate the total number of cases for each health facility (hospital) by diagnostic type.

```
hosp_dx_sum <- tb_outcomes %>%
  group_by(hospital, diagnosis_type) %>%
  summarise(total_cases = sum(cases))

hosp_dx_sum
```

We could use this dataset to create a percent-stacked bar plot. You may remember from the last lesson that for percent stacked plots, we need to use the `fill` position. By now, you should recognize that we want to use the more customizable `position_fill()` instead of the simpler `position = "fill"`. Let's apply this position to both the bars and the labels.

```
hosp_dx_sum %>%
  ggplot(aes(x = hospital, y = total_cases, fill = diagnosis_type)) +
  geom_col(position = position_fill()) +
  geom_text(aes(label = total_cases),
            position = position_fill())
```



This is a good start but it need some improvements. For starters, we want percentages, not raw values.

In order to prepare our data for this, we need to calculate the proportion of cases for each hospital and diagnosis type before we create the plot:

```
hosp_dx_prop <- tb_outcomes %>%
  group_by(hospital, diagnosis_type) %>%
  summarise(total_cases = sum(cases)) %>%
  mutate(prop = total_cases / sum(total_cases))
```

```
hosp_dx_prop
```

```
## # A tibble: 12 × 4
## # Groups:   hospital [6]
##   hospital      diagnosis_type  total_cases     prop
##   <chr>        <chr>            <dbl>      <dbl>
## 1 CHPP Akron  bacteriological  695  0.794
## 2 CHPP Akron  clinical         180  0.206
## 3 CS Abomey-Calavi bacteriological 671  0.848
## 4 CS Abomey-Calavi clinical        120  0.152
## 5 Hopital Bethesda bacteriological 139  0.543
## 6 Hopital Bethesda clinical         117  0.457
## 7 Hopital Savalou  bacteriological 70   0.875
```

```

##  8 Hopital Savalou  clinical          10  0.125
##  9 Hopital St Luc  bacteriological 149  0.887
## 10 Hopital St Luc  clinical         19  0.113
## 11 St Jean De Dieu bacteriological 100  0.585
## 12 St Jean De Dieu clinical        71  0.415

```

Now we have a proportion column, `prop`, that we can use to create our percent-stacked bar plot.

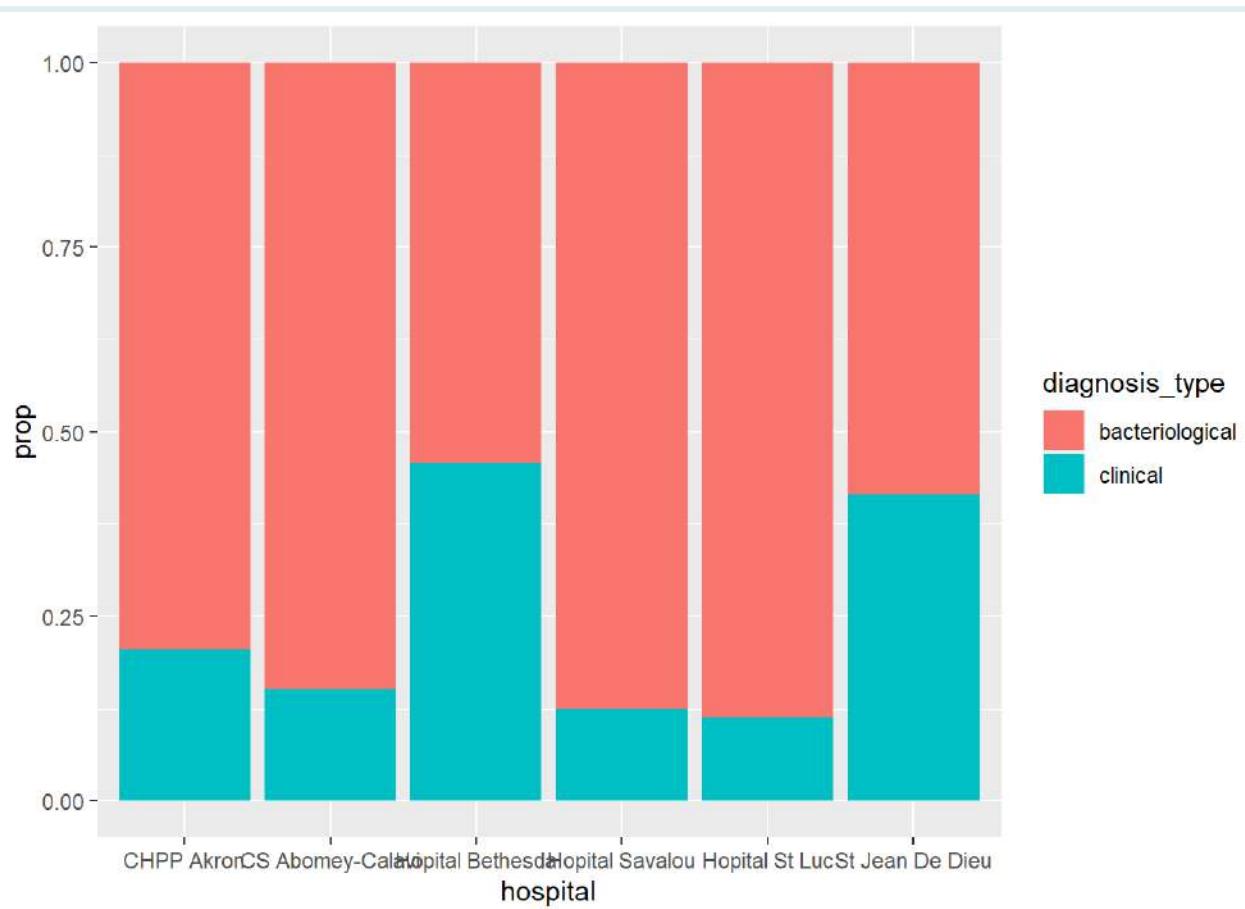
Let's create a bar chart using our new dataset `hosp_dx_prop` with `prop` as our new y variable:

```

hosp_dx_fill <- hosp_dx_prop %>%
  ggplot(aes(x = hospital, y = prop, fill = diagnosis_type)) +
  geom_col(position = position_fill())

```

`hosp_dx_fill`

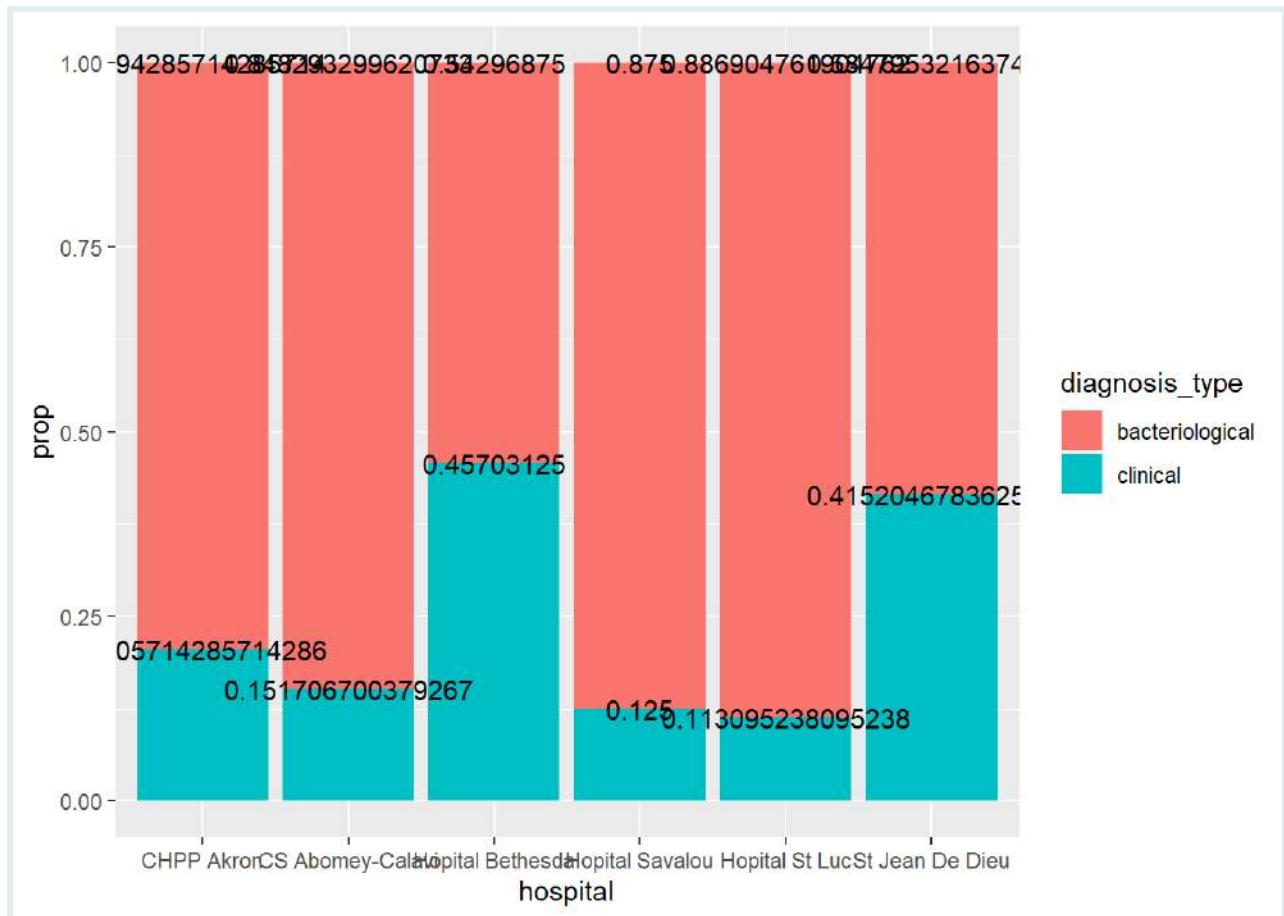


Now, we can use `geom_text()` and specify the position to the labels:

```

hosp_dx_fill +
  geom_text(aes(label = prop),
            position=position_fill())

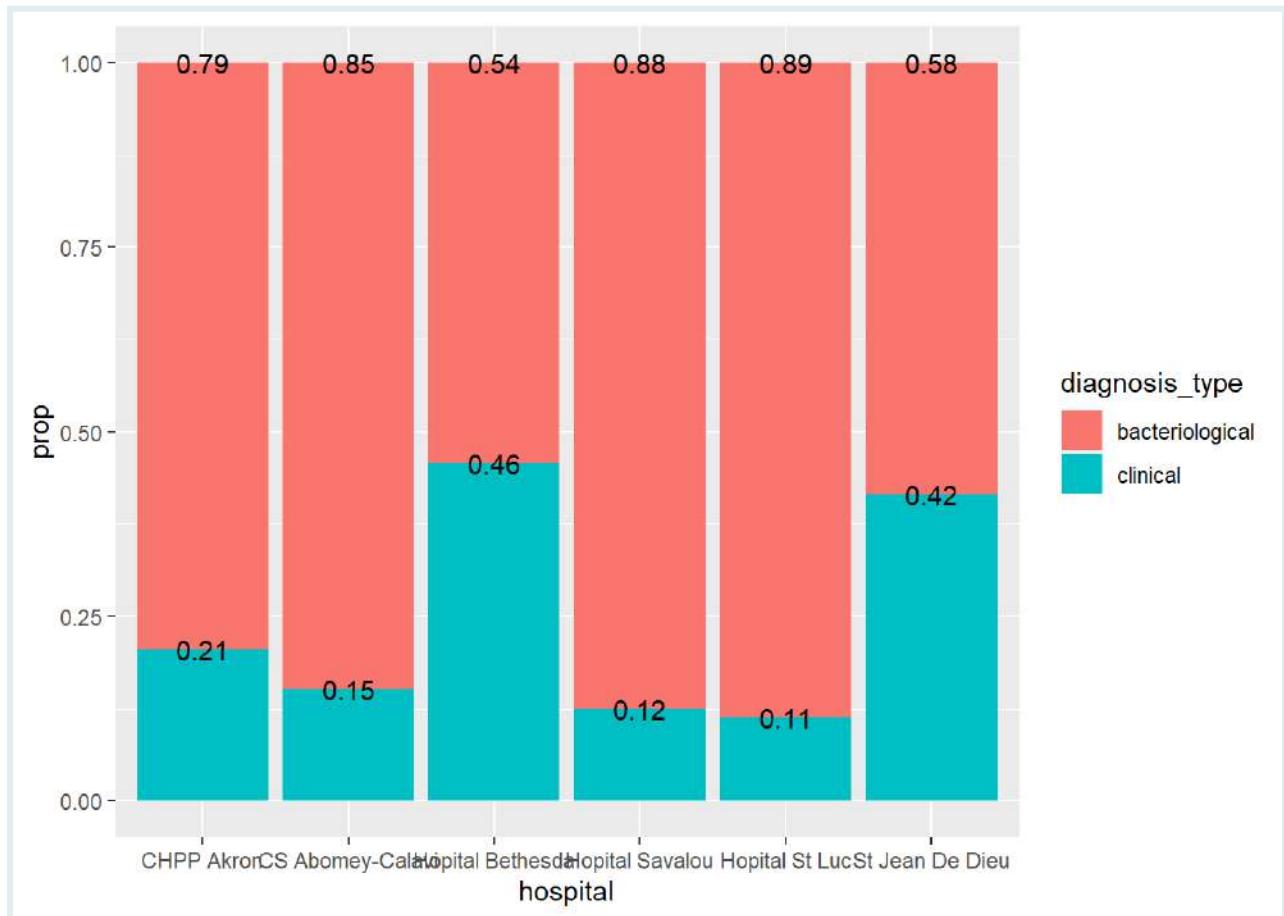
```



It's a good start, but obviously, we still have some work to do to make it look nicer!

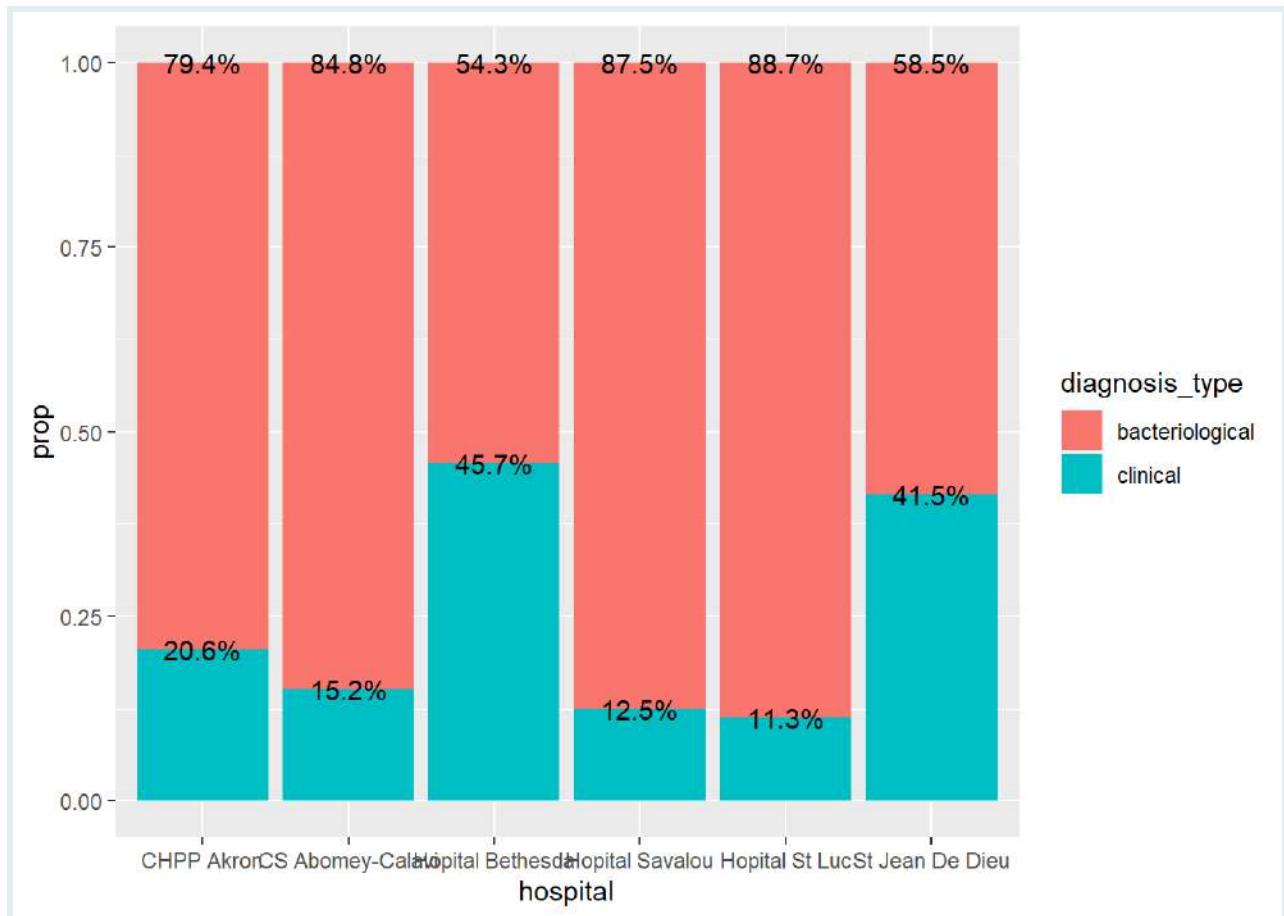
Before adjusting our labels, let's handle those decimals. We could reduce the number of decimals like this:

```
hosp_dx_fill +
  geom_text(aes(label = round(prop, 2)),
            position = position_fill())
```



However, the better method is this:

```
hosp_dx_fill +  
  geom_text(aes(label = scales::percent(prop)),  
            position = position_fill())
```



SIDE NOTE

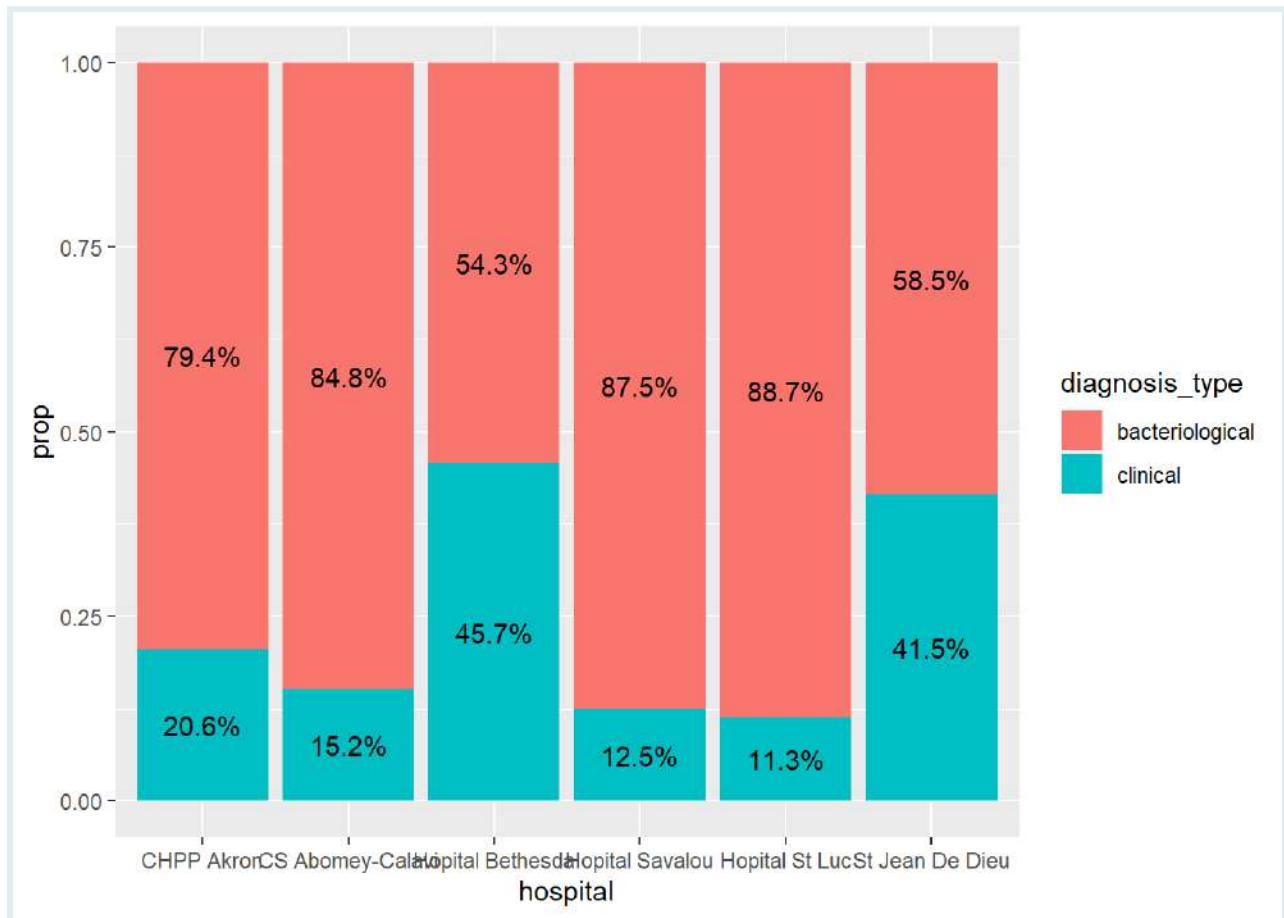


The `{scales}` package is commonly used with `{ggplot2}` for customizing aesthetics, transforming axis scales, formatting labels, defining color palettes, and more.

The `scales::percent(prop)` function we used in the code above with `geom_text()` converts the proportions (values from our `prop` variable) into a percentage format and adds percentage signs. We can also control the number of displayed digits using the `accuracy` argument (see below).

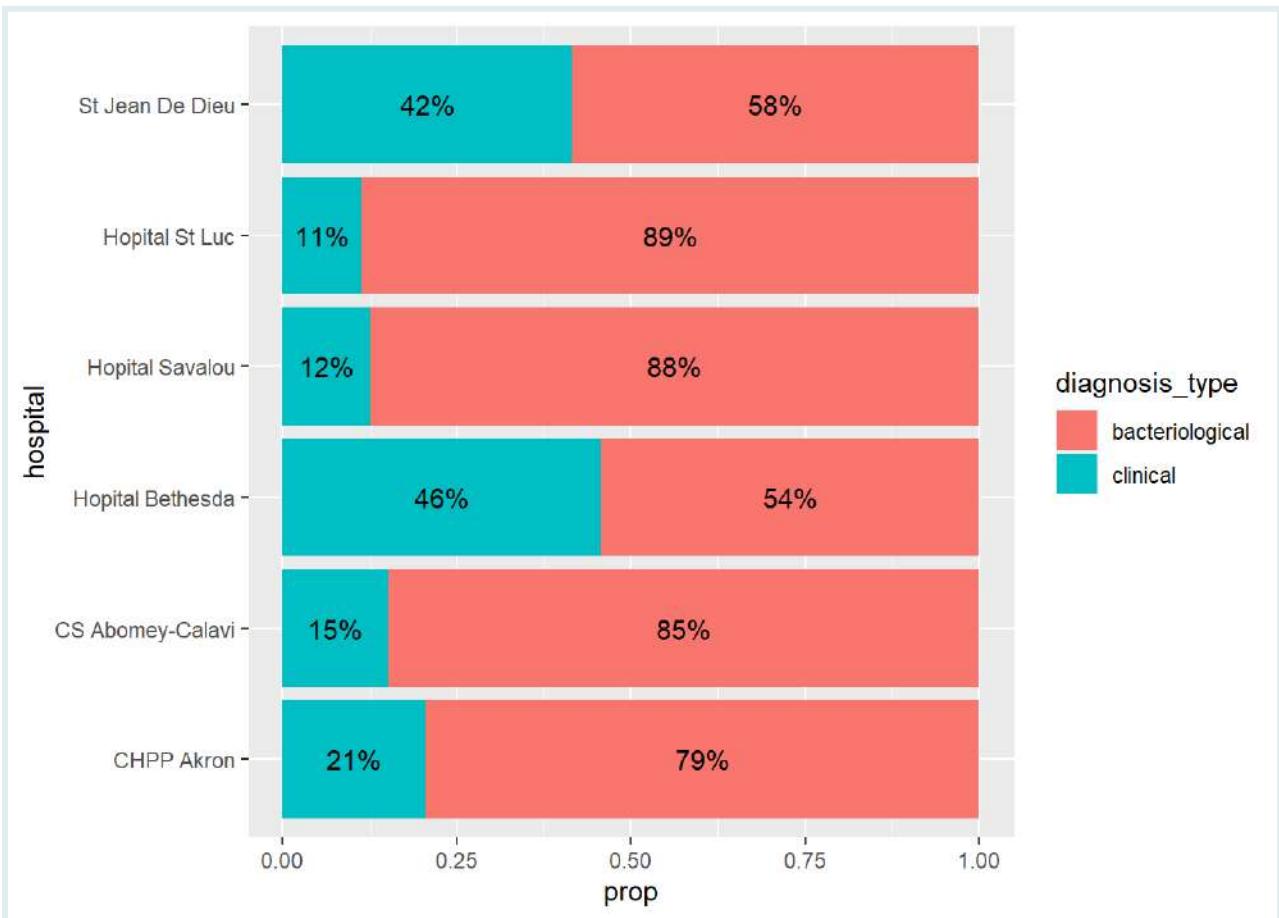
Next, we can center the labels using `vjust` in the `position_fill()` function

```
hosp_dx_fill +
  geom_text(aes(label = scales::percent(prop)),
            position = position_fill(vjust = 0.5)) # center labels
```



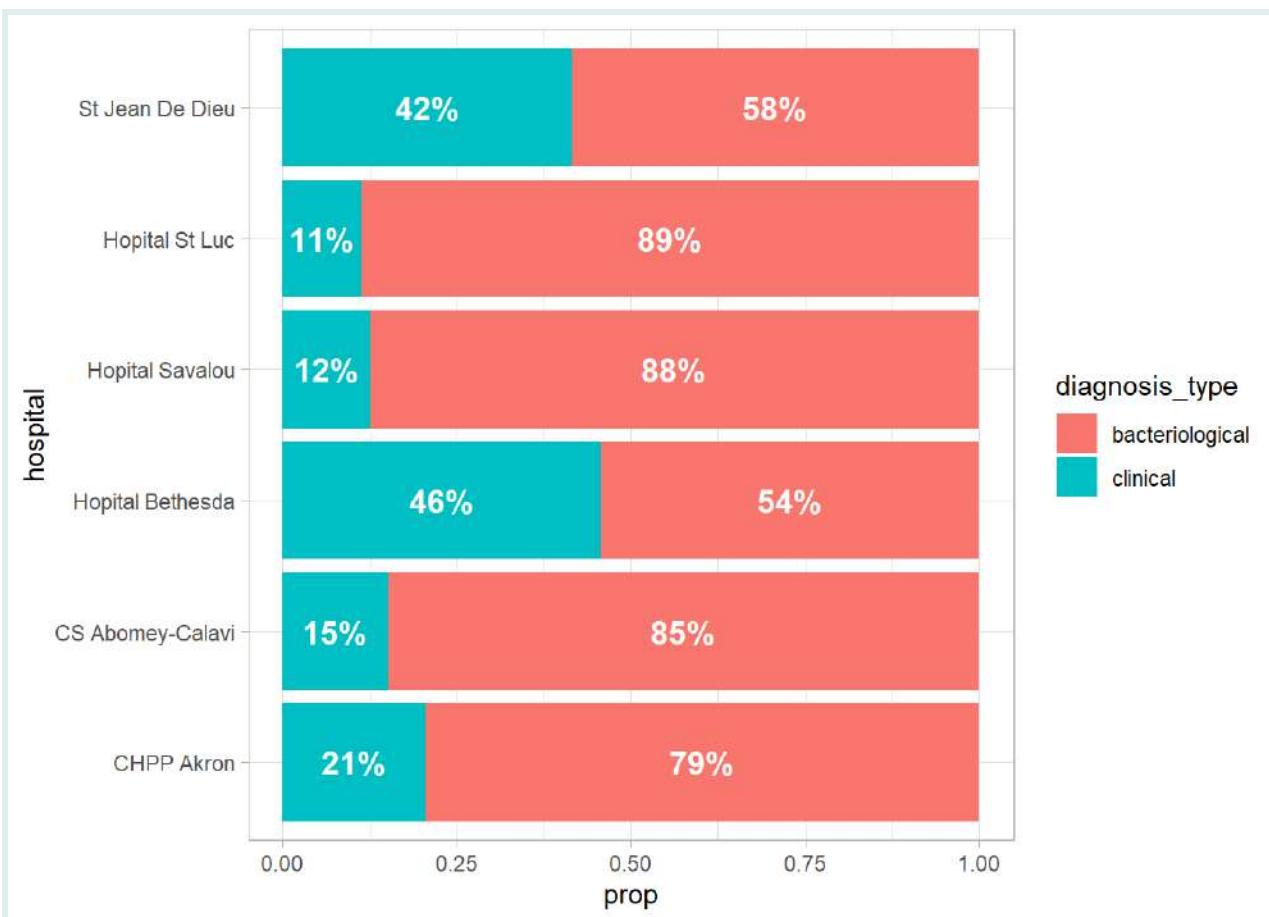
It looks great, but we can do better! Using flipped coordinates in bar charts can greatly improve readability:

```
hosp_dx_fill +
  geom_text(aes(label = scales::percent(prop, accuracy = 1)),
            position = position_fill(vjust = 0.5)) +
  coord_flip()
```



Great, now we can add some additional aesthetic tweaks:

```
hosp_dx_fill +
  geom_text(aes(label = scales::percent(prop, accuracy = 1)),
            position = position_fill(vjust = 0.5),
            color = "white", # Change text color
            fontface = "bold", # Make it bold
            size = 4.5) + # Change font size
  theme_light() +
  coord_flip()
```



Amazing! Let's move on to our last section where we'll take a look at circular plots.

Q: Creating Percent-Stacked Bar Plots with Labels

Transform the `aus_tb_notifs` data into a percent-stacked bar plot, with a bar for each year, and the fill aesthetic mapped to the area type (rural vs urban).



Label each segment with the percentage of cases using `geom_text()`. Format the labels as percentages.

You can use the code and comments below as a guide:

```
# Pivot the data
aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
              names_to = "area_type",
              values_to = "cases")
```



```
## # A tibble: 104 × 4
##   year quarter area_type cases
##   <dbl> <chr>   <chr>     <dbl>
## 1 2010 Q1    rural      4
## 2 2010 Q1    urban     87
## 3 2010 Q2    rural      4
## 4 2010 Q2    urban     98
## 5 2010 Q3    rural      5
## 6 2010 Q3    urban    101
## 7 2010 Q4    rural     10
## 8 2010 Q4    urban    124
## 9 2011 Q1    rural      5
## 10 2011 Q1   urban    81
## # i 94 more rows
```

```
# Then summarize and calculate proportions
```

```
# Next create the percent-stacked bar plot
# For the label, use the scales::percent() function with an
# accuracy of 1
# Use position_fill() to center the labels
```

Labeling circular plots

As usual, let's begin by summarizing the data.

We'll calculate the total number of cases for each hospital by grouping the data based on the `hospital` variable and then calculating the sum of cases in each group.

```
total_results <- tb_outcomes %>%
  group_by(hospital) %>%
  summarise(
    total_cases = sum(cases))
total_results
```

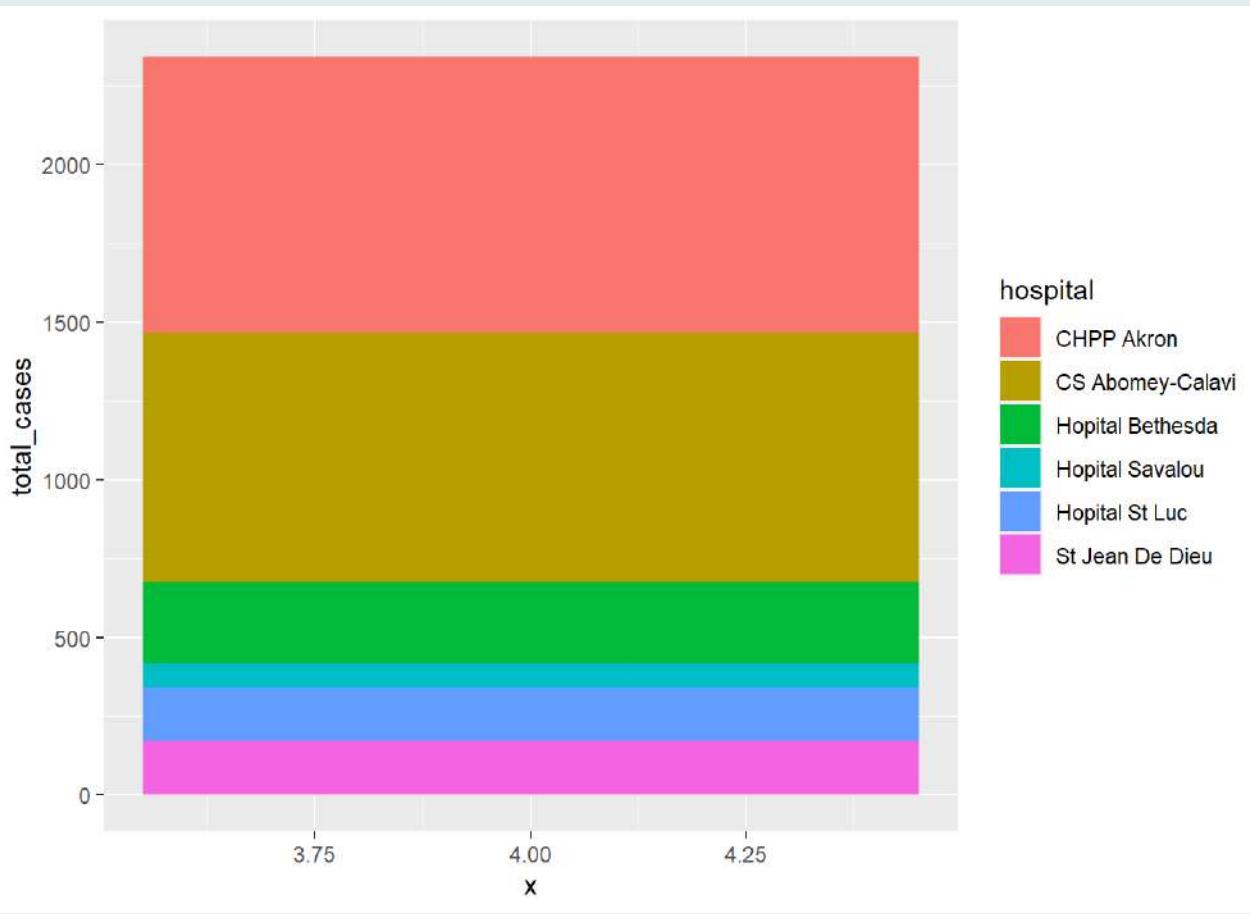
Now that we have our new dataset, let's start by creating a simple bar chart. You may recall from the previous lesson that a pie chart is essentially a round version of a 100% stacked bar chart.

```

results_stack <- ggplot(total_results,
  aes(x=4, # Set an arbitrary x value
      y=total_cases,
      fill=hospital)) +
  geom_col()

results_stack

```



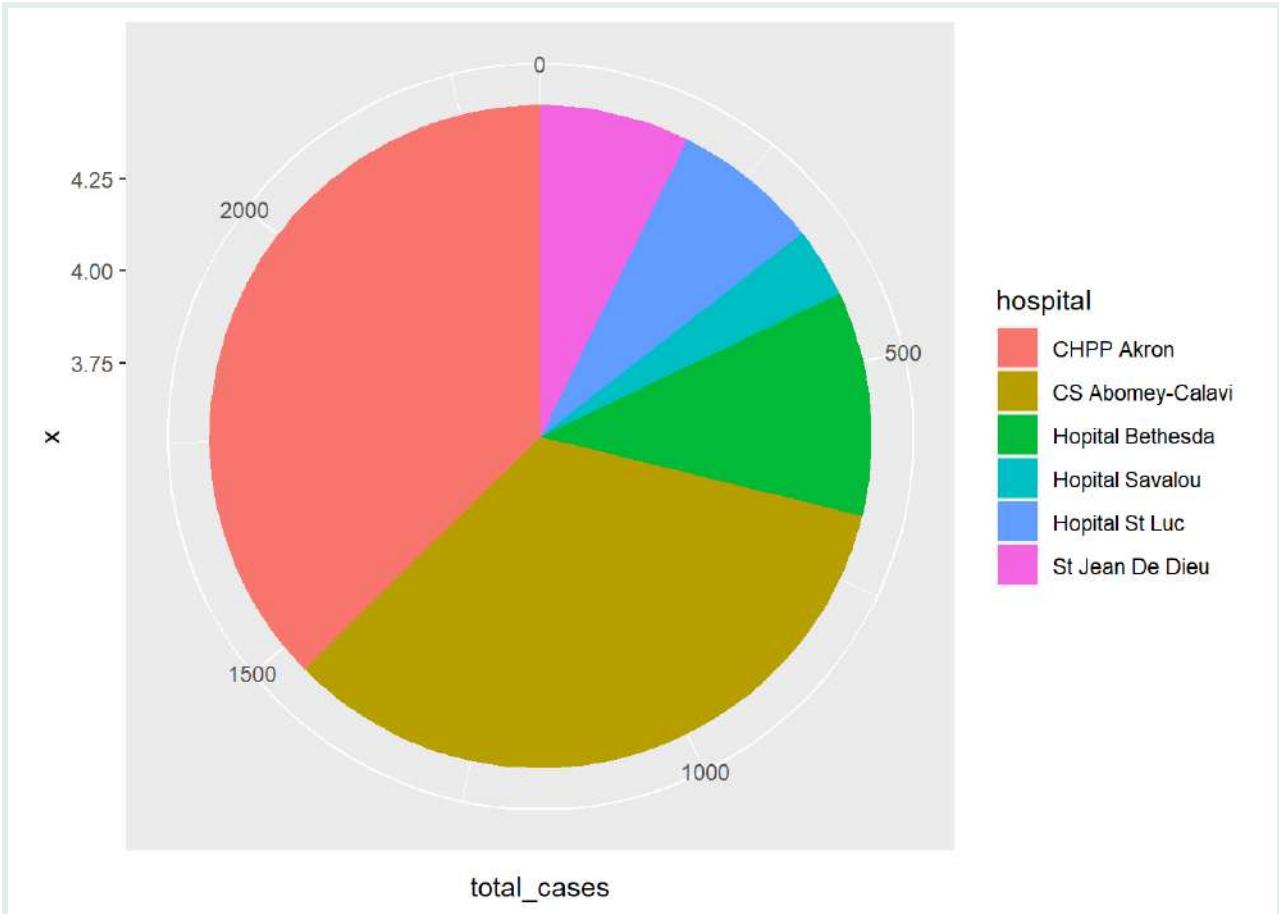
Now, we can create our basic pie chart. As we learned in the last lesson, to transform linear coordinates into polar coordinates, we use the `coord_polar()` function. The `theta` parameter defines which aesthetic variable should be mapped to the angular coordinate in the polar coordinate system. By specifying "`y`", we use the height of the bars to determine the angle of each slice in our pie chart.

```

outcome_pie <- results_stack +
  coord_polar(theta = "y")

outcome_pie

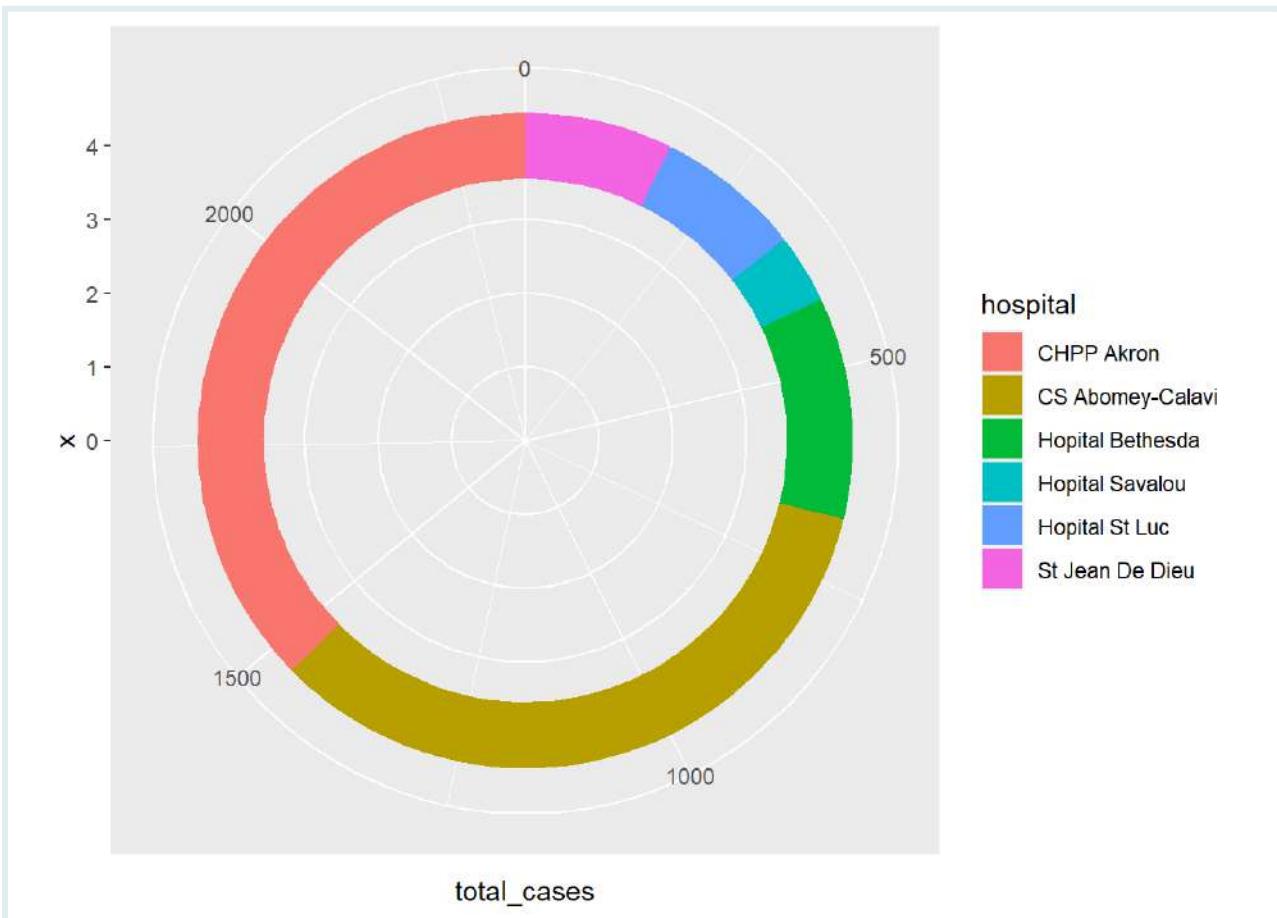
```



Great! This will serve as our base pie chart. Next, let's create a base donut chart using `xlim()`.

```
outcome_donut <- ggplot(total_results,
  aes(x = 4,
      y = total_cases,
      fill = hospital)) +
  geom_col() +
  xlim(c(0, 4.5)) +
  coord_polar(theta = "y")

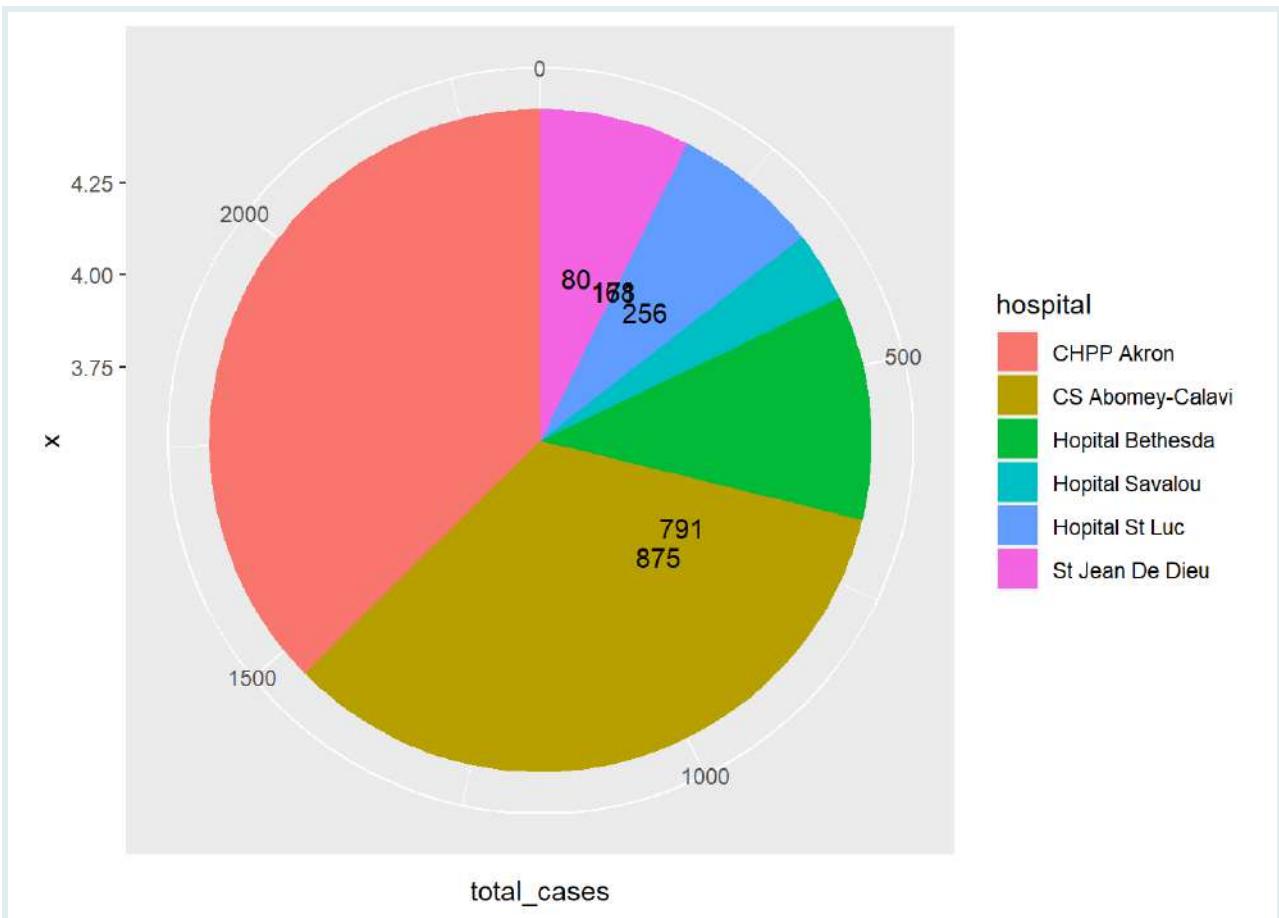
outcome_donut
```



Alright, we're ready to move on to labelling!

Let's add labels to our pie chart using `geom_text()`.

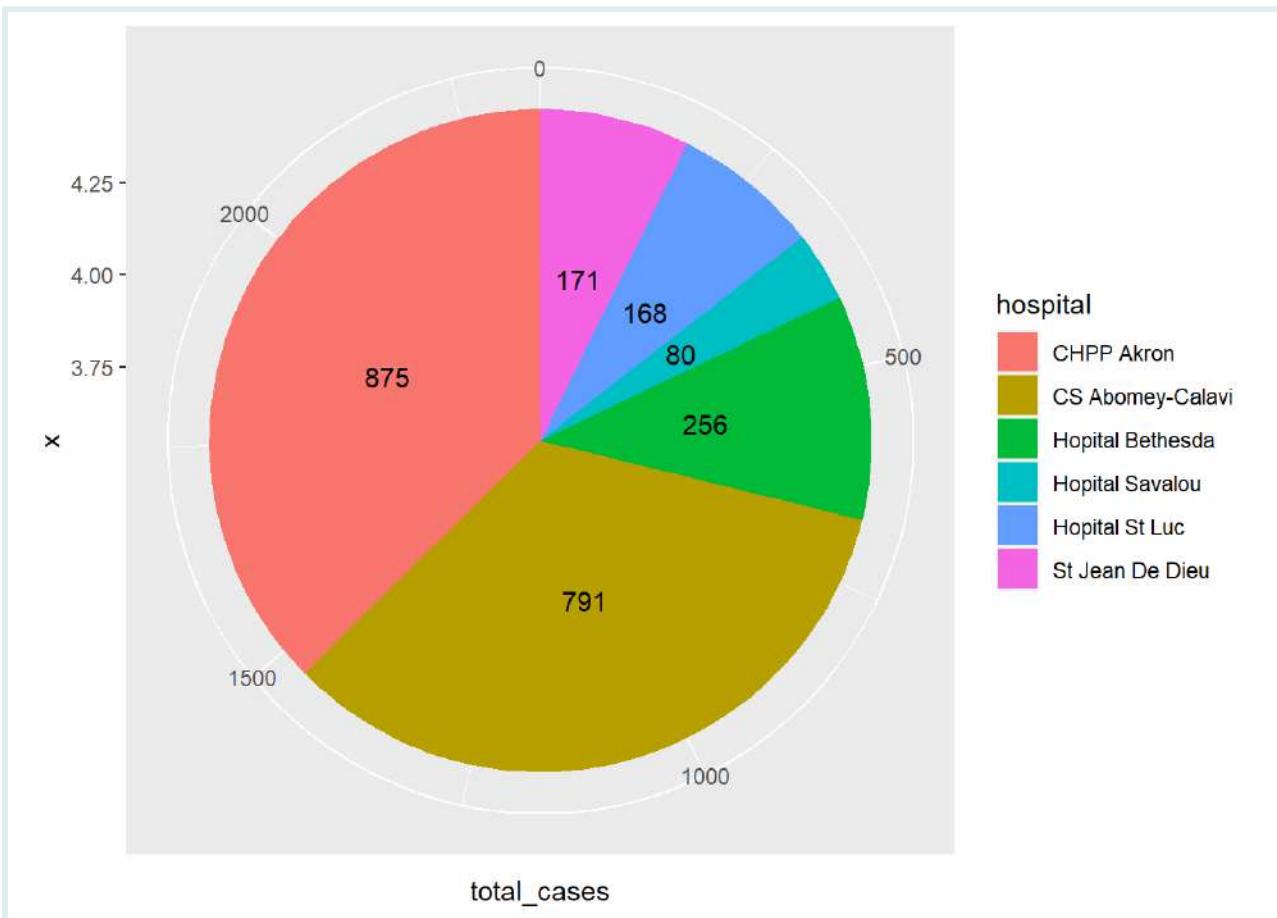
```
outcome_pie +  
  geom_text(aes(label = total_cases))
```



You'll notice that the numbers appear in the wrong segments because we haven't added a `position` adjustment to the labeling geometry yet.

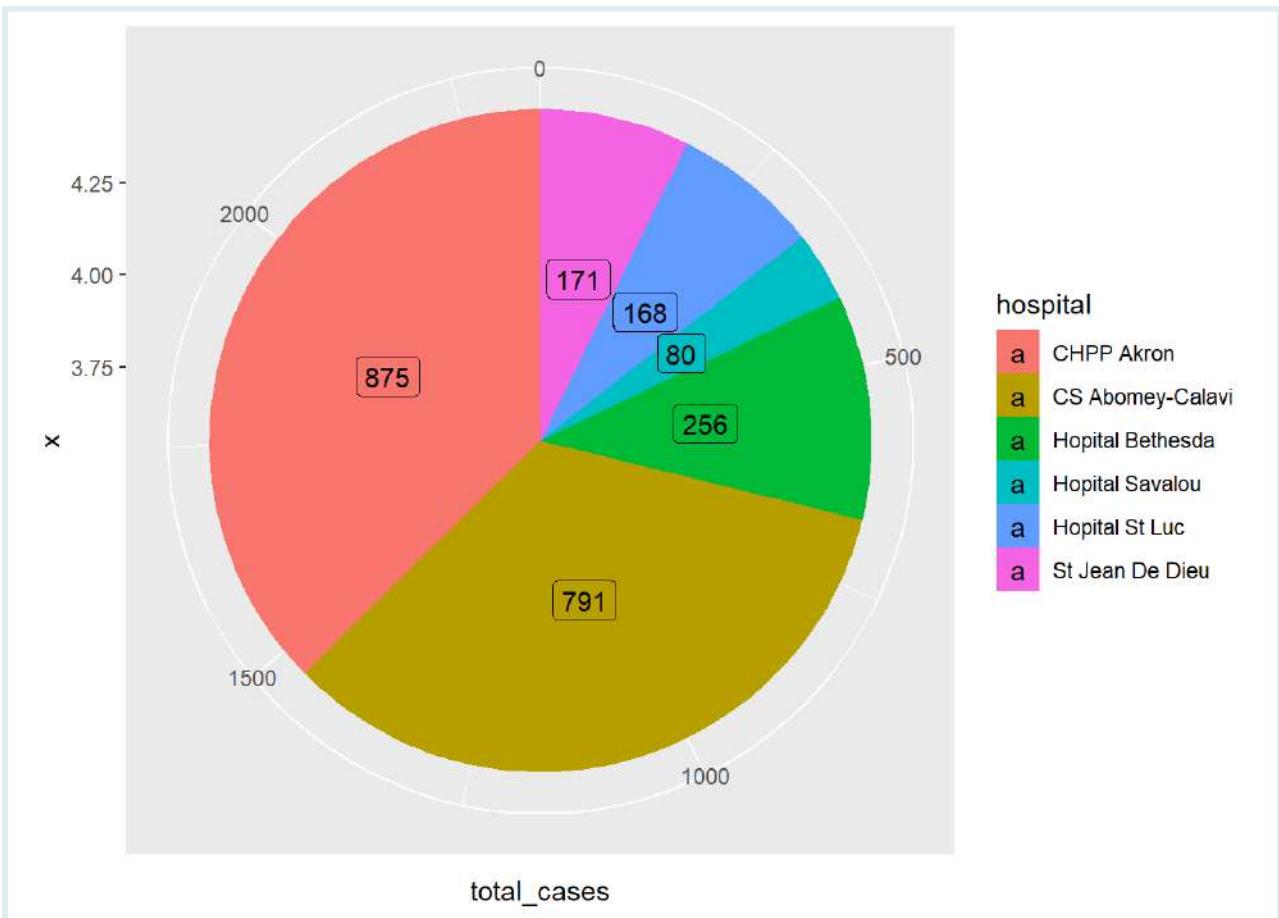
Now, just as we did previously, we will use the `position_stack()` argument with `vjust` to center the labels.

```
outcome_pie +
  geom_text(aes(label = total_cases),
            position = position_stack(vjust = 0.5)) # Center the labels
```



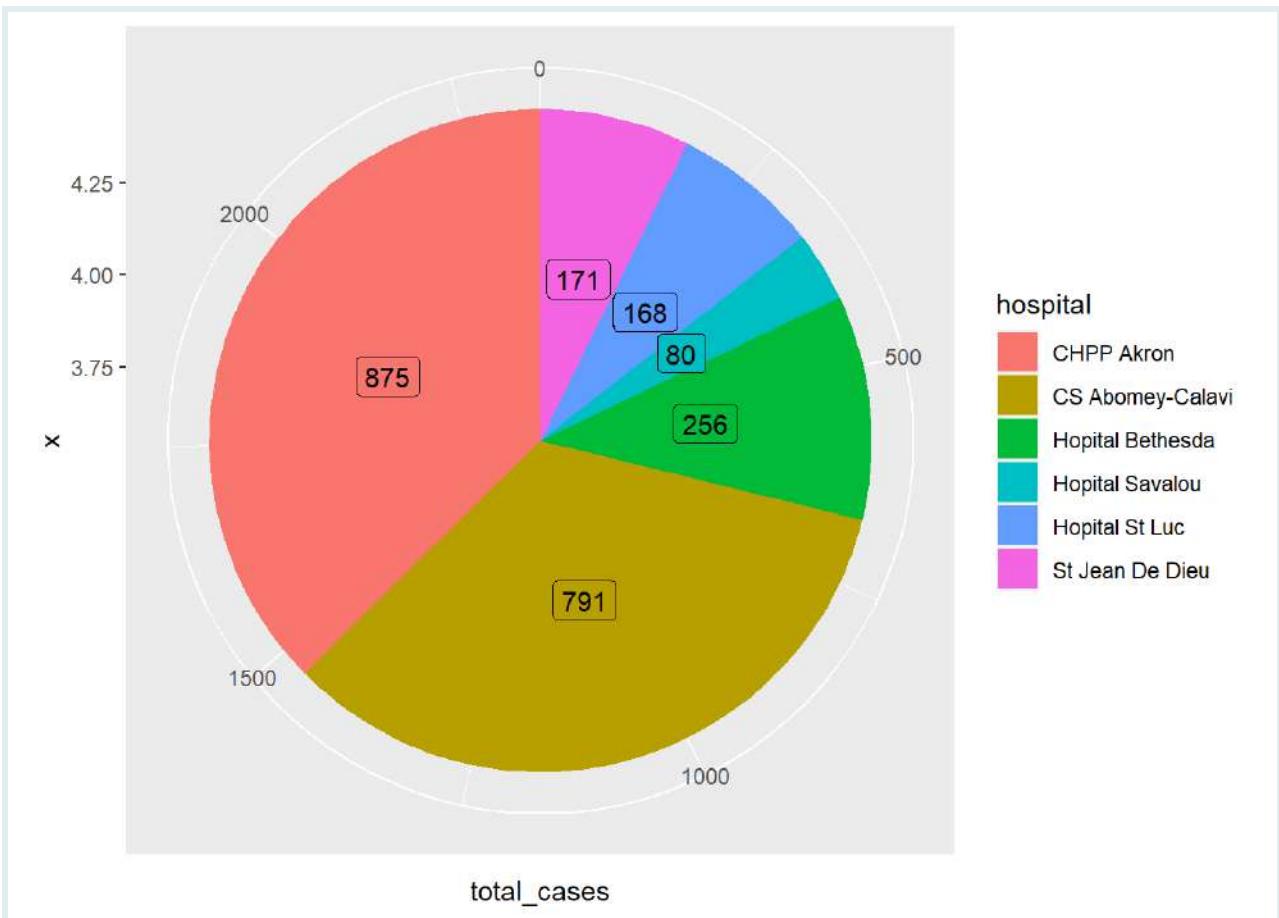
We can do the same with `geom_label()`.

```
# Similar adjustment with geom_label()
outcome_pie +
  geom_label(aes(label = total_cases),
             position = position_stack(vjust = 0.5))
```



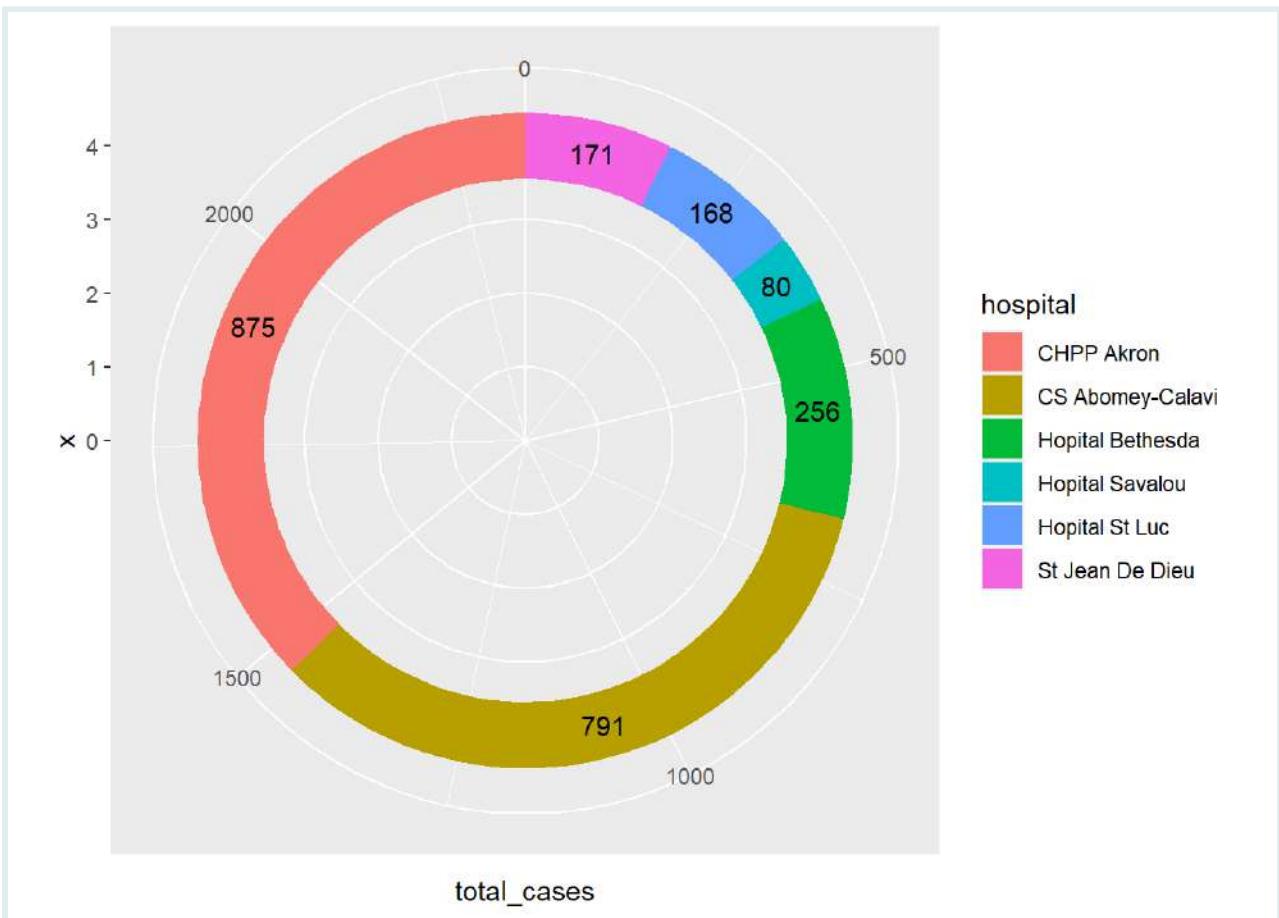
To remove the letter “a” from the legend, we can use `show.legend = FALSE`:

```
outcome_pie +
  geom_label(aes(label = total_cases),
             position = position_stack(vjust = 0.5),
             show.legend = FALSE)
```



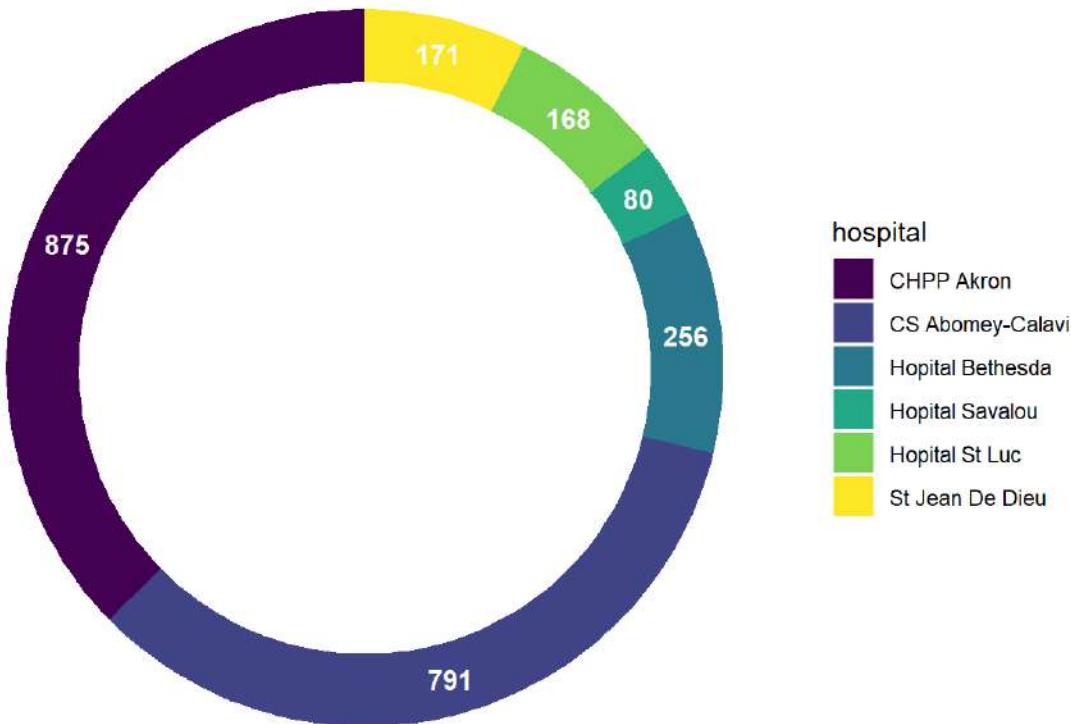
Next, let's move on to our basic donut chart. We'll label it using `geom_text()`:

```
outcome_donut +  
  geom_text(aes(label = total_cases),  
            position = position_stack(vjust = 0.5))
```



To finish, we can make some additional aesthetic adjustments. Here, we enhance the chart's aesthetics by applying `theme_void()` to remove cluttered background elements, introducing a new color palette with `scale_fill_viridis_d()`, and adjusting the text labels using `geom_text()` with white and bold text for better visibility and contrast.

```
# Additional aesthetic modifications
outcome_donut +
  geom_text(aes(label = total_cases),
            position = position_stack(vjust = 0.5),
            color = "white",
            fontface = "bold") +
  theme_void() +
  scale_fill_viridis_d()
```



Congratulations, it looks great!



Q: Labeling Pie Charts

Plot total TB cases in all rural vs urban areas in the `aus_tb_notifs` dataset as a pie chart. Use `geom_text()` to place labels correctly, indicating the number of cases in that area.

You can use the code and comments below as a guide:

```
# Pivot then summarize the total cases per area type
aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases") %>%
  group_by(area_type) %>%
  summarise(total_cases = sum(cases))
```



```
## # A tibble: 2 × 2
##   area_type total_cases
##   <chr>          <dbl>
## 1 rural            394
## 2 urban           4981
```

```
# Now, create the pie chart
# For the text labels, use geom_text() and position_stack(vjust = 0.5)
```



Pro-Tip: Enhancing Text Labels with ggtext

For advanced plotters seeking even more sophisticated control over text formatting in ggplot2, the {ggtext} package may come in handy. It allows the use of CSS to precisely format text elements, including options to embolden, italicize, change color and size, add superscripts/subscripts, and even embed images. Notably, you can apply multiple styles within the same text element, opening up new levels of creativity and customization.

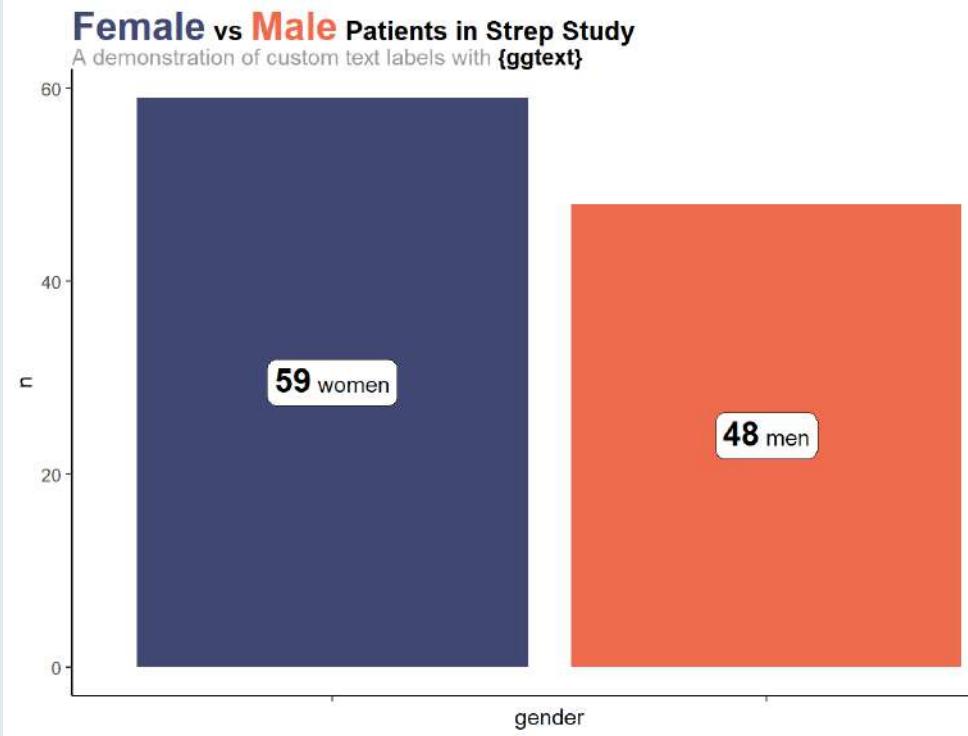
Consider the example below, which uses {ggtext} for the plot title, subtitle and bar labels:

PRO TIP



```
pacman:::p_load(tidyverse, ggtext, medicaldata)

# Data and Plot
medicaldata:::strep_tb %>%
  count(gender) %>%
  mutate(gender_label = paste0("**<span style='font-
    size:16pt'>", n, "</span>**",
    if_else(gender == "M", " men", "-
      women")) %>%
  ggplot(aes(x = gender, fill = gender, y = n)) +
  geom_col() +
  scale_fill_manual(values = c("M" = "#ee6c4d", "F" =
    "#424874")) +
  labs(
    title = "<b><span style='color:#424874; font-
      size:19pt'>Female</span> vs
      <span style='color:#ee6c4d; font-size:19pt'>Male</span>
      Patients in Strep Study</b>",
    subtitle = "<span style='color:gray60'>A demonstration of
      custom text labels with </span>**{ggtext}**") +
  theme_classic() +
  theme(plot.title = element_textbox_simple(),
        plot.subtitle = element_textbox_simple(),
        legend.position = "none",
        axis.text.x = element_blank()) +
  geom_richtext(aes(label = gender_label, y = n/2),
                label.r = grid::unit(5, "pt"), fill = "white")
```





To learn more about {ggtext}, visit [the package website](#).

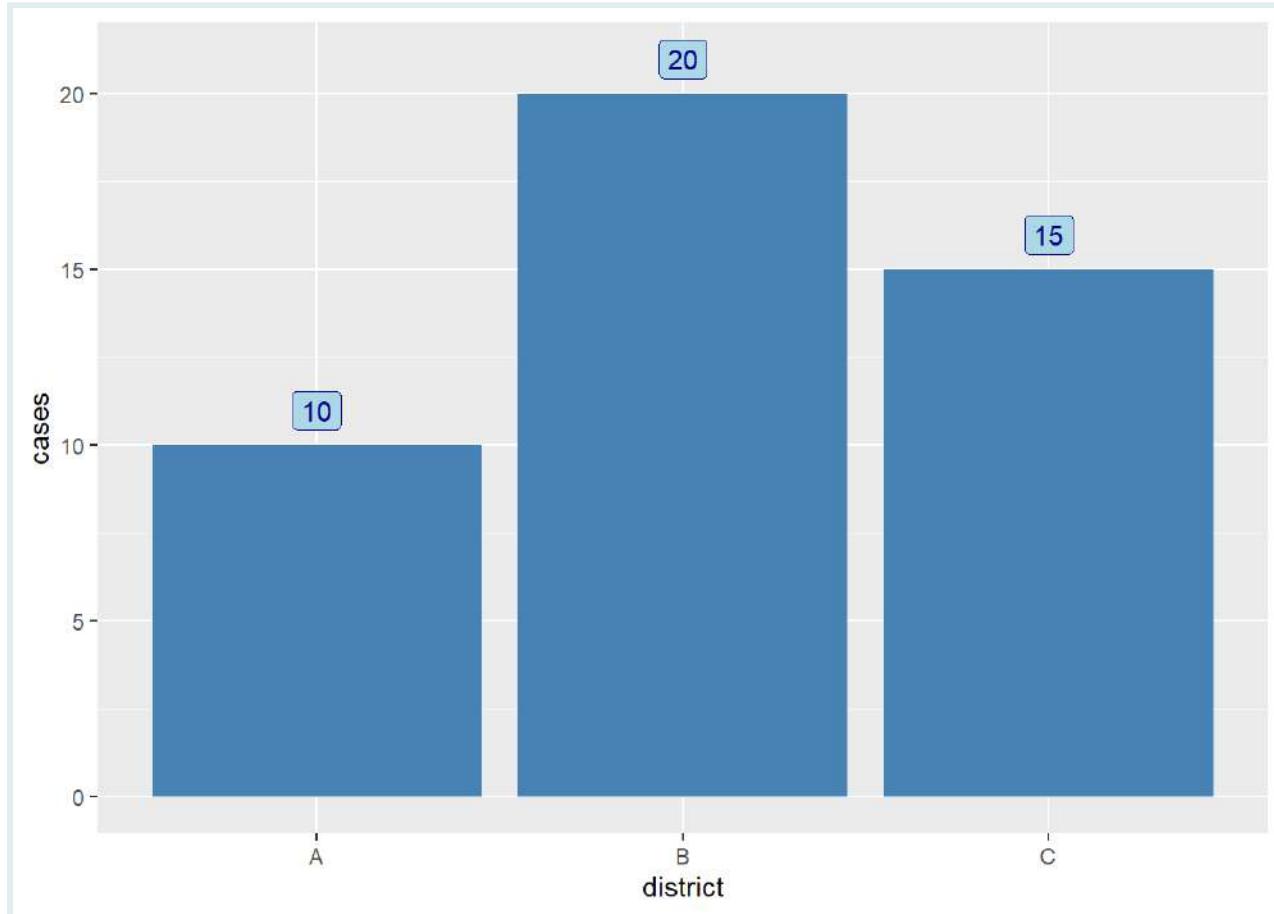
Wrap Up!

This lesson covered the use of `geom_text()` and `geom_label()` in ggplot2, demonstrating their application in various plot types. We learned how to add and adjust text labels for clarity and effective data presentation, from basic bar plots to more complex formats like stacked and circular plots. These skills are essential for enhancing the readability and informative value of graphical data representations in R.

Solutions

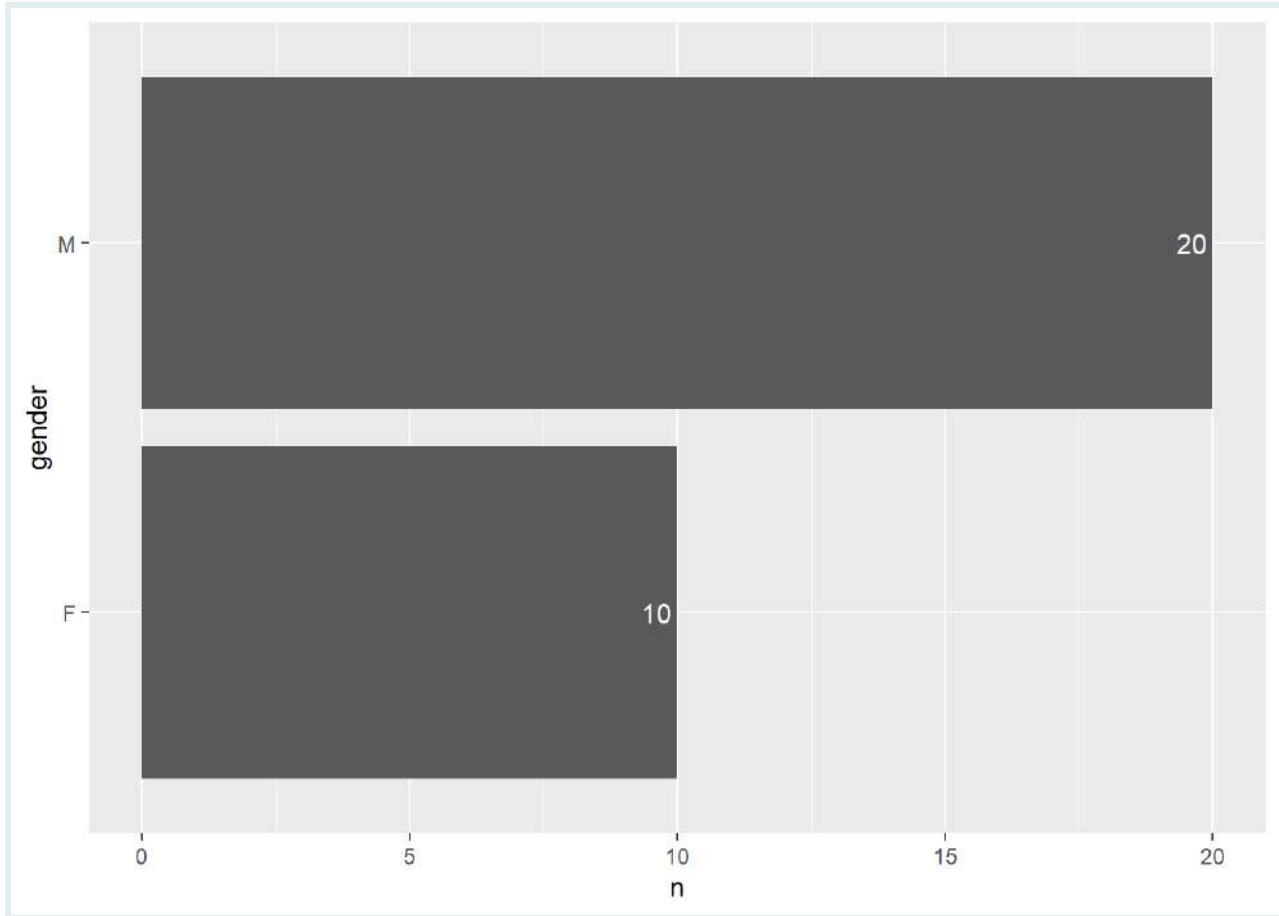
Q: Simple labeling

```
ggplot(district_cases, aes(x = district, y = cases)) +  
  geom_col(fill = "steelblue") +  
  geom_label(aes(label = cases),  
             nudge_y = 1, # Adjust to position the labels above the bars  
             fill = "lightblue", # Background color of the labels  
             color = "darkblue" # Text color  
  )
```



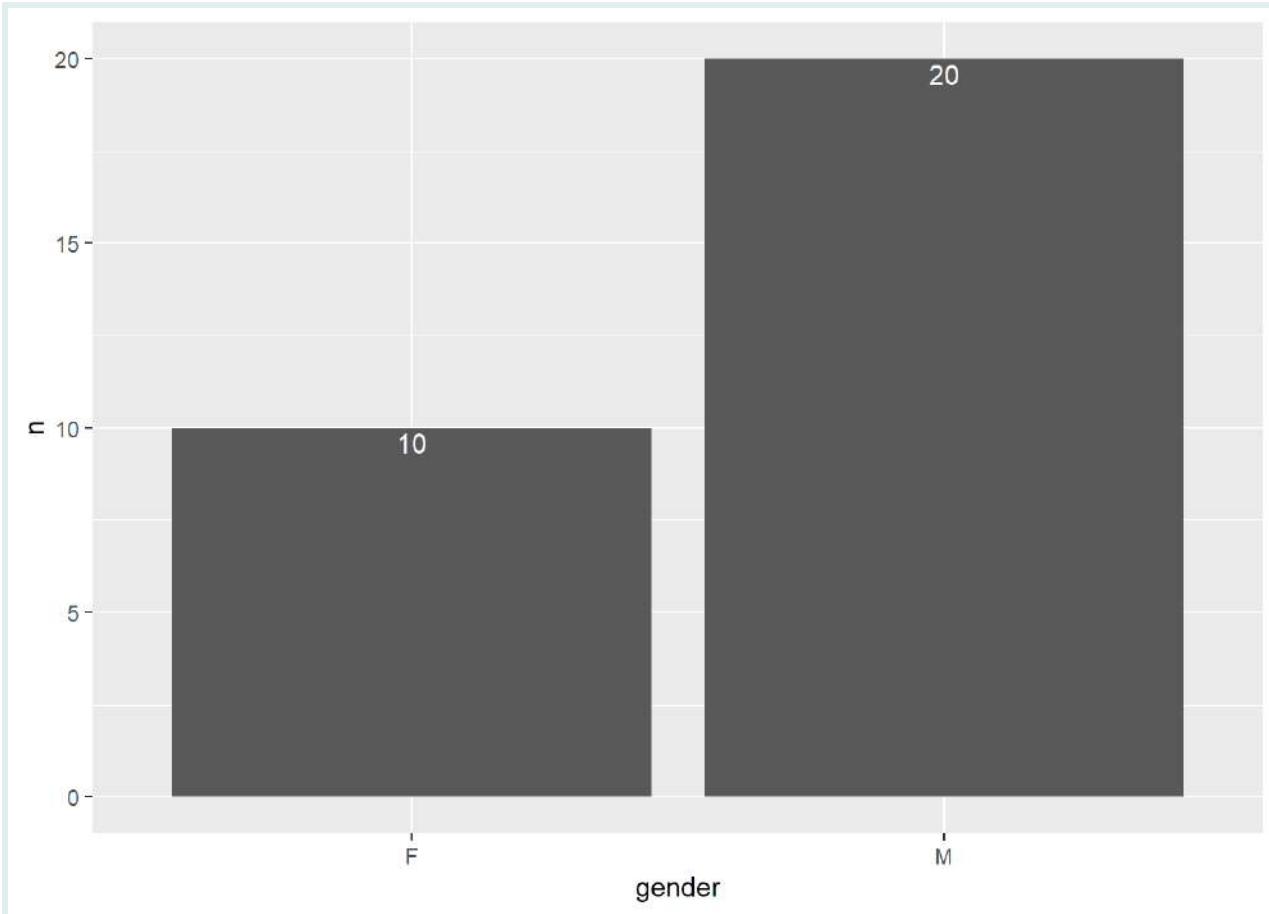
Q: Horizontal adjustment practice

```
# Adjusting text position inside the bar
ggplot(sample_gender, aes(x = n, y = gender)) +
  geom_col() +
  geom_text(aes(label = n), color = "white", hjust = 1.2)
```



Q: Vertical adjustment practice

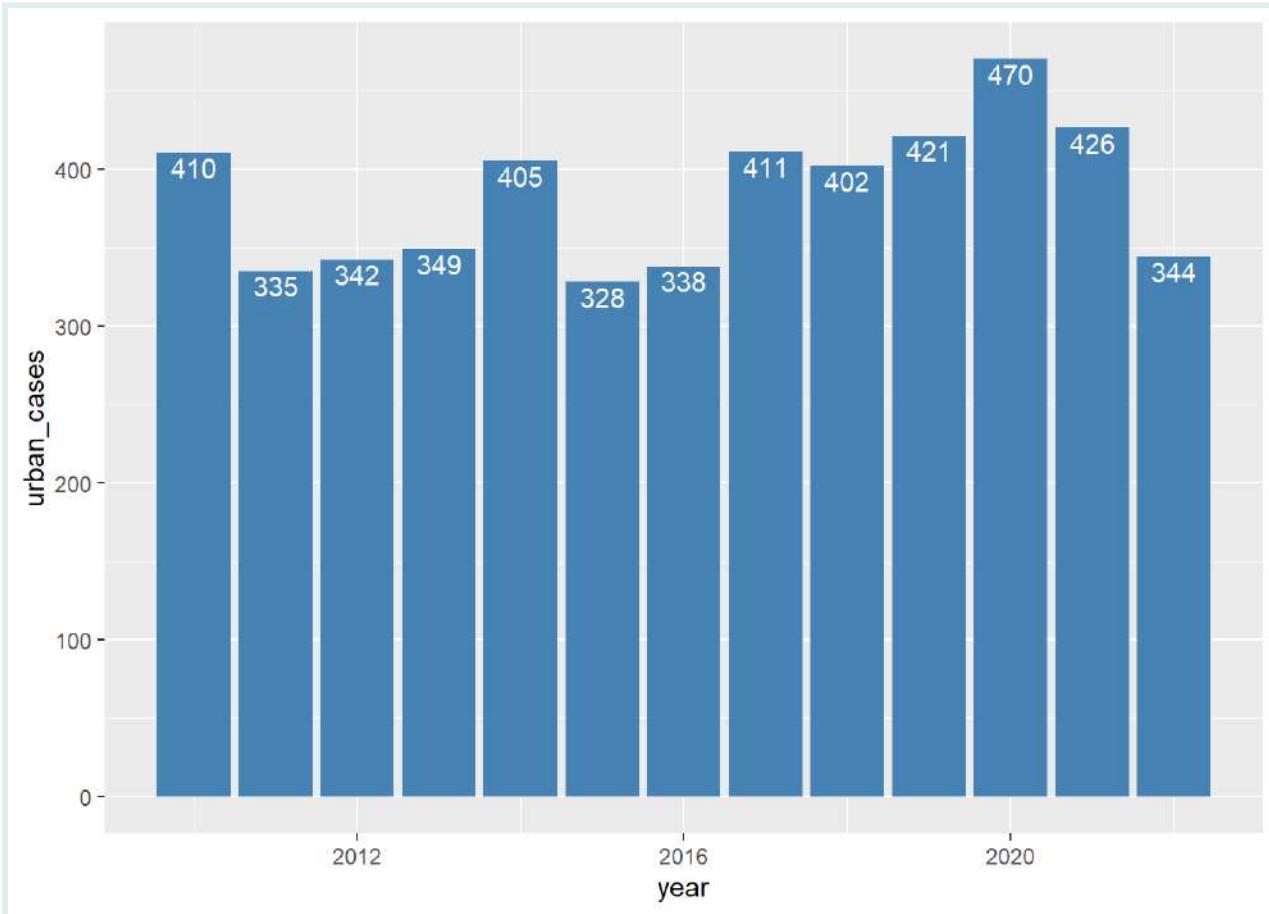
```
# Adjusting text position inside the bar
ggplot(sample_gender,
       aes(x = gender, y = n)) +
  geom_col() +
  geom_text(aes(label = n), color = "white", vjust = 1.2)
```



Q: Summarize then plot

```
# aggregate the data by year and sum up the urban cases
urban_cases <-
  aus_tb_notifs %>%
  group_by(year) %>%
  summarize(urban_cases = sum(urban))

# plot the data
ggplot(urban_cases, aes(x = year, y = urban_cases)) +
  geom_col(fill = "steelblue") +
  geom_text(aes(label = urban_cases),
            color = "white",
            vjust = 1.2)
```

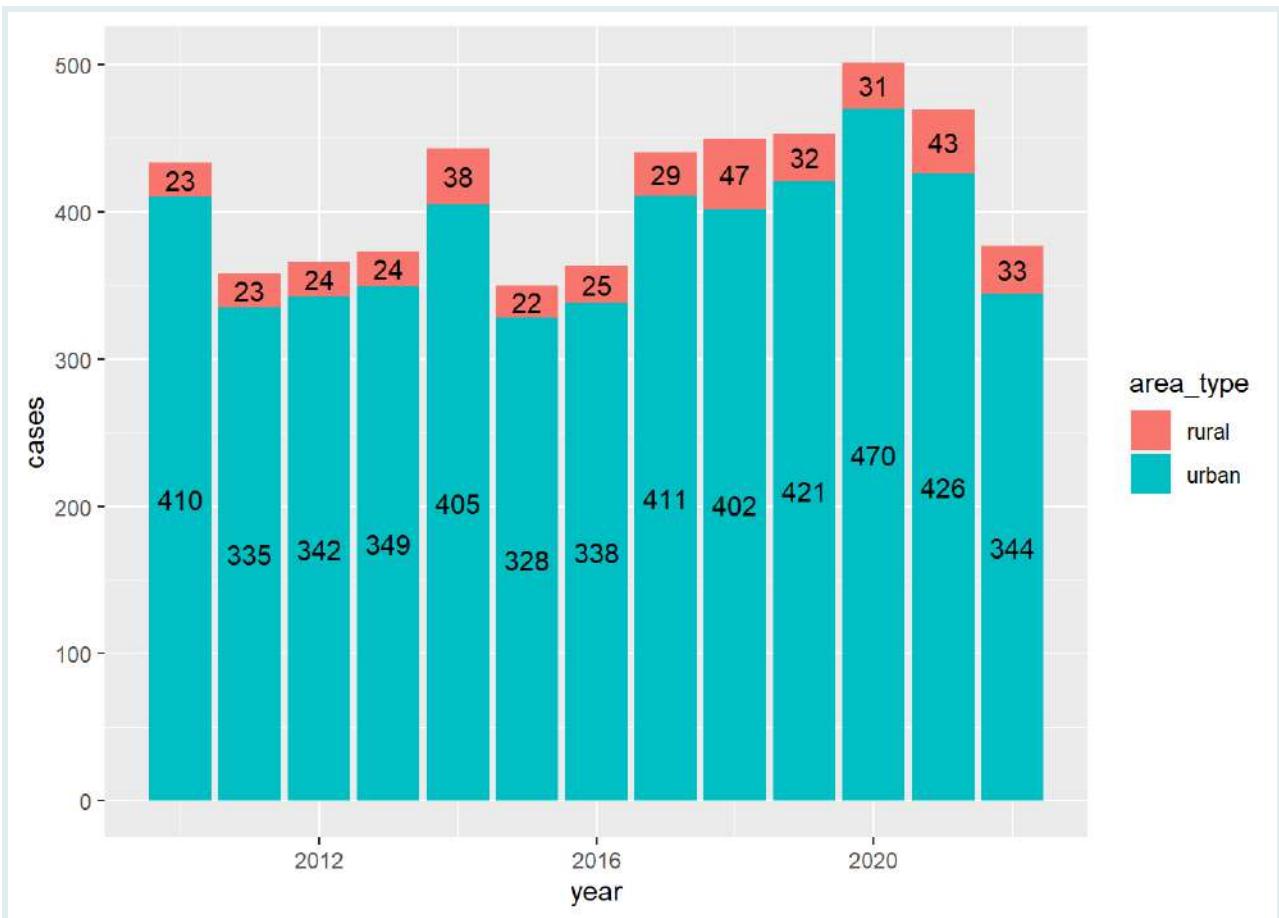


Q: Practice with labeling stacked plots

```
# Pivot the data
aus_tb_pivoted <- aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases")

# Summarize the data by year and area type
aus_tb_summarized <- aus_tb_pivoted %>%
  group_by(year, area_type) %>%
  summarise(cases = sum(cases))

# Create the stacked bar plot
ggplot(aus_tb_summarized, aes(x = year, y = cases, fill = area_type)) +
  geom_col() +
  geom_text(aes(label = cases),
            position = position_stack(vjust = 0.5))
```

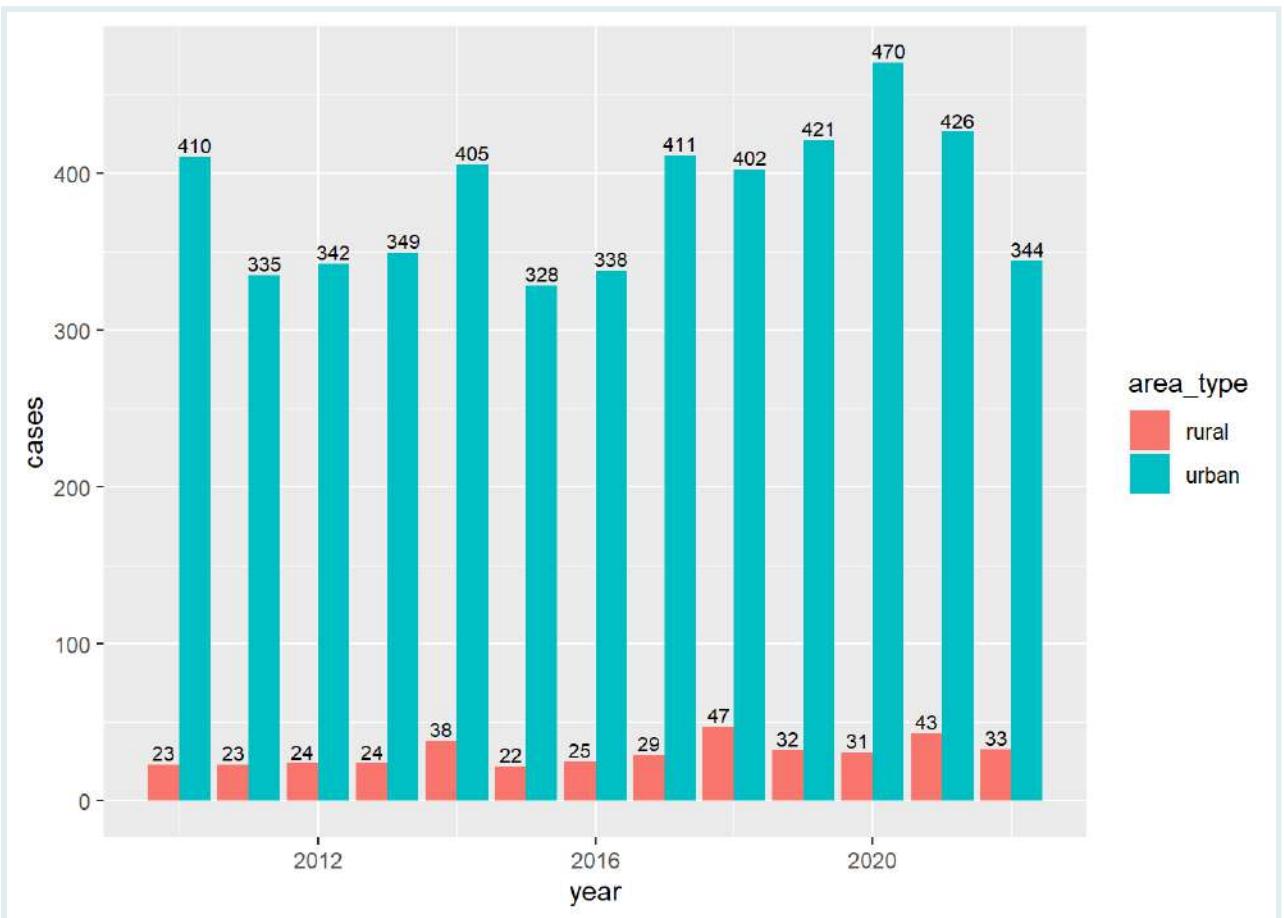


Q: Practice with labeling dodged bar plots

```
# Pivot the data
aus_tb_pivoted <- aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases")

# Summarize the data by year and area type
aus_tb_summarized <- aus_tb_pivoted %>%
  group_by(year, area_type) %>%
  summarise(cases = sum(cases))

# Create the dodged bar plot
ggplot(aus_tb_summarized, aes(x = year, y = cases, fill = area_type)) +
  geom_col(position = "dodge") +
  geom_text(aes(label = cases),
            position = position_dodge(width = 0.9),
            vjust = -0.3,
            size = 2.8)
```

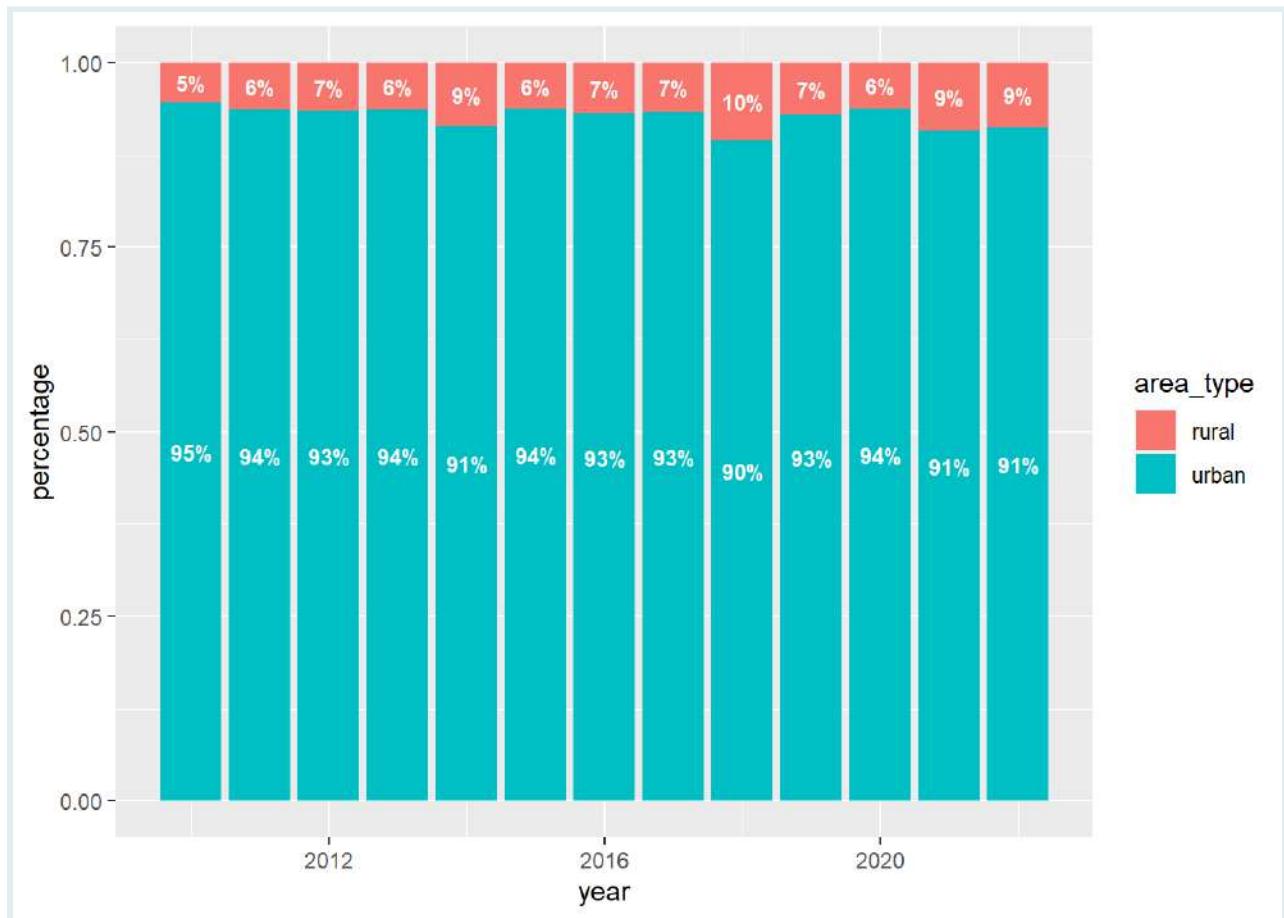


Q: Creating Percent-Stacked Bar Plots with Labels

```
# Pivot the data
aus_tb_pivoted <- aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases")

# Summarize and calculate proportions
aus_tb_percent <- aus_tb_pivoted %>%
  group_by(year, area_type) %>%
  summarise(cases = sum(cases)) %>%
  mutate(percentage = cases / sum(cases))

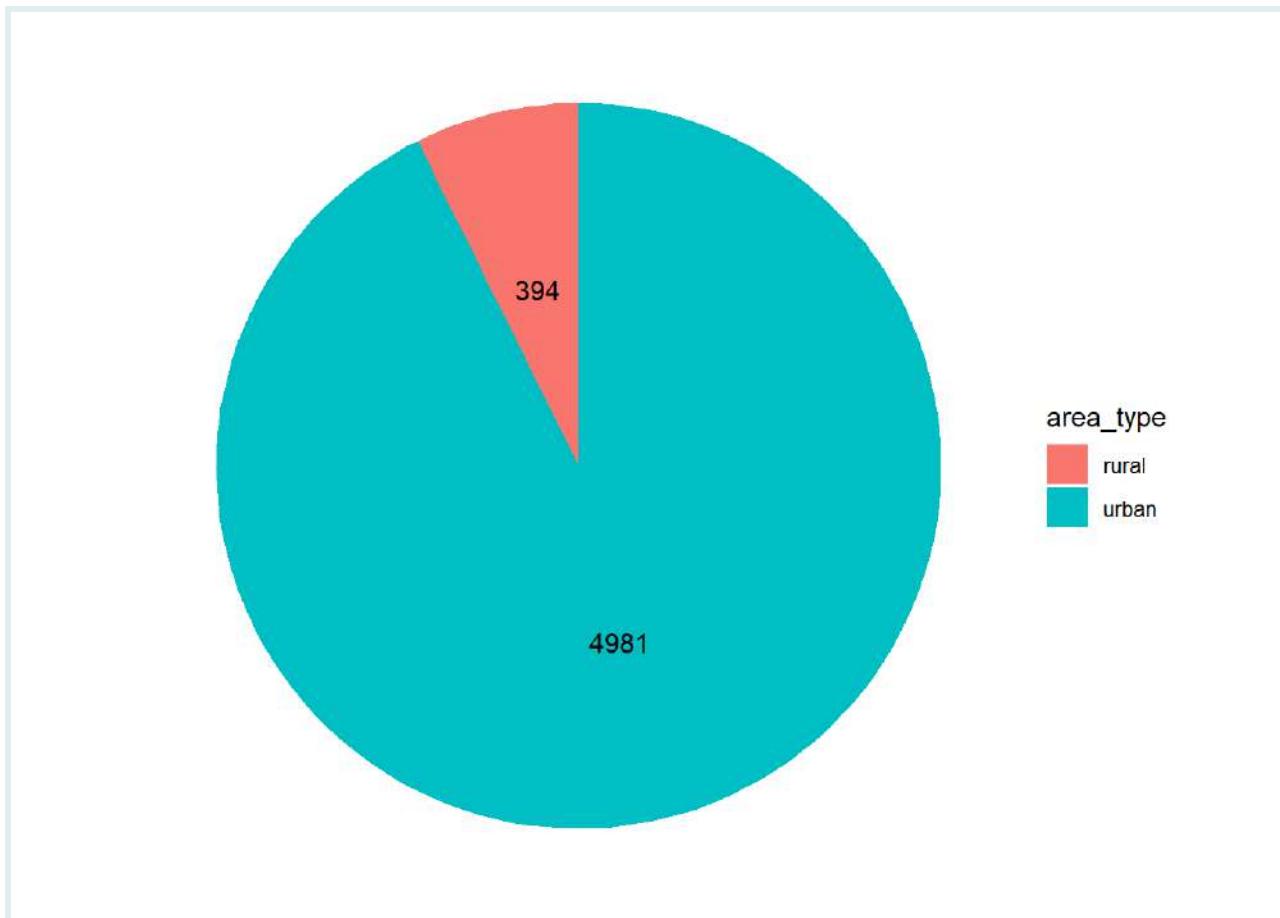
# Create the percent-stacked bar plot
ggplot(aus_tb_percent, aes(x = year, y = percentage, fill = area_type)) +
  geom_col(position = "fill") +
  geom_text(aes(label = scales::percent(percentage, accuracy = 1)),
            position = position_fill(vjust = 0.5),
            size = 3,
            color = "white",
            fontface = "bold")
```



Q: Labeling Pie Charts

```
# Summarize the total cases per quarter
aus_tb_rural_urban <- aus_tb_notifs %>%
  pivot_longer(cols = c(rural, urban),
               names_to = "area_type",
               values_to = "cases") %>%
  group_by(area_type) %>%
  summarise(total_cases = sum(cases))

# Create the pie chart
ggplot(aus_tb_rural_urban, aes(x = "", y = total_cases, fill = area_type)) +
  geom_col() +
  geom_text(aes(label = total_cases), position = position_stack(vjust = 0.5)) +
  coord_polar(theta = "y") +
  theme_void()
```



Contributors

The following team members contributed to this lesson:



BENNOEUR HSIN

Data Science Education Officer
Data Visualization enthusiast



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

Some material in this lesson was adapted from the following sources:

- Horst, Allison. "Allisonhorst/Dplyr-Learnr: A Colorful Introduction to Some Common Functions in Dplyr, Part of the Tidyverse." GitHub. Accessed April 6, 2022. <https://github.com/allisonhorst/dplyr-learnr>.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Creating Choropleth maps with {ggplot2}

Creating Choropleth maps with {ggplot2}
Introduction
Data Preparation
Creating a Beautiful Choropleth Map with {ggplot2}
Color Scaling
Facet Wrap vs. Grid
WRAP UP!
Solutions

Creating Choropleth maps with {ggplot2}

Learning objectives

1. Choropleth Maps with {ggplot2}:

- Master `ggplot()` and `geom_sf()` functions for map visualization.

2. Data Matching with Polygons:

- Obtain boundaries and disease-related data.
- Combine data based on administrative levels.

3. Color Scaling Techniques:

- Implement color scales for both continuous and discrete data types.

4. Faceting for Map Visualization:

- Use `facet_wrap()` and `facet_grid()` to create small multiple maps.

Introduction

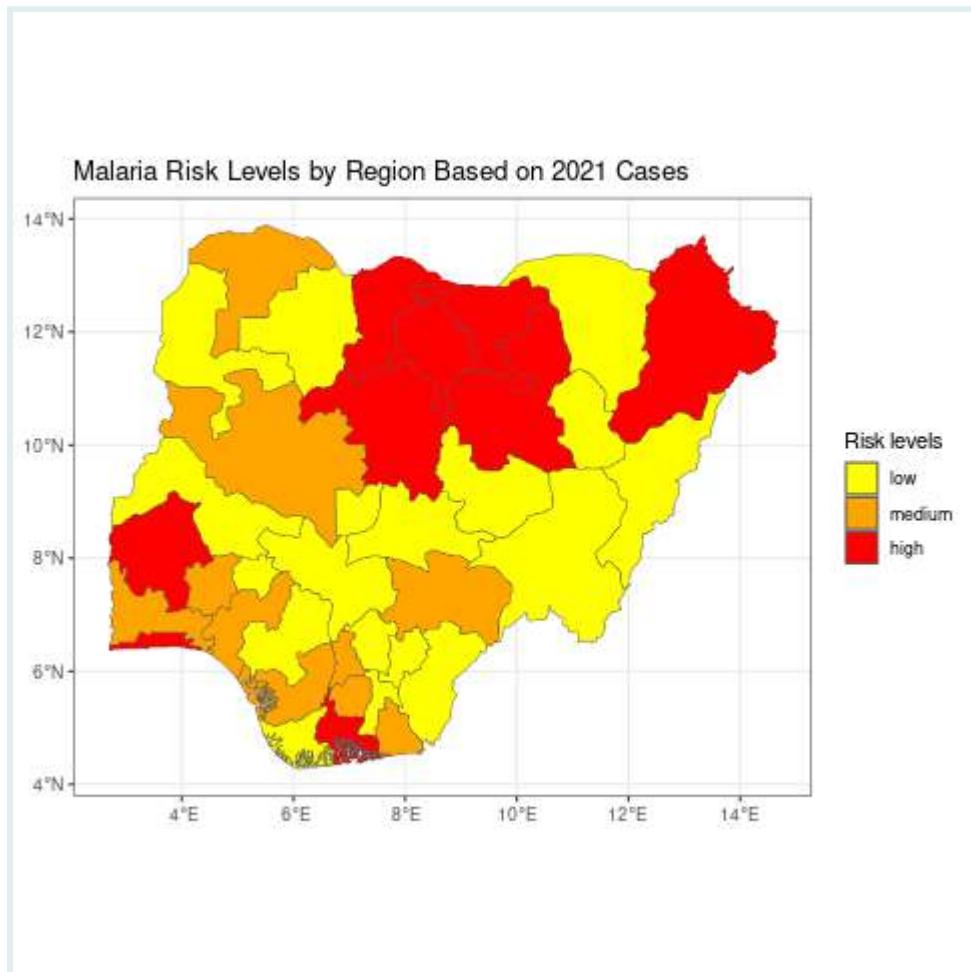
A choropleth map is a thematic map in which geographic regions are shaded or patterned in proportion to the value of a variable being represented. This variable can be an epidemiological indicator such as disease prevalence or mortality rate. Choropleth maps are particularly useful for visualizing spatial patterns and variations across different regions.

The essential components of a choropleth map include:

Geographic Regions: These are the areas that will be represented on the map, such as countries, states, districts, or any other geographic divisions.

Data Values: These are the values associated with each geographic region that will be represented on the map, including population density, prevalence, incidence, mortality rate, etc.

Color Scale: This is the range of colors used to represent the different data values. Typically, a gradient of colors is used, with lighter colors representing lower values and darker colors representing higher values.



Choropleth maps offer:

SIDE NOTE



- Clear visuals highlighting spatial data trends.
- Intuitive designs comprehensible without expertise.

However, they come with challenges:

- Data classification and color choices can skew appearances.
- Larger regions might dominate, causing visual bias.

In the following section, you will learn how to create a choropleth map using the `{ggplot2}` package in R.

Packages

This code snippet below shows how to load necessary packages using `p_load()` from the `{pacman}` package. If not installed, it installs the package before loading.

```
# Load necessary packages

# Use pacman to load multiple packages; it will install them if they're not
# already installed
pacman::p_load(tidyverse,      # for data wrangling and visualization
               here,        # for project-relative file paths
               sf)          # for spatial data

# Disable scientific notation for clearer numeric displays
options(scipen=10000)
```

Data Preparation

Before creating a choropleth map, it is essential to prepare the data. The data should contain the geographic regions and the values you want to visualize.

In this section, you will go through the data preparation process, which includes the following steps:

- 1. Import polygon data:** This is the geographical data that contains the boundaries of each region that you want to include in your map.
- 2. Import attribute data:** This is the data that contains the values you want to visualize on the map, such as disease prevalence, population density, etc.
- 3. Join the polygon and attribute data:** This step involves merging the polygon data with the attribute data based on a common identifier, such as the administrative level or region name. This will create a single dataset that contains both the geographic boundaries and the corresponding data values.

Now, let's go through each step in detail!

Step 1: Import polygon data

Polygons are closed shapes with three or more sides. In spatial data, polygons are used to represent areas such as the boundary of a city, lake, or any land use type. They are essential in geographical information systems (GIS) for tasks like mapping, spatial analysis, and land cover classification.

SIDE NOTE



Shapefiles are a common format for storing spatial data. They consist of at least three files with extensions `.shp` (shape), `.shx` (index), and

SIDE NOTE

.dbf (attribute data).

In R, you can load shapefiles using the `sf` package. In our lesson today, we will be working with malaria data sourced from the Nigeria Epi Review (2022).

```
# Reading the shapefile
nga_adm1 <-
  sf::st_read(here::here("data/raw/NGA_adm_shapefile/NGA_adm1.shp"))
```

```
## Reading layer `NGA_adm1` from data source
## `C:\GitHub\graph_courses\epi_reports_staging\EPIREP_EN_choropleth_maps\data\raw\NGA_adm_
##   using driver `ESRI Shapefile'
## Simple feature collection with 38 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 2.668431 ymin: 4.270418 xmax: 14.67642 ymax: 13.89201
## Geodetic CRS:  WGS 84
```

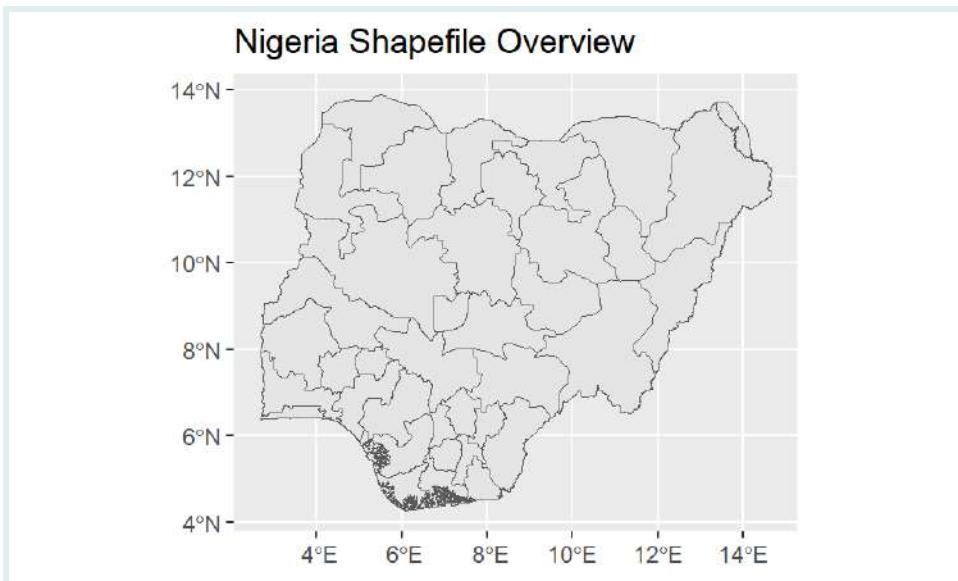
REMINDER

The important elements of any ggplot layer are the aesthetic mappings `aes(x, y, ...)` which tell ggplot where to place the plot objects.

We can imagine a map just like a graph with all features mapping to an x and y axis. All geometry (`geom_`) types in ggplot feature some kind of aesthetic mapping, and these can either be declared at the plot level, e.g., `ggplot(data, aes(x = variable1, y = variable2))`, or at the individual layer level, e.g., `geom_point(aes(color = variable3))`.

Bellow you can see that `geom_sf()` is used to trace the boundaries of the different states of Nigeria. Similarly, different layers can be added on top.

```
ggplot() +
  geom_sf(data = nga_adm1) +
  labs(title = "Nigeria Shapefile Overview")
```



Here, we first read the Nigeria shapefile using `sf::st_read()` and then plot it using `ggplot`. The `geom_sf()` function is used to render the spatial data, and `labs()` is used to add a title to the plot.



PRO TIP Since the shapefile was loaded using `sf::st_read()`, we don't need to specify the axis names. In fact, the coordinates are stored as a multipolygon object in the `geometry` variable, and `geom_sf()` automatically recognizes them.

For users who are interested in accessing subnational boundary data in R, there are a couple of packages available on GitHub which make this process straightforward:

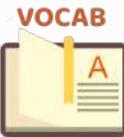
SIDE NOTE



- The `{rgeoboundaries}` (<https://github.com/wmgeolab/rgeoboundaries>) fetches data from the Geoboundaries, an open database of political administrative boundaries.
- The `{rnatural-earth}` (<https://github.com/ropensci/rnatural-earth>) fetches data from Natural Earth, a public domain map dataset.

Step 2: Import Attribute Data

VOCAB



In the context of choropleth maps, the “attribute data” refers to the quantitative or qualitative information that will be used to shade or color the various geographical areas on the map. For instance, if you have a map of a country’s regions and wish to shade each based on its population, the population data would be considered the attribute data.

As highlighted above, we will be using the number of malaria cases reported in Nigeria for the years 2000, 2006, 2010, 2015, and 2021.

```
# Reading the attribute data
malaria_cases <- read_csv(here::here("data/malaria.csv"))
malaria_cases
```

Step 3: Checking the Joined Data

It is essential to check and validate the joined data to ensure that the merge will be successful and the data will be accurate.

```
# Validate the joined data
all.equal(unique(nga_adm1$NAME_1), unique(malaria_cases$state_name))
```

```
## [1] "Lengths (38, 37) differ (string compare on first 37)"
## [2] "2 string mismatches"
```

```
# Identify the mismatches
setdiff(unique(nga_adm1$NAME_1), unique(malaria_cases$state_name))
```

```
## [1] "Water body"
```

In the code snippet provided, we’re comparing the unique region names between the `nga_adm1` and `malaria_cases` datasets using the `all.equal()` function. This is to ensure that all the regions in the shapefile align with their counterparts in the attribute data.

If the region names are identical, `all.equal()` will return `TRUE`. However, if there are discrepancies, it will detail the differences between the two sets of region names.

It’s noted that the “Water body” is present in the shapefile but not in the `malaria_cases`. Since “Water body” isn’t a proper region, it should be removed prior to joining the datasets. We can do this with `filter()`.

```
nga_adm1 <- filter(nga_adm1, NAME_1 != "Water body")
```

Step 4: Join Data by Administrative Levels

Now we'll get the data we want to plot on the map. The important thing is that the region names are the same in the shapefile as in your data you want to plot, because that will be necessary for them to merge properly.

Before joining the two datasets, we need to define the join key (`by =`).

REMINDER



Remember to review the lesson on joining tables for more details on how to merge data.frames in R if you are not familiar with joining.

```
# Add population data and calculate cases per 10K population
malaria <- malaria_cases %>%
  left_join(nga_adm1,
            by = c("state_name" = "NAME_1")) %>%
  st_as_sf() # convert to shapefile

# Select the most important columns
malaria2 <- malaria%>%
  select(state_name, cases_2000, cases_2006, cases_2010, cases_2015,
         cases_2021, geometry)
```

In this step, we merge the shapefile data `nga_adm1` with `malaria_cases` data using `left_join()` from the `{dplyr}` package. The `by` argument is used to specify the common column on which to merge the datasets.

Finally, we convert the merged data to a shapefile using `st_as_sf()`.

We can keep only the important variables that will be used in the construction of the plots by using `select()`.



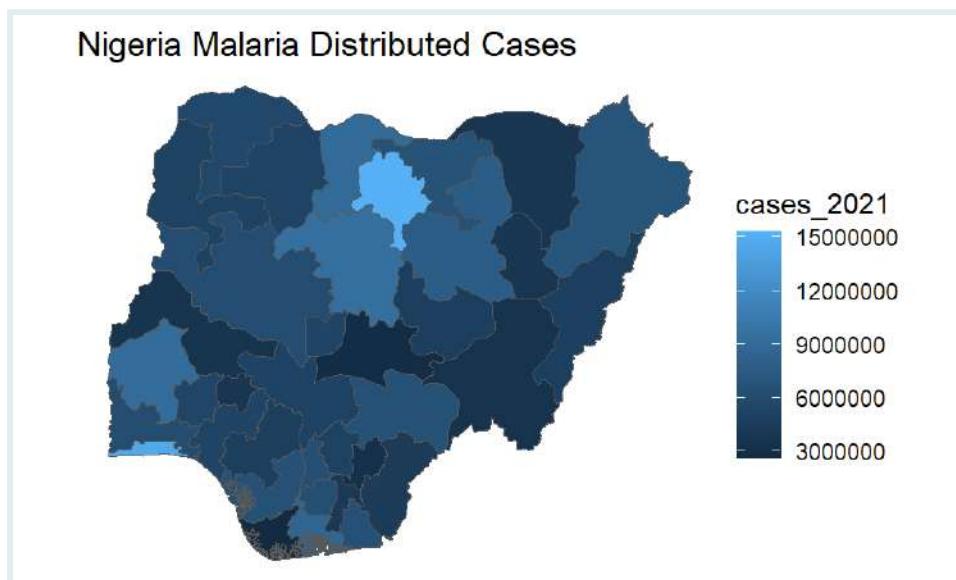
In this part of the lesson, we learned about the importance of administrative levels in spatial data and how to join spatial and attribute data by administrative levels using the `{dplyr}` package in R and how to check and validate the joined data.

Creating a Beautiful Choropleth Map with {ggplot2}

Using the Fill Variable (i.e., Attribute Variable)

To display the number of cases, for 2021 for example, we need to fill the polygons drawn with `geom_sf()` using the `fill` variable. This is very straightforward as it follows the same logic as the syntax of `{ggplot2}` package.

```
ggplot(data=malaria2) +  
  geom_sf(aes(fill=cases_2021)) +  # set fill to vary by case count variable  
  labs(title = "Nigeria Malaria Distributed Cases") +  
  theme_void()
```

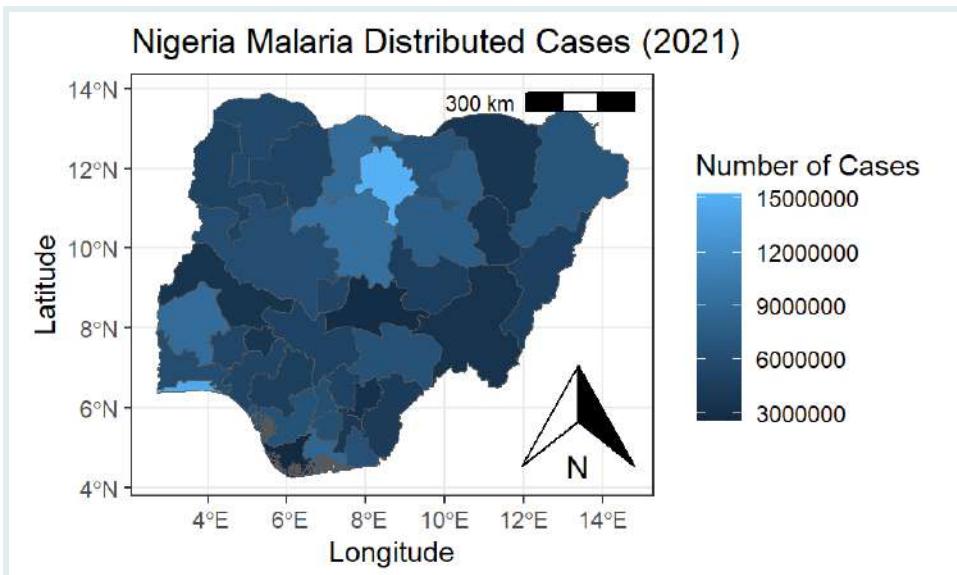


We create a basic choropleth map using `{ggplot2}`. The `fill` aesthetic is set to vary by the `cases` variable, which colors the regions based on the number of Malaria cases.

Customizing the Map

We can customize the map by adding titles to the axes and the legend, adding a north arrow and scale bar using `ggspatial::annotation_north_arrow()` and `ggspatial::annotation_scale()`, and changing the theme to `theme_bw()`.

```
ggplot(data=malaria2) +  
  geom_sf(aes(fill=cases_2021)) +  # set fill to vary by case count variable  
  labs(title = "Nigeria Malaria Distributed Cases (2021)",  
       fill = "Number of Cases") +  
  xlab("Longitude") +  
  ylab("Latitude") +  
  ggspatial::annotation_north_arrow(location = "br") +  
  ggspatial::annotation_scale(location = "tr") +  
  theme_bw()
```



In this section, we first crafted a basic choropleth map using `{ggplot2}` to visualize regional malaria cases in Nigeria for 2021. Then, we enhanced the map by adding **titles**, **axis-labels**, a **north arrow**, a **scale bar**, and applying a monochrome **theme**.

Q: Construct your beautiful choropleth map

Construct a choropleth map to display the distribution of Malaria cases in 2015, using the `cases_2015` column from the `malaria2` dataset. You can elevate your map's design and clarity by incorporating titles, axis labels, and any other pertinent labels.

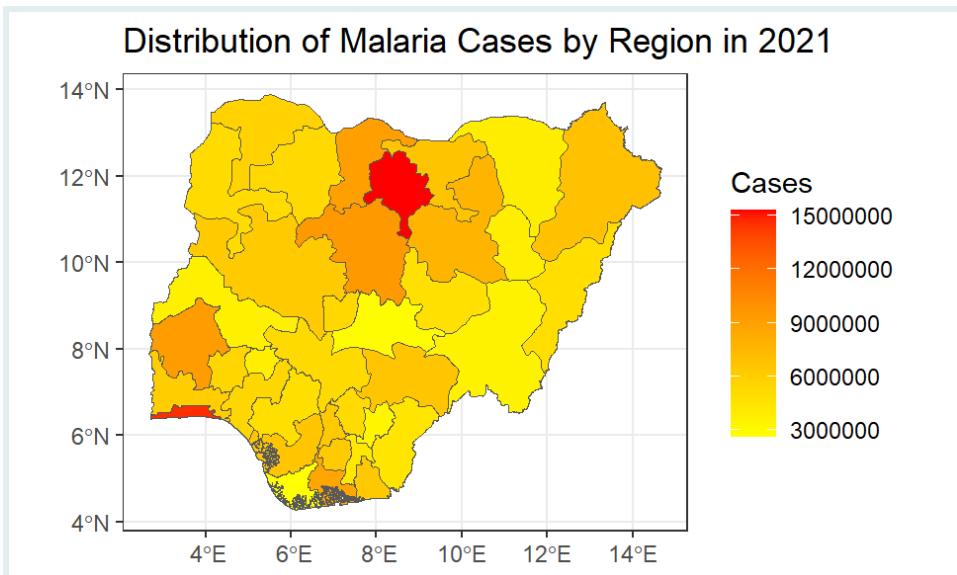
Color Scaling

Color Scaling for Continuous Attributes

The `scale_fill_continuous()` function from the `{ggplot2}` package in R is used to apply continuous color scaling to a choropleth map.

We can customize the color palette used for continuous color scaling by specifying the `low` and `high` parameters in the `scale_fill_continuous()` function.

```
# Create a ggplot object
ggplot(data = malaria2) +
  geom_sf(aes(fill = cases_2015)) +
  scale_fill_continuous(low = "yellow", high = "red", name = "Cases") + #
    apply continuous color scaling
  ggtitle("Distribution of Malaria Cases by Region in 2021") + # Add the
    corrected title here
  theme_bw()
```



In this section, we learned how to apply continuous color scaling to a choropleth map using the `scale_fill_continuous()` function from the `{ggplot2}` package in R and how to customize the color palette.

Color Scaling for Discrete Attributes

REMINDER



Discrete data is a type of quantitative data that can only take specific, separate values. It is often the result of counting objects or events. Discrete data is important in choropleth maps as it allows us to represent the count or quantity of objects or events in different regions.

The `scale_fill_brewer()` function from the `{ggplot2}` package in R is used to apply discrete color scaling to a choropleth map.

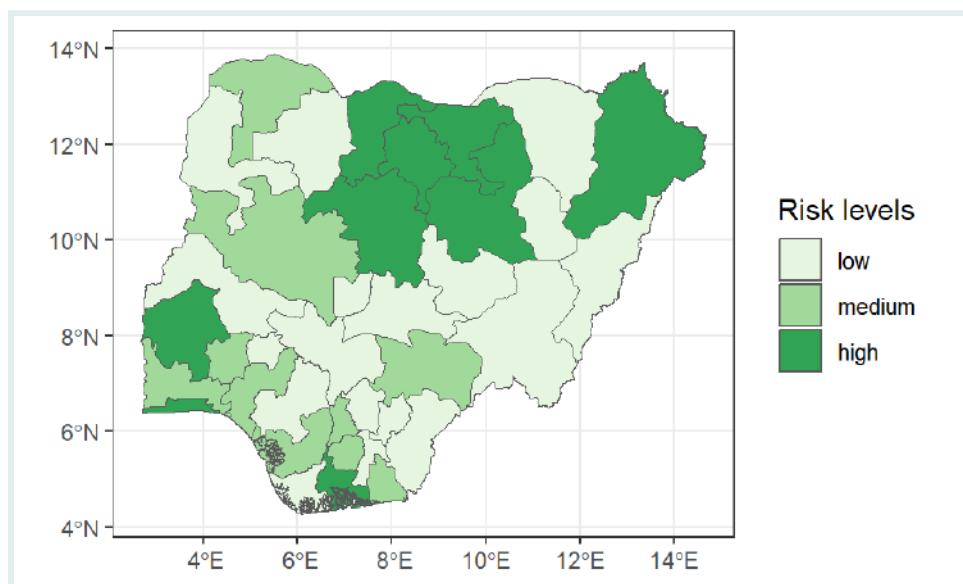
Before applying the discrete color scaling, we will need to create a new discrete column, which could be a risk level based on the number of cases. For this, we can use `mutate()` combined with `case_when()`.

```

# Transforming the data: Creating a new 'risk' column based on the number of
# cases in 2021
malaria3 <- malaria2 %>%
  mutate(risk = case_when(cases_2021 < quantile(cases_2021, 0.5) ~ 'low',
                          cases_2021 > quantile(cases_2021, 0.75) ~ 'high',
                          TRUE ~ 'medium'))

# Visualizing the data
ggplot(data = malaria3) +
  geom_sf(aes(fill = fct_reorder(risk, cases_2021))) + # The risk levels are
  # reordered based on the number of cases
  scale_fill_brewer(palette = "Set4", "Risk levels") +
  theme_bw()

```



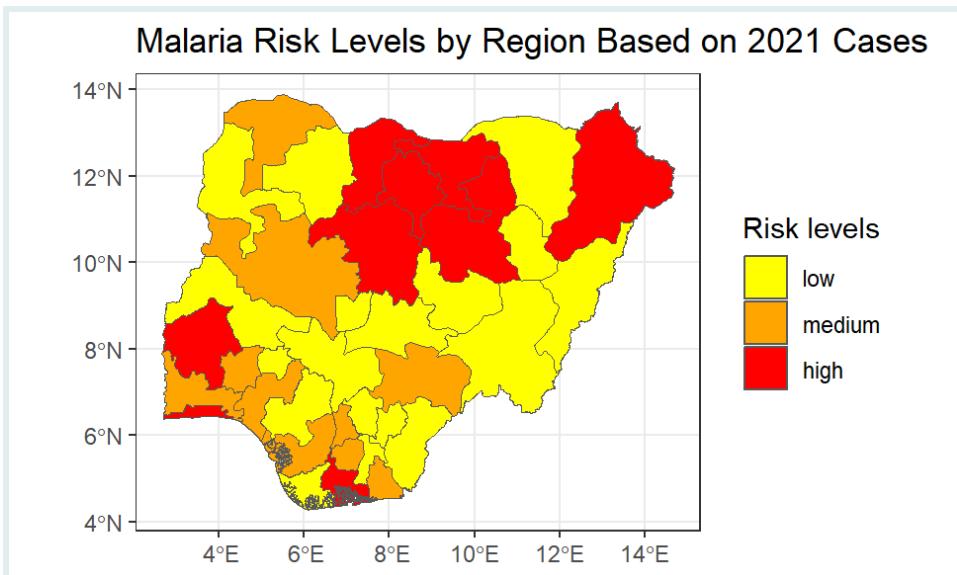
You can also create a custom color palette for discrete variables.

```

custom_palette <- c("yellow", "orange", "red") # create a custom color palette
# manually

# Apply custom color palette
ggplot(data = malaria3) + # Create a ggplot object
  geom_sf(aes(fill = fct_reorder(risk, cases_2021))) + # reorder the risk
  # labels according to the number of cases
  scale_fill_manual(values = custom_palette, name = "Risk levels") +
  coord_sf(expand = TRUE) +
  ggtitle("Malaria Risk Levels by Region Based on 2021 Cases") + # Add title
  # here
  theme_bw()

```



RECAP



In this section, we learned how to apply discrete color scaling to a choropleth map using the `scale_fill_brewer()` function from the `{ggplot2}` package and how to customize the color palette.

PRACTICE



Q: Create a your own color palette

Create a your own color palette distinct from the initial one, and display the Malaria cases across Nigeria for 2000 using this custom color palette. Don't forget to incorporate additional aesthetic enhancements.

Facet Wrap vs. Grid

- `facet_wrap()`: This function wraps a 1D sequence of panels into 2D. This is useful when you have a single variable with many levels and want to arrange the plots in a more space-efficient manner.
- `facet_grid()`: This function creates a matrix of panels defined by row and column faceting variables. It is most useful when you have two discrete variables, and all combinations of the variables exist in the data.

PRO TIP



- `facet_wrap()` is used for single variable faceting, while `facet_grid()` is used for two-variable faceting.



- `facet_wrap()` arranges the panels in a 2D grid, while `facet_grid()` arranges them in a matrix.

Creating Small Multiples using `facet_wrap()`

We can use the `facet_wrap()` function to create small multiples based on a given variable. In the following example, we can use the `year` variable.



Remember that color scaling can be adjusted by `scale_fill_continuous()` if the variable is continuous or by `scale_fill_brewer()` if the variable is discrete.

Here, we need to make a slight adjustment to the structure of our dataset. Since the cases are presented by year and each year is represented in a separate column, we'll use the `pivot_longer()` function to consolidate them into a single column, along with an identifier key.

```
malaria3_longer <- malaria3 %>%
  pivot_longer(cols = `cases_2000`:`cases_2021`, names_to = "year", values_to =
  = "cases")
```

It seems that the variable `year` in our dataset has a prefix `cases_` that we would like to remove for aesthetic reasons before plotting.

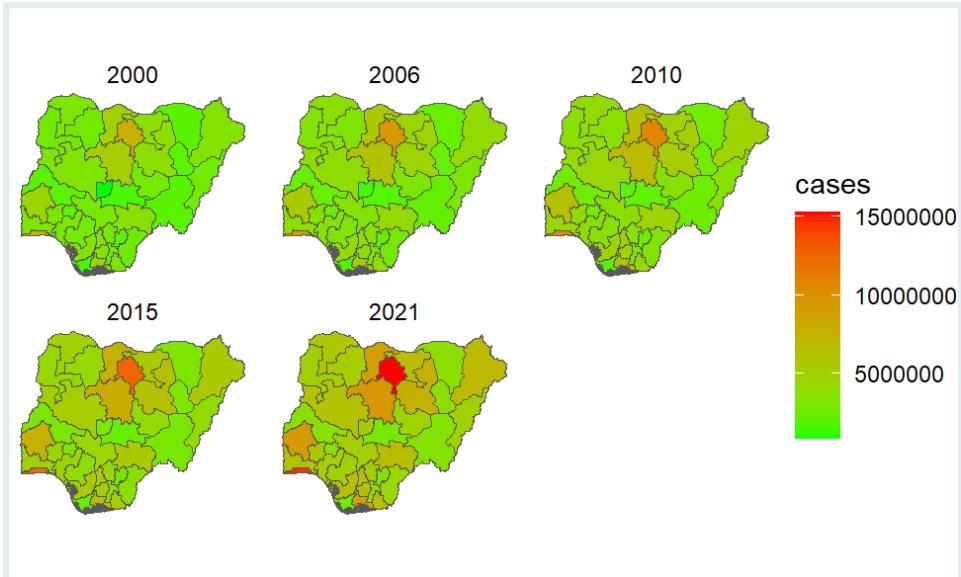
Here's how we can do it using the `str_replace()` function from `{stringr}` package, which is part of the tidyverse:

```
# Recode the 'year' variable by removing the 'cases_' prefix
malaria3_longer$year <- str_replace(malaria3_longer$year, "cases_", "")
```

After executing the code above, the `year` variable in our `malaria3_longer` dataset will no longer have the `cases_` prefix, making it cleaner for visualization purposes.

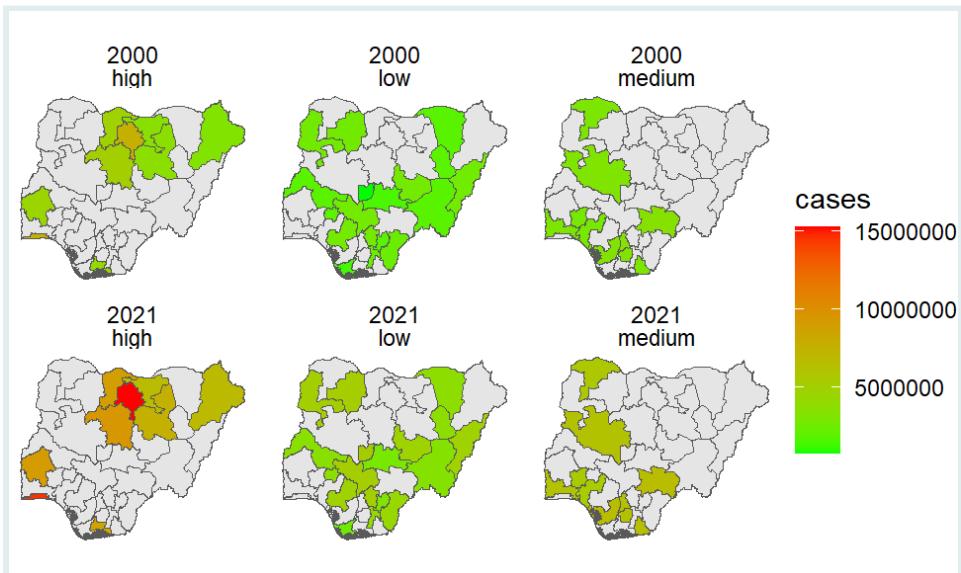
Now we want to create Small Multiples using `facet_wrap()`.

```
# Create ggplot object using facet_wrap
ggplot(data = malaria3_longer) +
  geom_sf(aes(fill = cases)) +
  facet_wrap(~ year) +
  scale_fill_continuous(low = "green", high = "red") + # apply continuous
  color scaling
  theme_void()
```



We can add another variable to the `facet_wrap()`. Here we add `risk`, but we will use only 2000 and 2021 cases for demonstration purposes:

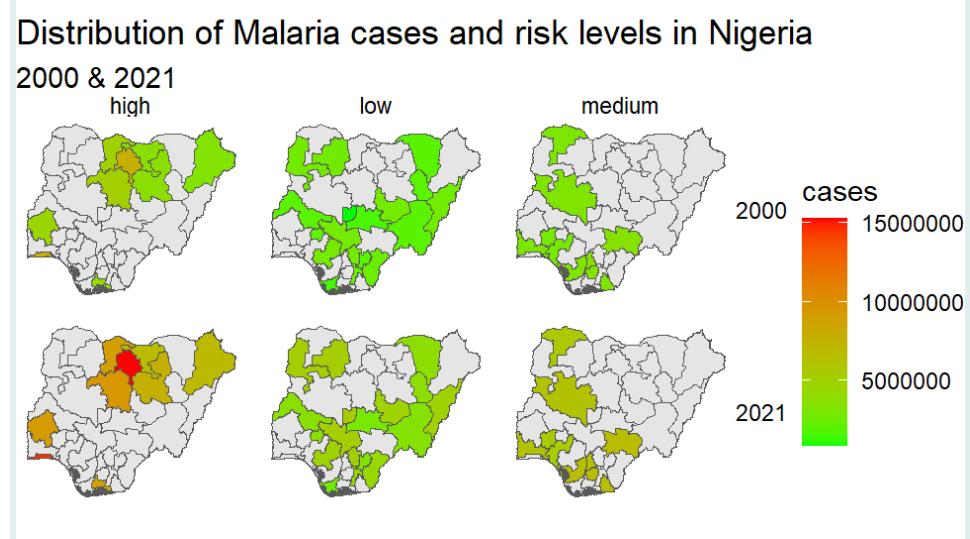
```
ggplot() +
  geom_sf(data = nga_adm1) +
  geom_sf(aes(fill = cases), data = filter(malaria3_longer, year %in%
    c("2000", "2021"))) +
  facet_wrap(year ~ risk) +
  coord_sf(expand = TRUE) +
  scale_fill_continuous(low = "green", high = "red") +
  theme_void()
```



In the last plot bellow, you will notice that we used a continuous fill variable, but we also used a `facet_wrap` on two other discrete variables.

Now, we will use `facet_grid()` to create a grid of plots. For each year (2000 and 2021), separate plots are made for each risk level, giving an easy visual comparison across both years and risks.

```
ggplot() +  
  geom_sf(data = nga_adm1) + # need the  
  geom_sf(aes(fill = cases), data = filter(malaria3_longer, year %in%  
    c("2000", "2021")))+  
  facet_grid(year ~ risk)+  
  coord_sf(expand = TRUE)+  
  scale_fill_continuous(low = "green", high = "red") +  
  labs(title = "Distribution of Malaria cases and risk levels in Nigeria",  
    subtitle = "2000 & 2021") +  
  xlab("Longitude") +  
  ylab("Latitude") +  
  theme_void()
```



`facet_grid()` is ideal when you have two factors and you want to explore every combination. `facet_wrap()` is better when you have just one factor or when you have multiple factors but want a simpler layout.

Choosing between them often depends on the specific needs of your data and how you want to visualize the relationships.

CHALLENGE



Q: Analyze the distribution of malaria cases



CHALLENGE Your goal now is to analyze the distribution of malaria cases in Nigeria for the years 2000 and 2021. But you will need first to categorize the data into risk levels using the median (high/low), and then visualize this information on a map.



RECAP In this section, we learned the differences between `facet_wrap()` and `facet_grid()`, and how to create small multiples using both of these functions.

WRAP UP!

Today, we went deep into the world of choropleth maps, unlocking the power of visual geographical representation.

You've gained insights into:

- The essence and components of a choropleth map.
- The pros and cons of these maps.
- Data preparation essentials for choropleth visualization.
- Crafting maps with `{ggplot2}` and applying varied color palettes.
- Harnessing `facet_wrap()` and `facet_grid()` for intricate map designs.

Learning outcomes

Congratulations on completing this lesson! Let's recap the cognitive journey you've taken:

1. Knowledge & Understanding:

- Recognized the definition and components of a choropleth map.
- Comprehended the advantages and limitations of choropleth maps.

2. Application:

- Prepare data specifically for the creation of a choropleth map.
- Employed `{ggplot2}` in R to design a choropleth map.
- Implemented continuous and discrete color scaling techniques.

3. Analysis:

- Differentiated between continuous and discrete color scaling.

4. Synthesis:

- Integrated various components to create small multiples using `facet_wrap()` and `facet_grid()`.

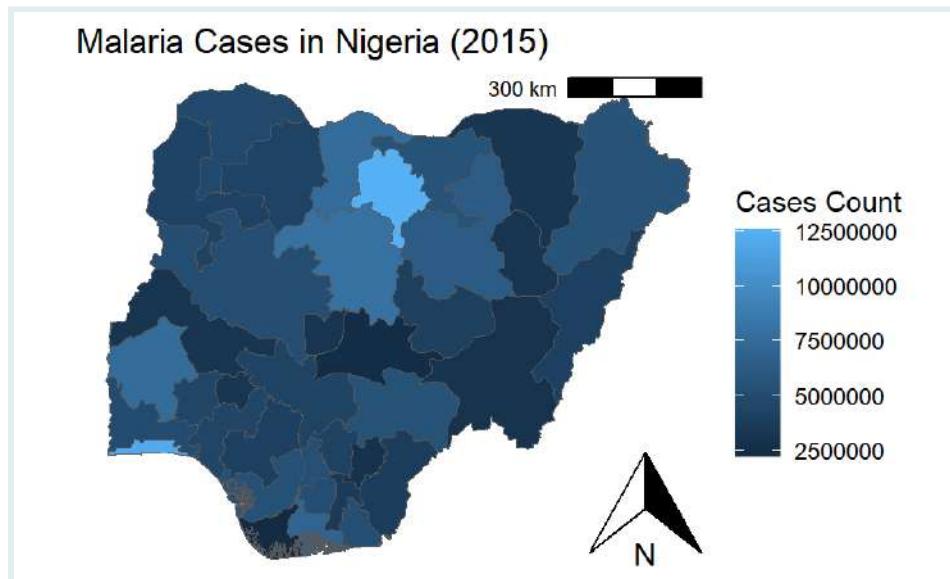
With these skills in hand, you're now equipped to innovate with different datasets and visualize them in impactful ways. Dive in, experiment, and enhance your map-making journey using `{ggplot2}` in R. Happy mapping!

Solutions

1. Construct your beautiful choropleth map

Construct a choropleth map to display the distribution of Malaria cases in 2019, using the `cases_2019` column from the `malaria2` dataset. You can elevate your map's design and clarity by incorporating titles, axis labels, and any other pertinent labels.

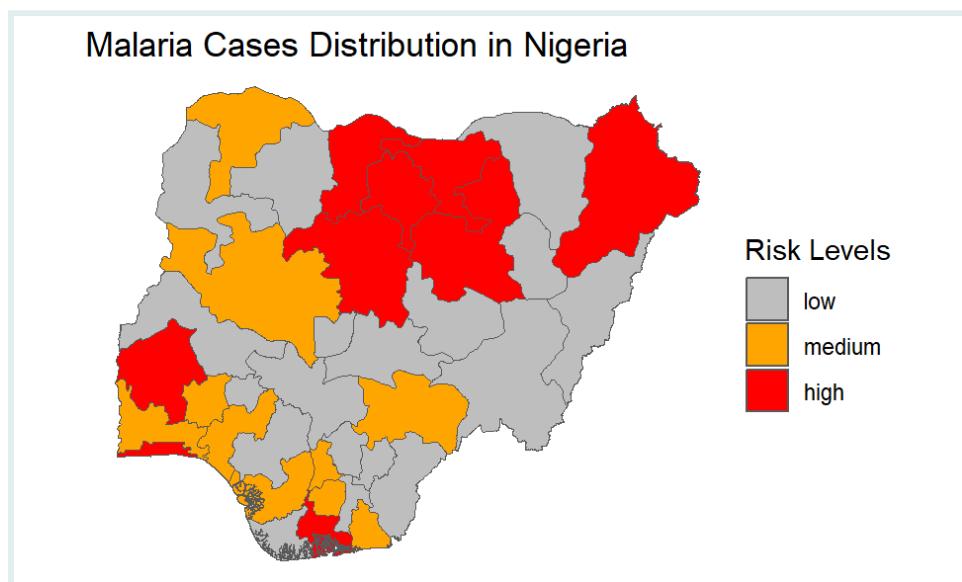
```
ggplot(data=malaria2) +  
  geom_sf(aes(fill=cases_2019)) +  
  labs(title = "Malaria Cases in Nigeria (2015)",  
       fill = "Cases Count",  
       x = "Longitude",  
       y = "Latitude") +  
  ggspatial::annotation_north_arrow(location = "br") +  
  ggspatial::annotation_scale(location = "tr") +  
  theme_void()
```



2. Create your own color palette

Create a your own color palette distinct from the initial one provided bellow, and display the Malaria cases across Nigeria for 2000 using this custom color palette. Don't foreget to incorporate additional aesthetic enhancements.

```
new_palette <- c("gray", "orange", "red") # Define a different set of colors
ggplot(data=malaria3) +
  geom_sf(aes(fill = fct_reorder(risk, cases_2000))) + # Reorder risk labels
  scale_fill_manual(values = new_palette, "Risk Levels") +
  labs(title = "Malaria Cases Distribution in Nigeria",
       fill = "Cases Count",
       x = "Longitude",
       y = "Latitude") +
  coord_sf(expand = TRUE) +
  theme_void()
```



3. Analyze the distribution of malaria cases

Your goal now is to analyze the distribution of malaria cases in Nigeria for the years 2000 and 2021. But you will need first to categorize the data into risk levels using the median (high/low), and then visualize this information on a map.

```

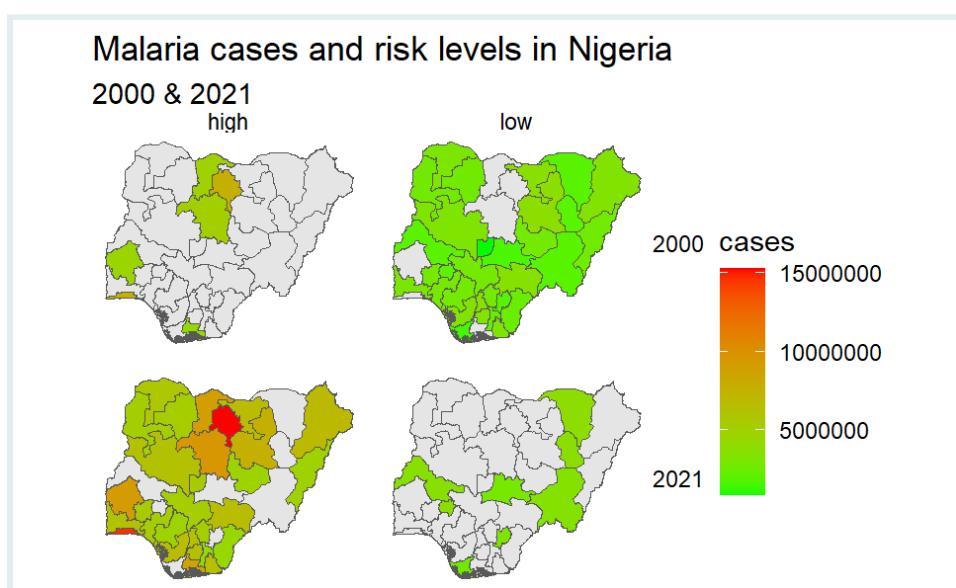
# Pivoting the data
malaria3_longer <- malaria2 %>%
  pivot_longer(cols = `cases_2000`:`cases_2021`, names_to = "year", values_to
  = "cases")

# Categorizing risk based on median
malaria3_longer %>%
  mutate(risk = case_when(
    cases <= median(cases) ~ 'low',
    cases > median(cases) ~ 'high'
  )) -> malaria3_longer2

# Cleaning up the year values
malaria3_longer2$year <- str_replace(malaria3_longer2$year, "cases_", "")

# Plotting the data
ggplot() +
  geom_sf(data = nga_adm1) +
  geom_sf(aes(fill = cases), data = filter(malaria3_longer2, year %in%
    c("2000", "2021"))) +
  facet_grid(year ~ risk) +
  coord_sf(expand = TRUE) +
  scale_fill_continuous(low = "green", high = "red") +
  labs(title = "Malaria cases and risk levels in Nigeria",
       subtitle = "2000 & 2021") +
  xlab("Longitude") +
  ylab("Latitude") +
  theme_void()

```



Contributors

The following team members contributed to this lesson:



IMAD EL BADISY

Data Science Education Officer
Deeply interested in health data



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

- Wickham, Hadley, Winston Chang, and Maintainer Hadley Wickham. “Package ‘ggplot2’.” *Create elegant data visualisations using the grammar of graphics.* Version 2, no. 1 (2016): 1-189.
- Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. *R for data science.* ” O'Reilly Media, Inc.”, 2023.
- Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. *Geocomputation with R.* CRC Press, 2019.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



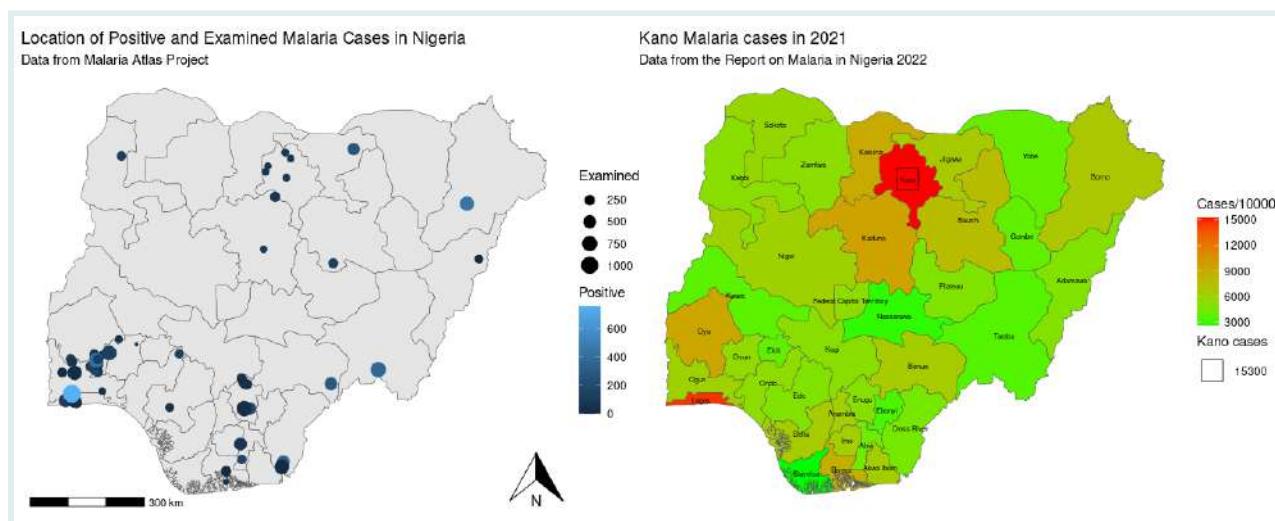
Enhancing Disease Maps with Labels in R

Introduction
Packages
Data Preparation
Building a Simple Choropleth Map
Adding Continuous Data Indicators to the Choropleth Map
Exploring Malaria Case Increase Rates Using Choropleth Maps
Labeling the Choropleth Map with State Names
Displaying Combined State Names and Increase Rates on the Choropleth Map
Highlighting a Specific Region on the Map while Preserving Context
Labeling Point Locations: Exploring Malaria Positive Rate and Incidence
WRAP UP!
Answer Key

Introduction

In the geospatial data visualization, maps are powerful storytelling tools. However, a map without clear annotations and labels is like a book without titles or chapter headings. While the story might still be there, it becomes significantly harder to understand, interpret, and appreciate.

In this lesson, we place a special emphasis on the importance of annotating and labeling maps. Proper annotation transforms a simple visualization into an informative guide, allowing viewers to quickly grasp complex spatial data. With precise labeling, areas of interest can be immediately recognized, facilitating a clearer comprehension of the data's narrative.



Learning objectives

Learning Objectives: Advanced Geospatial Visualization Techniques

By the end of this section, you should be able to:

- Incorporate continuous data indicators into choropleth maps for enhanced granularity.

- Effectively overlay state names onto choropleth maps, ensuring clarity and readability.
- Seamlessly integrate state names with increase rates on maps without compromising legibility.
- Apply techniques to accentuate specific regions on a map while retaining the overall context.
- Determine optimal point placement strategies and integrate them effectively into geospatial visualizations.

Upon mastering these objectives, you will have the tools and knowledge to create rich, detailed, and informative geospatial visualizations.

Packages

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(malariaAtlas,
  ggplot2,
  geodata,
  dplyr,
  tidyr,
  here,
  readr,
  sf,
  patchwork)

# Unable scientific notation
options(scipen=100000)
```

Data Preparation

Before diving into any visualization or analysis, it's essential to load and preprocess our data. This includes reading in datasets, merging related information, and filtering out unnecessary or irrelevant entries.

```
# Reading the geographical shapefile data
nga_adm1 <- sf::st_read(here::here("data/raw/NGA_adm_shapefile/NGA_adm1.shp"))

## Reading layer `NGA_adm1` from data source
##
`C:\GitHub\graph_courses\epi_reports_staging\EPIREP_EN_choropleth_maps_labeling\
```

```
data\raw\NGA_adm_shapefile\NGA_adm1.shp'
##   using driver `ESRI Shapefile'
## Simple feature collection with 38 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:  xmin: 2.668431 ymin: 4.270418 xmax: 14.67642 ymax: 13.89201
## Geodetic CRS:  WGS 84
```

The geographical shapefile data for Nigeria's administrative boundaries (like states or provinces) is read and stored in the `nga_adm1` object.

```
# Reading the attribute data related to malaria cases
malaria_cases <- read_csv(here::here("data/malaria.csv"))
```

This step loads data related to malaria cases in different states of Nigeria.

```
# Filtering out the 'Water body' entries from the geographical data
nga_adm1 <- filter(nga_adm1, NAME_1 != "Water body")

# Merging the geographical data with the malaria cases data
malaria <- malaria_cases %>%
  left_join(nga_adm1, by = c("state_name" = "NAME_1")) %>%
  st_as_sf()
```

Here, we're combining the geographical boundaries data with the malaria cases data using state names as a reference. This merged data is then converted into a format suitable for geospatial visualizations.

```
# Filtering down to essential columns for our analysis
malaria2 <- malaria %>%
  select(state_name, cases_2000, cases_2006, cases_2010, cases_2015,
         cases_2021, geometry)
```

We're narrowing down our dataset to specific columns, mainly the state names, the malaria cases from various years, and the geographical boundaries of these states (i.e. geometry).

```
# Reading in population data for different regions of Nigeria
population_nigeria <- read_csv(here::here("data/population_nigeria.csv"))
```

This step loads data indicating the population of different states or regions in Nigeria.

```
# Combining the population data with our malaria data
malaria3 <- malaria2 %>%
  left_join(population_nigeria, by = c("state_name")) %>%
  st_as_sf()
```

By merging the population data with our malaria cases data, we enrich our dataset. This combined data allows for more comprehensive visualizations and analyses, such as calculating incidence rates or assessing trends relative to population size.

To address aesthetic concerns with labeling a small area like the “Federal Capital Territory” on a map, we can modify the labels before plotting. We can do this by creating a new column for the modified state names or by directly changing the `state_name` column within the `malaria3` dataset. Given that the region in question has a small surface area and the full name may overflow its borders, here’s how you can adjust the name to “Capital” for better display:

```
# Modify the state names, changing "Federal Capital Territory" to "Capital"
malaria3$state_name <- ifelse(malaria3$state_name == "Federal Capital
Territory", "Capital", malaria3$state_name)
```

Building a Simple Choropleth Map

REMINDER

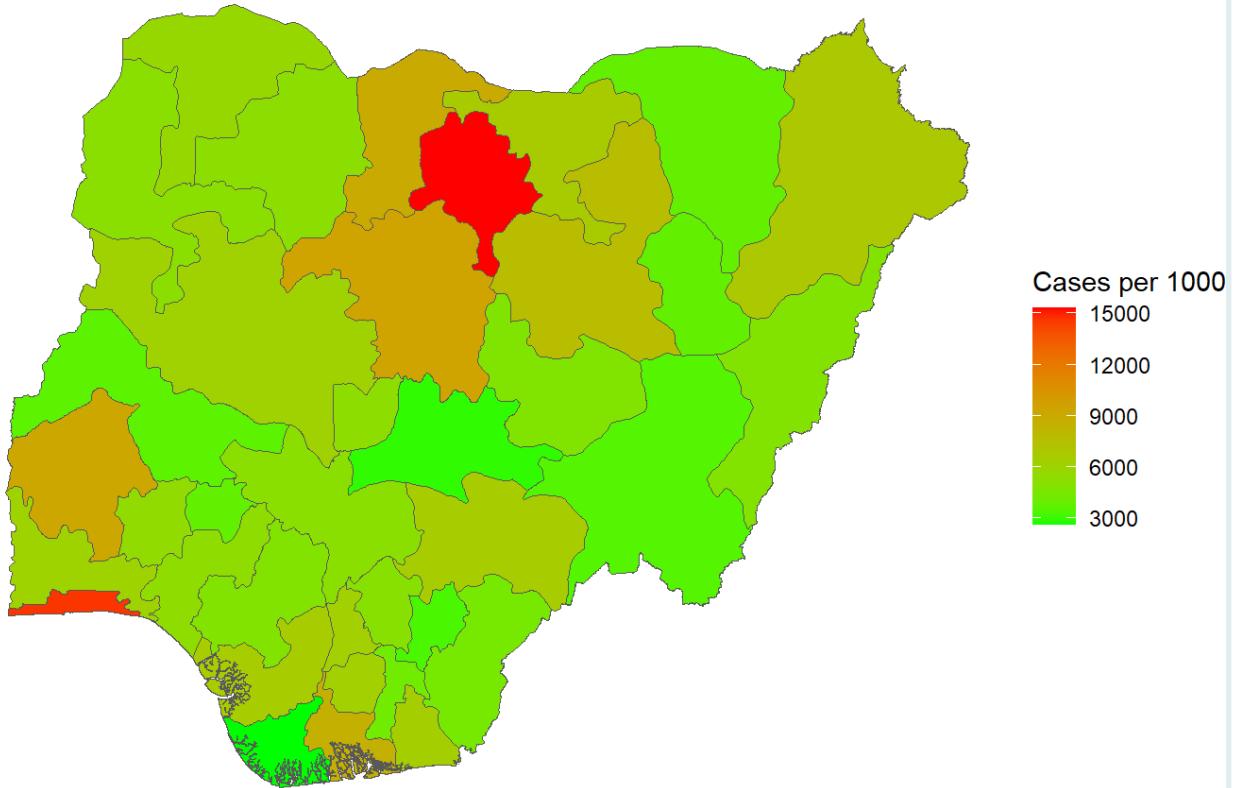


Choropleth maps are powerful visualization tools that display divided geographical areas shaded or patterned in proportion to the value of a variable.

In this example, we’ll be using a choropleth map to visualize the distribution of malaria cases across different regions of Nigeria for the year 2021.

```
# Constructing the choropleth map using ggplot2
ggplot(data=malaria3) +
  geom_sf(aes(fill=cases_2021/1000)) + # The fill color is determined by the
  # number of malaria cases in 2021, scaled per 1000
  labs(title = "Nigeria Malaria Distributed Cases in 2021", fill = "Cases per
    1000") + # Adding labels and title to the plot
  scale_fill_continuous(low = "green", high = "red") + # Using a continuous
  # color scale transitioning from green to red
  theme_void() # Using a minimal theme for better visualization of the map
```

Nigeria Malaria Distributed Cases in 2021



Here's a detailed explanation of the code:

- `ggplot(data=malaria3)`: Initiates a ggplot object using the `malaria3` dataset.
- `geom_sf(aes(fill=cases_2021/1000))`: Adds the geographical data from `malaria3` and fills each region based on the number of malaria cases in 2021, scaled down by a factor of 1000. This effectively represents the number of cases per thousand people.
- `labs(title = "Nigeria Malaria Distributed Cases in 2021", fill = "Cases per 1000")`: Specifies the title of the plot and the label for the color scale.
- `scale_fill_continuous(low = "green", high = "red")`: Applies a continuous color scale where regions with fewer cases are colored green and regions with more cases are colored red.
- `theme_void()`: Removes axis text, ticks, and other non-data ink to emphasize the map.

The resulting plot provides a clear view of how malaria cases are distributed across Nigeria in 2021, with the color intensity indicating the magnitude of cases in each region.

Q: Modify the provided choropleth map to visualize the distribution of malaria cases in Nigeria for the year 2015.

Instructions



1. Update the data mapping in the `geom_sf()` function to reflect malaria cases for the year 2015.
2. Adjust the title in the `labs()` function to indicate that the visualization pertains to 2015.
3. Change the color gradient in `scale_fill_continuous()` to transition from blue (`low cases`) to yellow (`high cases`).

Below a starter code:

```
# Constructing the choropleth map for 2015 using ggplot2
ggplot(data=malaria3) +
  geom_sf(aes(fill= _____)) + # Fill in the correct data
  # column for 2015
  labs(title = " _____", fill = "Cases per
       1000") + # Update the title appropriately
  scale_fill_continuous( _____) + # Modify the color scale
  theme_void()
```

Adding Continuous Data Indicators to the Choropleth Map

When analyzing disease data, it's often useful to look beyond raw case numbers and focus on rates, particularly incidence rates. The incidence rate provides a normalized measure that can account for differing population sizes across regions, making comparisons more meaningful.

Understanding Incidence

The incidence of a disease is a cornerstone of epidemiological research. It quantifies the number of new cases of a disease that occur within a specific time frame, typically a year, relative to a population at risk.

Mathematically, it's given by:

$$\text{Incidence} = \frac{\text{Number of new cases during the time period}}{\text{Population at risk at the start of the period}}$$

In this context, we're looking at the incidence of malaria in different states of Nigeria for the year 2021. Specifically, we'll compute the incidence rate by dividing the number of new malaria cases in 2021 by the population of each state from the last available census in 2019.

The following R code accomplishes this and visualizes the data:

```
# Visualizing Malaria Incidence in 2021 using a Choropleth Map
ggplot(data = malaria3) +
  # Filling each state with a color representing the incidence rate in 2021.
  geom_sf(aes(fill = round(cases_2021/population_2019, 2))) +
  # Add the name of each state to its centroid
  geom_sf_text(aes(label = str_wrap(state_name, 1)), size = 2) +
  # Adding title and legend title.
  labs(title = "Nigeria Malaria Incidence in 2021", fill = "Incidence") +
  # Using a continuous color scale from green (low incidence) to red (high
  # incidence).
  scale_fill_continuous(low = "green", high = "red") +
  # Using a minimalist theme for clearer visualization.
  theme_void()
```

Nigeria Malaria Incidence in 2021



Here's a brief explanation of the visualization:

- The `geom_sf()` function creates a choropleth map where each state's color represents its malaria incidence rate in 2021.
- The `geom_sf_text()` function labels each state with its specific incidence rate, positioning automatically each label at the state's centroid.
- The `str_wrap()` function is part of the `{stringr}` package and is being used here to wrap the text of `state_name`. Since the second argument to `str_wrap()` is 1, this would cause the state names to be wrapped after every character, to avoid overlapping.
- The color scale, transitioning from green to red, visually emphasizes regions with higher incidence rates.

This visualization offers an immediate grasp of the malaria situation in Nigeria, revealing areas of high incidence that might need more focused public health interventions.

Exploring Malaria Case Increase Rates Using Choropleth Maps

Understanding the change in the number of disease cases over time can provide insights into the effectiveness of interventions, the progression of the disease, and areas where increased resources might be needed. In this context, we're looking to visualize the percentage increase in malaria cases from 2015 to 2021 across different states in Nigeria.

Computing the Increase Rate

The increase rate for each state is computed as:

$$\text{Increase Rate} = \left(\frac{\text{Cases in 2021} - \text{Cases in 2015}}{\text{Cases in 2015}} \right) \times 100\%$$

This formula provides the percentage growth (or decrease) in malaria cases from 2015 to 2021.

Visualization of Increase Rates

Let's delve into the code that accomplishes this visualization:

```

# Calculate the increase rate for each state
malaria3 %>%
  mutate(increase_rate = round(((cases_2021 - cases_2015) / cases_2015) *
    100)) -> malaria3

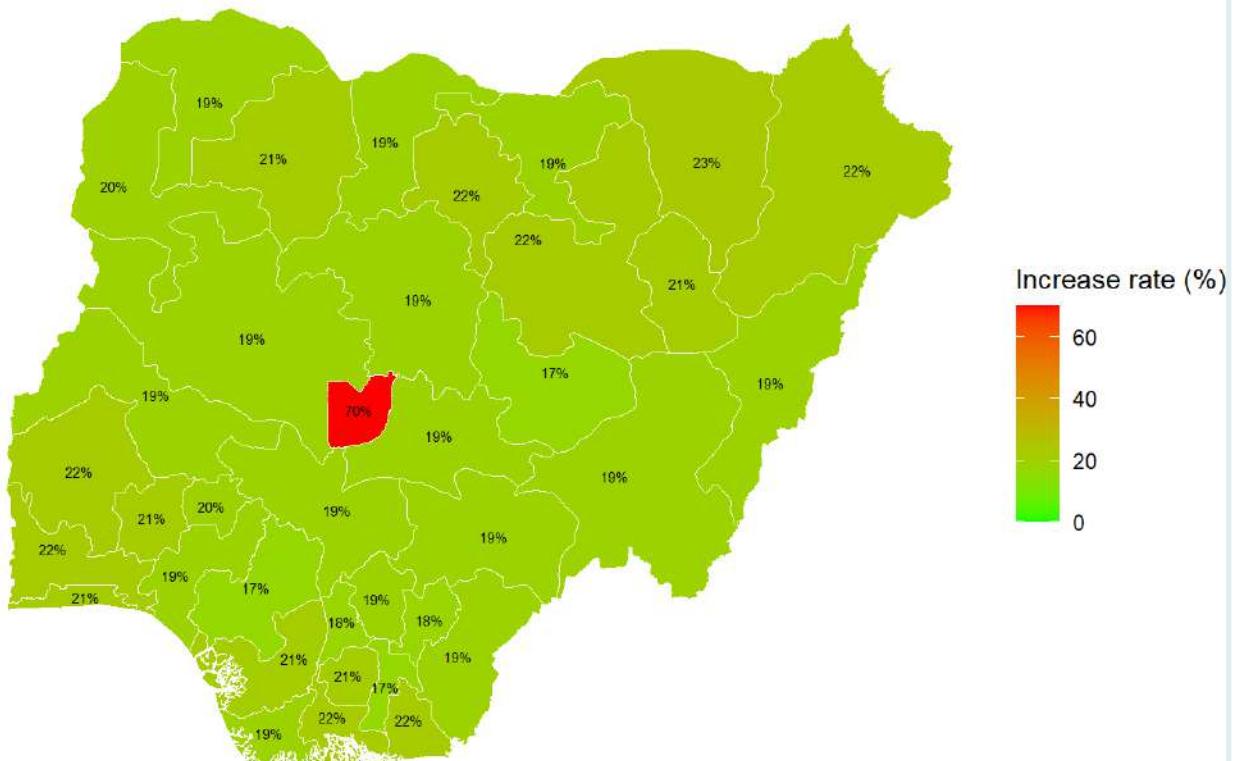
# Visualizing the increase rates using a choropleth map
ggplot(data = malaria3) +
  # Coloring each state based on its increase rate
  geom_sf(aes(fill = increase_rate), color="white", size = 0.2) +
  # Using a continuous color scale to represent increase rates, transitioning
  # from green to red
  scale_fill_continuous(name="Increase rate (%)", limits=c(0,70), low =
    "green", high = "red", breaks=c(0, 20, 40, 60))+

  # Labeling each state with its increase rate percentage
  geom_sf_text(aes(label = str_wrap(paste0(increase_rate, "%"), 1))), size =
    2)+

  # Adding a title to the plot
  labs(title = "Nigeria Malaria Increase rate in 2021 compared with 2015")+
  theme_void()

```

Nigeria Malaria Increase rate in 2021 compared with 2015



In this visualization:

- The `geom_sf()` function creates the choropleth map where each state's color intensity represents its malaria increase rate.

- `scale_fill_continuous()` sets the color scale for the increase rates, making areas of higher increase more prominent.
- `geom_sf_text()` adds labels to each state.
- `str_wrap()` ensures that the text does not wrap (since it's set to wrap at 1 character).

The resulting map allows us to quickly identify regions with significant growth in malaria cases over the selected period.

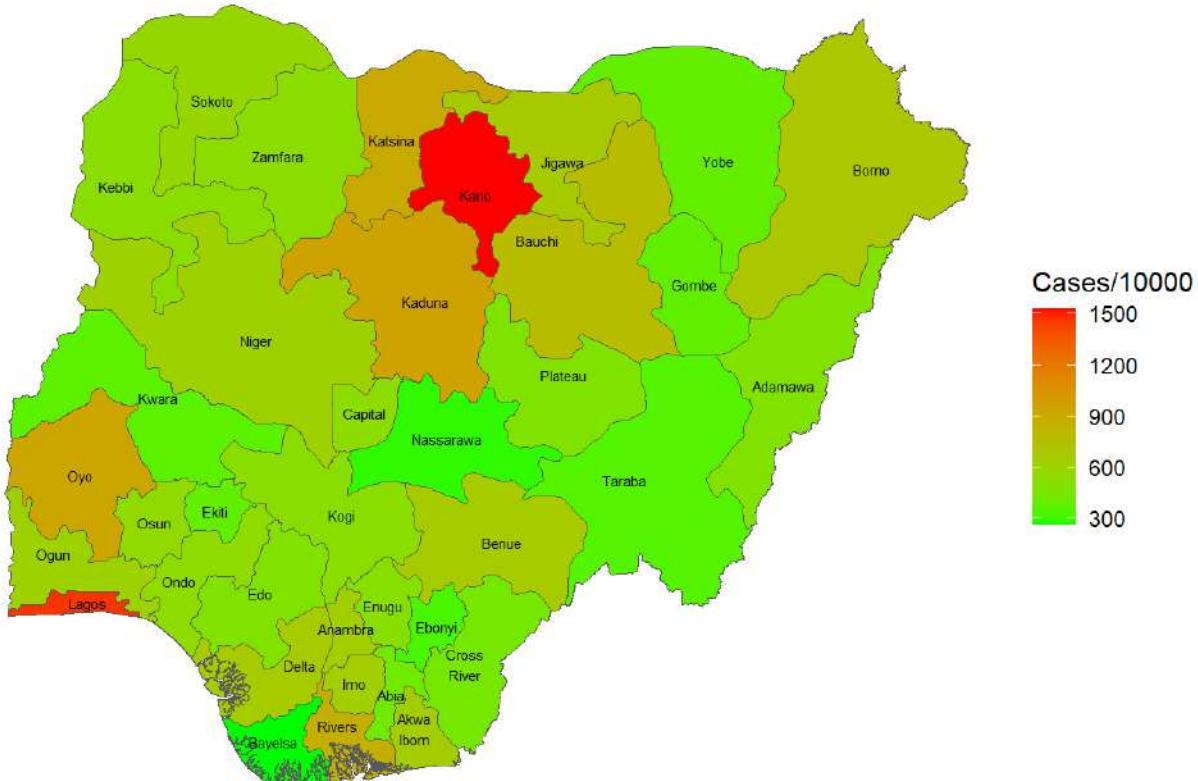
Through this visualization, you can pinpoint regions where malaria is on the rise and potentially allocate resources more effectively.

Labeling the Choropleth Map with State Names

In a choropleth map, while color gradients offer a visual cue to understand the distribution of a variable across regions, adding labels can significantly enhance the clarity of the visualization. This is especially true when audiences might not be familiar with all geographical boundaries shown. In the case of our malaria dataset, adding state names to the map makes the data more accessible and understandable.

```
# Constructing a choropleth map with state names
ggplot(data = malaria3) +
  # Fill each state based on the number of malaria cases in 2021, scaled per
  # 10,000
  geom_sf(aes(fill = cases_2021/10000)) +
  # Add the name of each state to its centroid
  geom_sf_text(aes(label = str_wrap(state_name, 1))), size = 2) +
  # Add titles and labels
  labs(title = "Nigeria Malaria cases in 2021", fill = "Cases/10000") +
  # Use a continuous color scale from green (low case numbers) to red (high
  # case numbers)
  scale_fill_continuous(low = "green", high = "red") +
  theme_void()
```

Nigeria Malaria cases in 2021



Here's a detailed explanation of the visualization:

- `geom_sf(aes(fill = cases_2021/10000))`: This creates the choropleth map where each state's color represents the number of malaria cases in 2021, scaled down by a factor of 10,000.
- `labs(title = "Nigeria Malaria cases in 2021", fill = "Cases/10000")`: This function adds a title to the plot and a label to the color scale.
- `scale_fill_continuous(low = "green", high = "red")`: This sets the color scale for the map, transitioning from green for states with fewer cases to red for those with more cases.

The resulting visualization is a clear and informative map of malaria cases across Nigeria in 2021, with each state labeled for easy reference.

Displaying Combined State Names and Increase Rates on the Choropleth Map

Visualizations can convey a wealth of information when they incorporate multiple data points in an intuitive manner. By pairing state names with their corresponding increase rates, we can provide a richer, more detailed view of the data without overwhelming the audience.

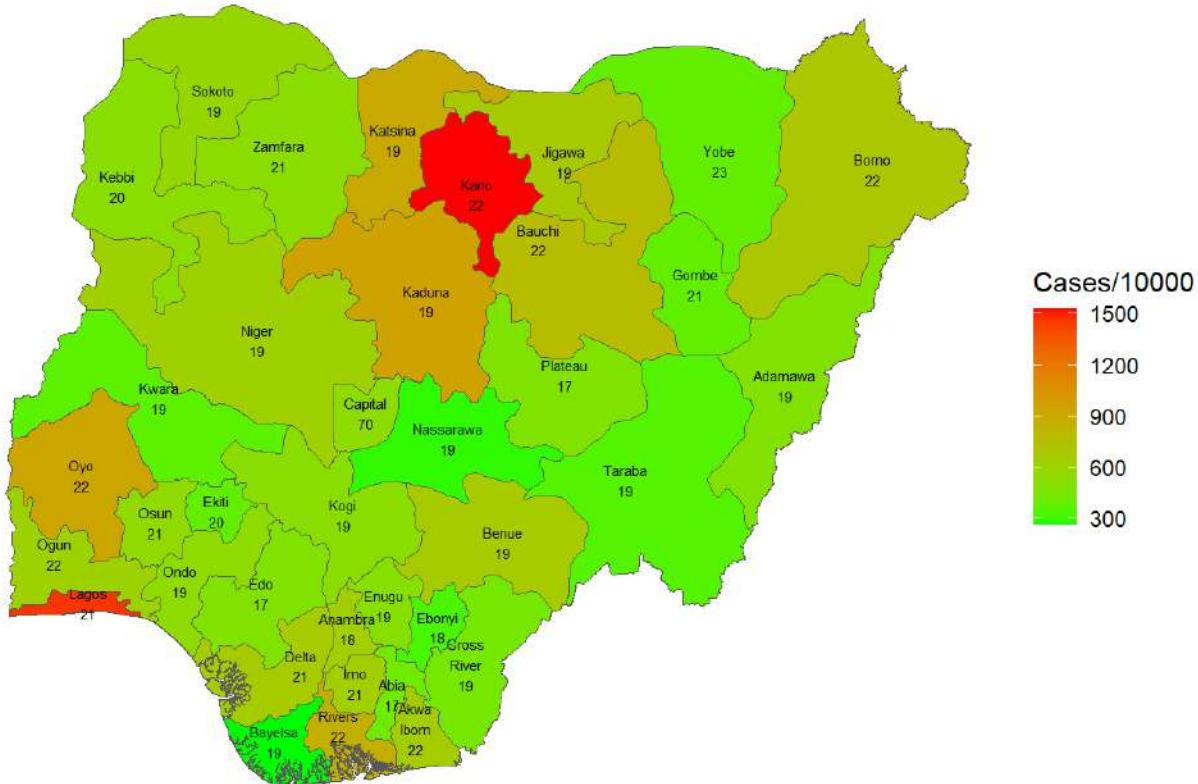
Let's delve into this visualization:

We want to present a choropleth map showcasing the malaria cases per 10000 residents across Nigerian states in 2021, with labels that combine state names and their respective increase rates from 2015.

```
# Combine the state names and their respective increase rates into a single
# label
malaria3$label_text <- paste(malaria3$state_name, malaria3$increase_rate)

# Visualize the data
ggplot(data = malaria3) +
  # Create a choropleth map shaded based on the number of malaria cases in
  # 2021 per 10,000 residents
  geom_sf(aes(fill = cases_2021/10000)) +
  # Add combined labels (state name and increase rate) to each state's
  # centroid
  geom_sf_text(aes(label = str_wrap(label_text, 2)), size = 2) +
  # Add titles and customize the color legend
  labs(title = "Nigeria Malaria cases in 2021 (%)", fill = "Cases/10000") +
  scale_fill_continuous(low = "green", high = "red") +  # Set color gradient
  theme_void()  # Apply a minimal theme for clarity
```

Nigeria Malaria cases in 2021 (%)



In this visualization:

- The line `malaria3$label_text <- paste(malaria3$state_name, malaria3$increase_rate)` constructs our combined labels by concatenating the state name with its increase rate
- `scale_fill_continuous(low = "green", high = "red")` assigns a color gradient based on the number of malaria cases. States with fewer cases will be colored green, transitioning to red for states with more cases.

This approach allows us to efficiently communicate two important pieces of information (cases per 10000 and increase rate) within the same visualization while keeping the name of states for easy identification.



Q: Create a single choropleth map that showcases the malaria increase rates across different states in Nigeria with each state labeled by its name

Instructions

1. Start by setting up the base plot using the `malaria3` dataset.
2. Create a choropleth map where the fill color represents the increase rate from 2015 to 2021.
3. Label each state with its name using the centroids.
4. Customize the color scale to transition from green (low increase) to red (high increase).
5. Add appropriate titles and labels to the plot.

Below a starter code:

```
# Combining visualization of increase rates with state names
ggplot(data = malaria3) +
  _____ + # Fill in the code to generate the choropleth map
  _____ + # Add state names to each region
  _____ + # Specify the color scale for increase rates
  _____ # Add titles and labels
```

Highlighting a Specific Region on the Map while Preserving Context

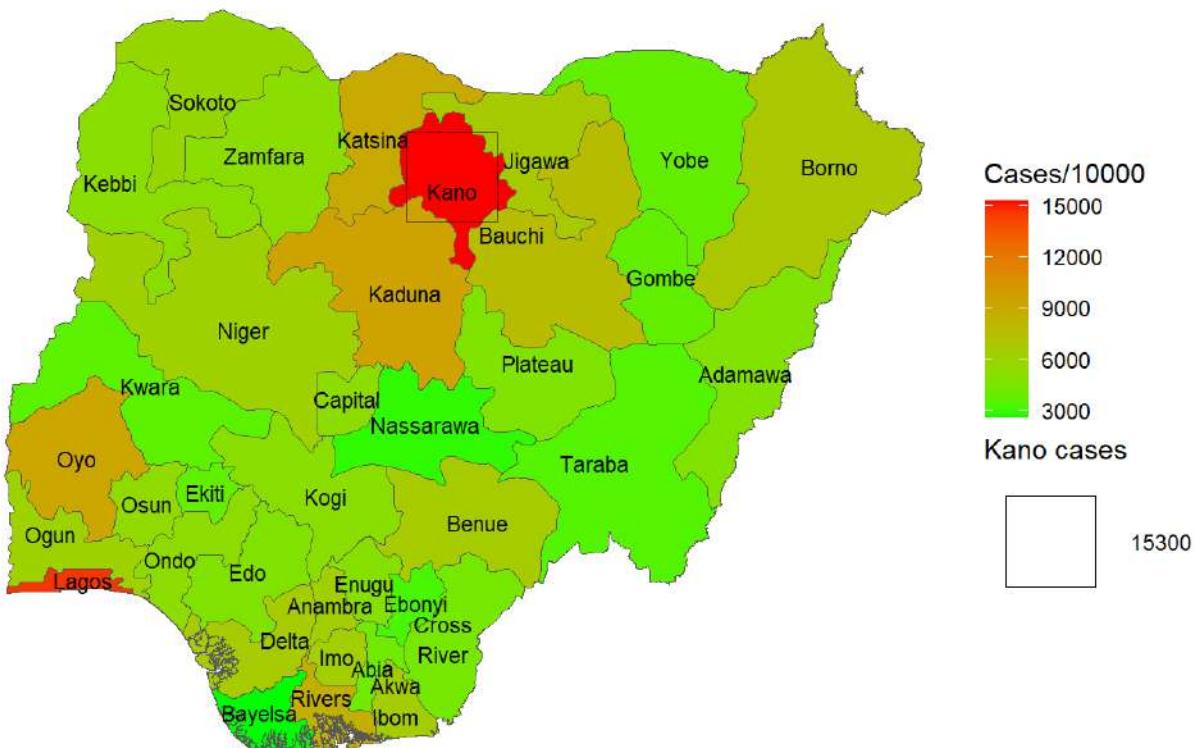
Sometimes, you might want to draw attention to a particular area or region on your map, without omitting details from the surrounding areas. By using specific graphical elements, like larger markers or distinctive colors, you can emphasize certain regions while still showcasing the broader data. In this example, we're focusing on the "Kano" region of Nigeria.

```
# Calculate centroid coordinates for labeling
centroid_coords <- st_coordinates(st_centroid(malaria3$geometry))

# Visualize the malaria cases across Nigeria with an emphasis on Kano
ggplot(data = malaria3) +
  # Create a choropleth map with color based on malaria cases in 2021
  geom_sf(aes(fill = cases_2021/1000)) +
  # Set a continuous color gradient from green to red
  scale_fill_continuous(low = "green", high = "red") +
  
  # Overlay a point on Kano to emphasize it. The size of the point corresponds
  # to the number of cases
  geom_point(data = subset(malaria3, state_name == "Kano"),
             aes(x = st_coordinates(st_centroid(geometry))[1],
                  y = st_coordinates(st_centroid(geometry))[2], size =
round(cases_2021/1000)),
             color = "black", shape = 22, fill = "transparent") +
  
  # Label each region with its name
  geom_sf_text(aes(label = str_wrap(state_name, 1)), size = 3) +
  
  # Customize the scale of the size of the emphasized point
  scale_size_continuous(range = c(15, 20)) +
  
  # Add title and legends
  labs(title = "Kano Malaria cases in 2021", subtitle = "Data from the Report
on Malaria in Nigeria 2022", size = "Kano cases", fill =
"Cases/10000",) +
  
  # Apply a minimal theme for clarity
  theme_void()
```

Kano Malaria cases in 2021

Data from the Report on Malaria in Nigeria 2022



In this visualization:

- The `geom_point()` function is utilized to lay a circle over the Kano region. The size of the circle signifies the number of cases in Kano in 2021. The circle is designed transparent (`fill = "transparent"`) with a bold black boundary (`color = "black"`) to set it apart.
- The `scale_size_continuous()` function used to customize the size scale for points (like the one on Kano), setting the range between 15 and 20. This ensures that the point size for Kano stands out relative to other elements on the map.



While “Kano” is emphasized, all other regions are also displayed with their respective color shading based on malaria cases. This offers a holistic view of the situation across Nigeria, enabling viewers to compare Kano with other regions.

Such an approach is invaluable when you wish to spotlight specific details or areas of interest without sidelining the broader dataset, enriching your data presentations.

Labeling Point Locations: Exploring Malaria Positive Rate and Incidence

Mapping and visualizing specific data points on a geographical map can provide crucial insights, especially when dealing with epidemiological data. Let's delve into the code to understand the processes and the visualization we're aiming to achieve:

```
# Data Retrieval
# The malariaAtlas package provides the `getPR` function to access the
# parasite rate (PR).
# PR is an essential indicator of malaria prevalence.

# Fetching data for Nigeria for both malaria species
nigeria_pr <- malariaAtlas::getPR(ISO = "NGA", species = "both") %>%
  # Filtering out records with missing PR values
  filter(!is.na(pr)) %>%
  # Removing any rows with missing longitude or latitude
  drop_na(longitude, latitude) %>%
  # Converting the data into a spatial dataframe to facilitate mapping
  st_as_sf(coords = c("longitude", "latitude"), crs = 4326)
```

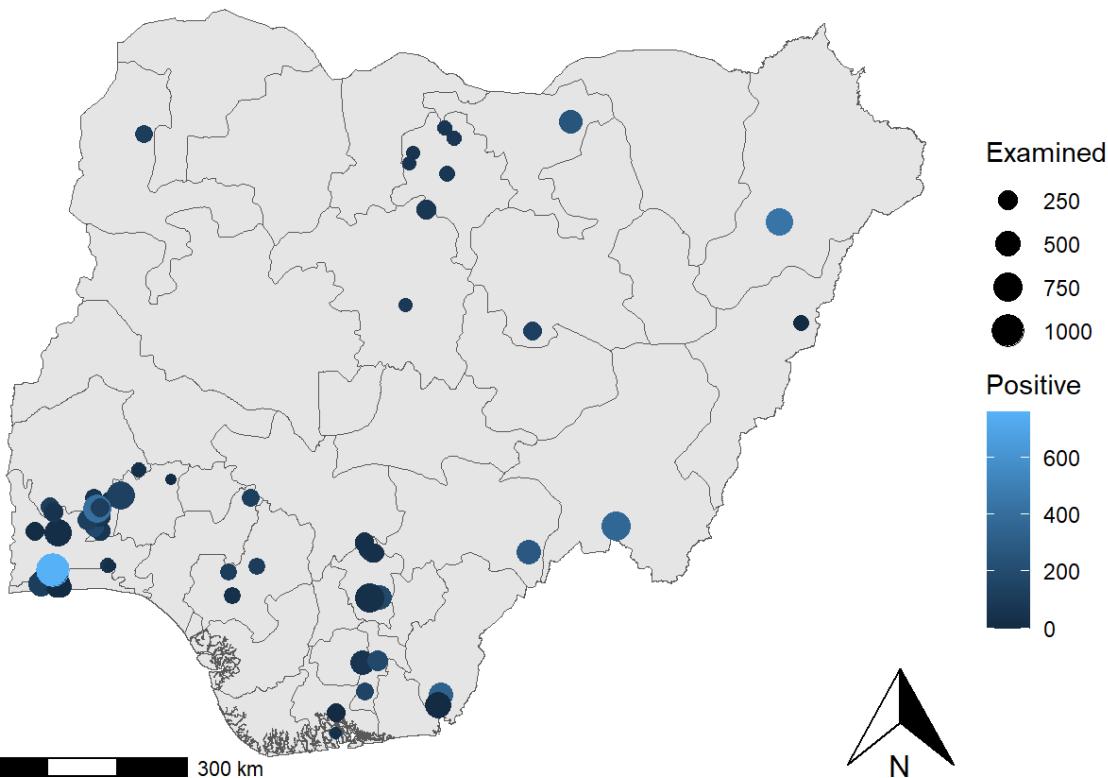
This chunk fetches malaria prevalence data for Nigeria. After retrieval, the data is cleansed by filtering out missing values. It's then transformed into a format suitable for geospatial visualization (`sf` object).

```
# Setting up a geospatial visualization using ggplot2

# Starting the plot
ggplot() +
  # Plotting the administrative boundaries of Nigeria
  geom_sf(data = nga_adm1) +
  # Adding points for each testing location
  # The color of each point indicates if the test was positive, and the size
  # represents the number of people tested
  geom_sf(aes(size = examined, color = positive), data = nigeria_pr) +
  # Setting titles, axis labels, and legends for the plot
  labs(title = "Location of Positive and Examined Malaria Cases in Nigeria",
       subtitle = "Data from Malaria Atlas Project",
       color = "Positive",
       size = "Examined") +
  # Adding labels for longitude and latitude
  xlab("Longitude") +
  ylab("Latitude") +
  # Incorporating a north arrow to provide orientation
  ggspatial::annotation_north_arrow(location = "br") +
  # Incorporating a scale bar to assist in distance interpretation
  ggspatial::annotation_scale(location = "bl") +
  # Applying a minimalist theme for visual clarity
  theme_void()
```

Location of Positive and Examined Malaria Cases in Nigeria

Data from Malaria Atlas Project



RECAP



This code chunk visualizes the malaria data on a map. It highlights regions based on the number of malaria tests and their outcomes. Specific tools from the `ggspatial` package are used to add cartographic elements, making the map more informative.

PRACTICE



Q: Visualize Positive Rate by Size and Color

Your final challenge is to create a visualization representing the positive rate of malaria for each location. Adjust the size of the points to reflect the positive rate. This task will test your understanding of combining different data indicators on a map. Good luck!



```
# Visualizing positive rate  
ggplot() +  
  _____ +  
  _____ +  
  _____ +  
  _____
```

WRAP UP!

Geospatial data visualization is more than just plotting data on a map. It's about narrating a story that's rooted in location and space. This lesson delved deep into a layers-based narrative approach, from the importance of clear annotations to the integration of diverse data types for a richer understanding.

Learning outcomes

By engaging with this lesson on map labeling, you should now be able to:

- Demonstrate the ability to use clear annotations and labels to make complex spatial data understandable.
- Show proficiency in integrating continuous indicators, overlaying labels, emphasizing regions, and determining optimal point placements on geospatial maps.
- Illustrate the importance of contextualizing highlighted areas within the larger geographical landscape to preserve the integrity and interpretability of the map.
- Utilize the functionalities of key R packages like ggplot2, sf, and ggspatial to facilitate advanced mapping processes.

Answer Key

Solution for practice 1

To visualize the distribution of malaria cases in Nigeria for the year 2015, follow the instructions provided in the exercise. Here's the complete solution:

```

# Constructing the choropleth map for 2015 using ggplot2
ggplot(data = malaria3) +
  geom_sf(aes(fill = cases_2015 / 1000)) + # Updated data column to reflect
  # 2015
  labs(title = "Nigeria Malaria Distributed Cases in 2015", fill = "Cases per
    1000") + # Updated title for 2015
  scale_fill_continuous(low = "blue", high = "yellow") + # Modified color
  # scale to blue-to-yellow gradient
  theme_void()

```

Solution for practice 2

To combine the visualization of increase rates with state names, follow the steps below:

```

# Combining visualization of increase rates with state names
ggplot(data = malaria3) +
  # Create a choropleth map with fill color based on the increase rate
  geom_sf(aes(fill = increase_rate), color="white", size = 0.2) +
  # Label each region with its state name
  geom_sf_text(aes(label = str_wrap(state_name, 1)), size = 2) +
  # Specify the color scale for increase rates
  scale_fill_continuous(name="Increase rate (%)",
    limits=c(0,70),
    low = "green",
    high = "red",
    breaks=c(0, 20, 40, 60)) +
  # Add a title and legend to the plot
  labs(title = "Nigeria Malaria Increase Rate from 2015 to 2021",
    fill = "Increase Rate (%)") +
  theme_void() # Apply a minimalistic theme for clarity

```

In this solution, the choropleth map is created based on the increase rate of malaria cases from 2015 to 2021. Each state in Nigeria is labeled by its name, and the color gradient (from green to red) showcases the magnitude of the increase rate. The map is enhanced with a title and a legend to ensure clarity and comprehension.

Solution for practice 3

Create a visualization that represents the positive rate:

```
# Adding a positive rate column
nigeria_pr$positive_rate <- (nigeria_pr$positive / nigeria_pr$examined) * 100

ggplot() +
  geom_sf(data = nga_adm1) +
  geom_sf(aes(size = positive_rate, color = positive_rate), data =
    nigeria_pr) +
  labs(title = "Location and Positive Rate of Malaria Cases in Nigeria",
       subtitle = "Data from Malaria Atlas Project",
       color = "Positive Rate (%)",
       size = "Positive Rate (%)") +
  xlab("Longitude") +
  ylab("Latitude") +
  ggspatial::annotation_north_arrow(location = "br") +
  ggspatial::annotation_scale(location = "bl") +
  theme_void()
```

Contributors

The following team members contributed to this lesson:



IMAD EL BADISY

Data Science Education Officer
Deeply interested in health data



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

- Wickham, Hadley, Winston Chang, and Maintainer Hadley Wickham. “Package ‘ggplot2’.” *Create elegant data visualisations using the grammar of graphics.* Version 2, no. 1 (2016): 1-189.
- Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. *R for data science.* ” O'Reilly Media, Inc.”, 2023.
- Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. *Geocomputation with R.* CRC Press, 2019.

Organizing Public Health Data with gt Tables in R - Basics

GRAPH Network & WHO, supported by the Global Fund
to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Packages
Introducing the dataset
Creating simple tables with {gt}
Customizing {gt} tables
Table Header and Footer
Stub
Spanner columns & sub columns
Renaming Table Columns
Summary rows
Wrap-up
Answer Key

Introduction

Tables are a powerful tool for visualizing data in a clear and concise format. With R and the `gt` package, we can leverage the visual appeal of tables to efficiently communicate key information. In this lesson, we will learn how to build aesthetically pleasing, customizable tables that support data analysis goals.

Learning objectives

- Use the `gt()` function to create basic tables
- Group columns under spanner headings
- Relabel column names
- Add summary rows for groups

By the end, you will be able to generate polished, reproducible tables like this:

Sum of HIV cases in Malawi

from Q1 2019 to Q2 2019

period	New cases		Previous cases	
	Positive	Negative	Positive	Negative
Central Region				
2019 Q1	2004	123018	3682	2562
2019 Q2	1913	116443	3603	1839
2019 Q3	1916	127799	4002	2645
2019 Q4	1691	124728	3754	1052
sum	—	7524.00	491988.00	15041.00
mean	—	1881.00	122997.00	3760.25
Northern Region				
2019 Q1	664	36196	1197	675
2019 Q2	582	35315	1084	590
2019 Q3	570	36850	1191	542
2019 Q4	519	34322	1132	346
sum	—	2335.00	142683.00	4604.00
mean	—	583.75	35670.75	1151.00
Southern Region				
2019 Q1	3531	125480	9937	3358
2019 Q2	3637	130491	10414	3176

Example summary table

Packages

We will use these packages:

- `{gt}` for creating tables
- `{tidyverse}` for data wrangling
- `{here}` for file paths

```
# Load packages
pacman::p_load(tidyverse, gt, here)
```

Introducing the dataset

Our data comes from the **Malawi HIV Program** and covers antenatal care and HIV treatment during 2019. We will focus on quarterly regional and facility-level aggregates (available [here](#)).

```
# Import data
hiv_malawi <- read_csv(here::here("data/clean/hiv_malawi.csv"))
```

Let's explore the variables:

```
# First 6 rows
head(hiv_malawi)
```

```
# Variable names and types
glimpse(hiv_malawi)
```

```
## Rows: 17,235
## Columns: 29
## $ region
## $ zone
## $ district
## $ traditional_authority
## $ facility_name
## $ datim_code
## $ system
## $ hsector
## $ period
## $ reporting_period
## $ sub_groups
## $ new_women_registered
## $ total_women_in_booking_cohort
## $ region
## $ zone
## $ district
## $ traditional_authority
## $ facility_name
## $ datim_code
## $ system
## $ hsector
## $ period
## $ reporting_period
## $ sub_groups
## $ new_women_registered
## $ total_women_in_booking_cohort
## $ region
## $ zone
## $ district
## $ traditional_authority
## $ facility_name
## $ datim_code
## $ system
## $ hsector
## $ period
## $ reporting_period
## $ sub_groups
## $ new_women_registered
## $ total_women_in_booking_cohort
```

```

## $ not_tested_for_syphilis <dbl> NA, 45, NA, ...
## $ syphilis_negative <dbl> NA, 10, NA, ...
## $ syphilis_positive <dbl> NA, 0, NA, ...
## $ hiv_status_not_ascertained <dbl> 4, 7, 9, 4, ...
## $ previous_negative <dbl> 0, 0, 0, 0, ...
## $ previous_positive <dbl> 0, 0, 0, 1, ...
## $ new_negative <dbl> 40, 47, 30, ...
## $ new_positive <dbl> 1, 1, 1, 1, ...
## $ not_on_cpt <dbl> NA, 0, NA, ...
## $ on_cpt <dbl> NA, 1, NA, ...
## $ no_ar_vs <dbl> 0, 0, 0, 0, ...
## $ already_on_art_when_starting_anc <dbl> 0, 1, 0, 1, ...
## $ started_art_at_0_27_weeks_of_pregnancy <dbl> 1, 0, 1, 1, ...
## $ started_art_at_28_weeks_of_preg <dbl> 0, 0, 0, 0, ...
## $ no_ar_vs_dispensed_for_infant <dbl> NA, 0, NA, ...
## $ ar_vs_dispensed_for_infant <dbl> NA, 1, NA, ...

```

The data covers geographic regions, healthcare facilities, time periods, patient demographics, test results, preventive therapies, antiretroviral drugs, and more. More information about the dataset is in the appendix section.

The key variables we will be considering are:

1. `previous_negative`: The number of patients who visited the healthcare facility in that quarter that had prior negative HIV tests.
2. `previous_positive`: The number of patients (as above) with prior positive HIV tests.
3. `new_negative`: The number of patients newly testing negative for HIV.
4. `new_positive`: The number of patients newly testing positive for HIV.

In this lesson, we will aggregate the data by quarter and summarize changes in HIV test results.

Creating simple tables with `{gt}`

`{gt}`'s flexibility, efficiency, and power make it a formidable package for creating tables in R. We'll explore some of its core features in this lesson.

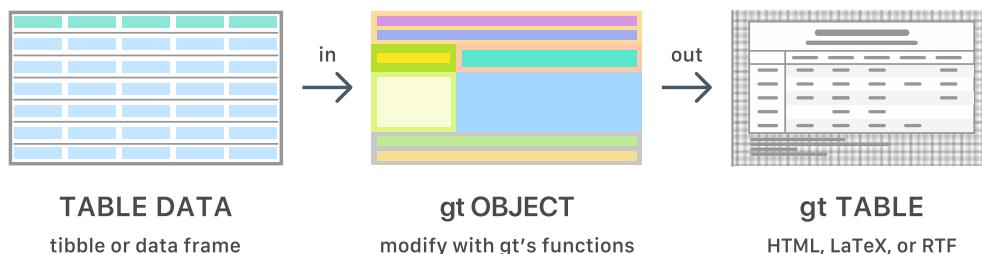
KEY POINT



KEY POINT



A Typical `gt` Workflow



The `{gt}` package contains a set of functions that take raw data as input and output a nicely formatted table for further analysis and reporting.

To effectively leverage the `{gt}` package, we first need to wrangle our data into an appropriate summarized form.

In the code chunk below, we use `{dplyr}` functions to summarize HIV testing in select Malawi testing centers by quarter. We first group the data by time period, then sum case counts across multiple variables using `across()`:

```
# Variables to summarize
cols <- c("new_positive", "previous_positive", "new_negative",
        "previous_negative")

# Create summary by quarter
hiv_malawi_summary <- hiv_malawi %>%
  group_by(period) %>%
  summarize(
    across(all_of(cols), sum) # Summarize all columns
  )

hiv_malawi_summary
```

```
## # A tibble: 4 × 5
##   period new_positive previous_positive new_negative
##   <chr>      <dbl>            <dbl>          <dbl>
## 1 2019 Q1     6199             14816         284694
## 2 2019 Q2     6132             15101         282249
## 3 2019 Q3     5907             15799         300529
## 4 2019 Q4     5646             15700         291622
## # i 1 more variable: previous_negative <dbl>
```

This aggregates the data nicely for passing to `{gt}` to generate a clean summary table.

To create a simple table from the aggregated data, we can then call the `gt()` function:

```
hiv_malawi_summary %>%  
  gt()
```

period	new_positive	previous_positive	new_negative	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

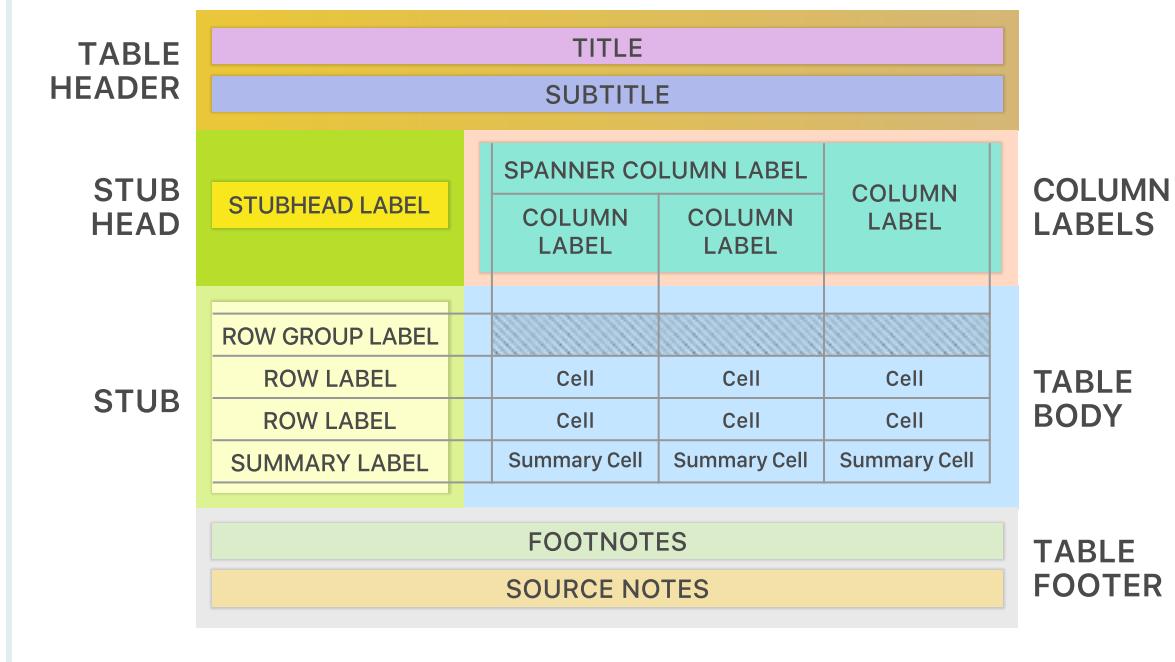
As you can see, the default table formatting is quite plain and unrefined. However, `{gt}` provides many options to customize and beautify the table output. We'll delve into these in the next section.

Customizing `{gt}` tables

The `{gt}` package allows full customization of tables through its “grammar of tables” framework. This is similar to how `{ggplot2}`’s grammar of graphics works for plotting.

To take full advantage of `{gt}`, it helps to understand some key components of its grammar.

The Parts of a gt Table



As seen in the figure from the package website, The main components of a `{gt}` table are:

- Table Header**: Contains an optional title and subtitle
- Stub**: Row labels that identify each row
- Stub Head**: Optional grouping and labels for stub rows
- Column Labels**: Headers for each column
- Table Body**: The main data cells of the table
- Table Footer**: Optional footnotes and source notes

Understanding this anatomy allows us to systematically construct `{gt}` tables using its grammar.

Table Header and Footer

The basic table we had can now be updated with more components.

Tables become more informative and professional-looking with the addition of headers, source notes, and footnotes. We can easily enhance the basic table from before by adding these elements using `{gt}` functions.

To create a header, we use `tab_header()` and specify a `title` and `subtitle`. This gives the reader context about what the table shows.

```

hiv_malawi_summary %>%
  gt() %>%
  tab_header(
    title = "HIV Testing in Malawi",
    subtitle = "Q1 to Q4 2019"
  )

```

HIV Testing in Malawi

Q1 to Q4 2019

period	new_positive	previous_positive	new_negative	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

We can add a footer with the function `tab_source_note()` to cite where the data came from:

```

hiv_malawi_summary %>%
  gt() %>%
  tab_header(
    title = "HIV Testing in Malawi",
    subtitle = "Q1 to Q4 2019"
  ) %>%
  tab_source_note("Source: Malawi HIV Program")

```

HIV Testing in Malawi

Q1 to Q4 2019

period	new_positive	previous_positive	new_negative	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

Source: Malawi HIV Program

Footnotes are useful for providing further details about certain data points or labels. The `tab_footnote()` function attaches footnotes to indicated table cells. For example, we can footnote the diagnosis columns:

```
hiv_malawi_summary %>%
  gt() %>%
  tab_header(
    title = "HIV Testing in Malawi",
    subtitle = "Q1 to Q2 2019"
  ) %>%
  tab_source_note("Source: Malawi HIV Program") %>%
  tab_footnote(
    footnote = "New diagnosis",
    locations = cells_column_labels(
      columns = c(new_positive, new_negative)
    )
  )
```

HIV Testing in Malawi

Q1 to Q2 2019

period	new_positive ¹	previous_positive	new_negative ¹	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

¹ New diagnosis

Source: Malawi HIV Program

These small additions greatly improve the professional appearance and informativeness of tables.

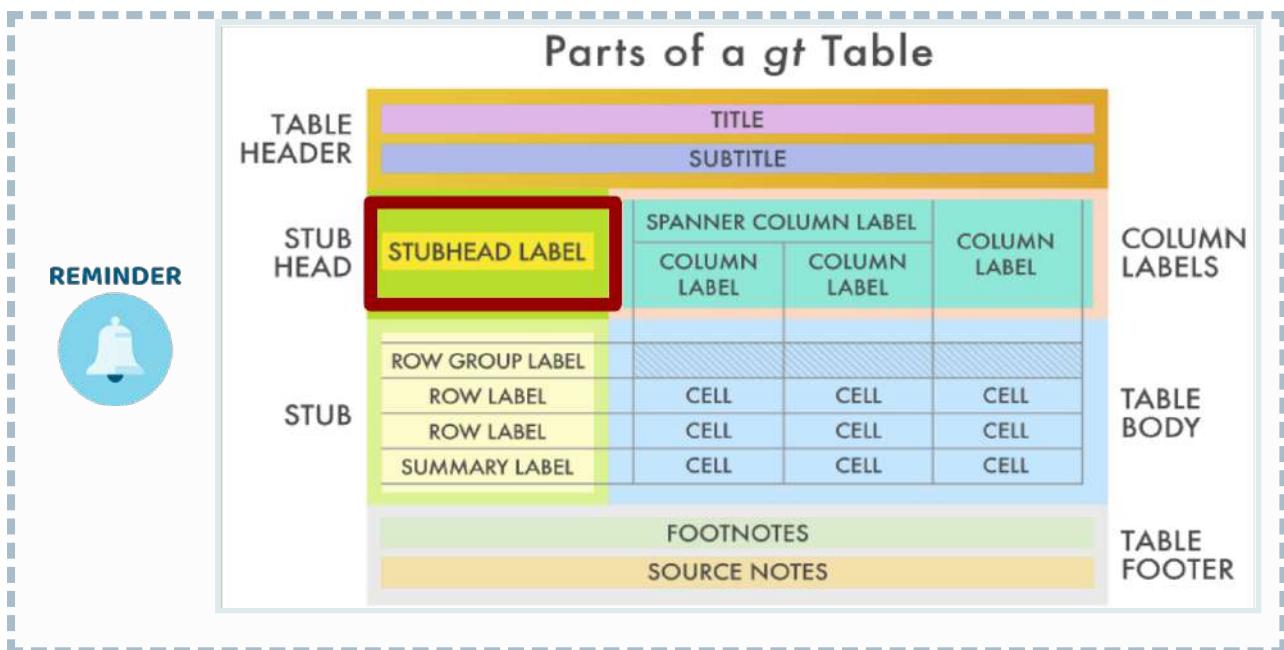
Stub

The stub is the left section of a table containing the row labels. These provide context for each row's data.

REMINDER



This image displays the stub component of a `{gt}` table, marked with a red square.



In our HIV case table, the `period` column holds the row labels we want to use. To generate a stub, we specify this column in `gt()` using the `rowname_col` argument:

```
hiv_malawi_summary %>%
  gt(rowname_col = "period") %>%
  tab_header(
    title = "HIV Testing in Malawi",
    subtitle = "Q1 to Q2 2019"
  ) %>%
  tab_source_note("Source: Malawi HIV Program")
```

	new_positive	previous_positive	new_negative	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

Source: Malawi HIV Program

Note that the column name passed to `rowname_col` should be in quotes.

For convenience, let's save the table to a variable `t1`:

```
t1 <- hiv_malawi_summary %>%
  gt(rownames_col = "period") %>%
  tab_header(
    title = "HIV Testing in Malawi",
    subtitle = "Q1 to Q2 2019"
  ) %>%
  tab_source_note("Source: Malawi HIV Program")

t1
```

HIV Testing in Malawi

Q1 to Q2 2019

	new_positive	previous_positive	new_negative	previous_negative
2019 Q1	6199	14816	284694	6595
2019 Q2	6132	15101	282249	5605
2019 Q3	5907	15799	300529	6491
2019 Q4	5646	15700	291622	6293

Source: Malawi HIV Program

Spanner columns & sub columns

To better structure our table, we can group related columns under “spanners”. Spanners are headings that span multiple columns, providing a higher-level categorical organization. We can do this with the `tab_spinner()` function.

Let's create two spanner columns for new and Previous tests. We'll start with the “New tests” spanner so you can observe the syntax:

```
t1 %>%
  tab_spinner(
    label = "New tests",
    columns = starts_with("new") # selects columns starting with "new"
  )
```

HIV Testing in Malawi

Q1 to Q2 2019

New tests

	new_positive	new_negative	previous_positive	previous_negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

The `columns` argument lets us select the relevant columns, and the `label` argument takes in the span label.

Let's now add both spanners:

```
# Save table to t2 for easy access
t2 <- t1 %>%
  # First spanner for "New tests"
  tab_spanner(
    label = "New tests",
    columns = starts_with("new")
  ) %>%
  # Second spanner for "Previous tests"
  tab_spanner(
    label = "Previous tests",
    columns = starts_with("prev")
  )

t2
```

HIV Testing in Malawi

Q1 to Q2 2019

	New tests		Previous tests	
	new_positive	new_negative	previous_positive	previous_negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Note that the `tab_spacer` function automatically rearranged the columns in an appropriate way.



Question 1: The Purpose of Spacers

What is the purpose of using “spacer columns” in a `gt` table?

- A. To apply custom CSS styles to specific columns.
- B. To create group columns and increase readability.
- C. To format the font size of all columns uniformly.
- D. To sort the data in ascending order.

Question 2: Spacers Creation

Using the `hiv_malawi` data frame, create a `gt` table that displays a summary of the `sum` of “new_positive” and “previous_positive” cases for each region. Create spacer headers to label these two summary columns. To achieve this, fill in the missing parts of the code below:



```
region_summary <- hiv_malawi %>%
  group_by(region) %>%
  summarize(
    _____ (
      c(new_positive, previous_positive),
      _____
    )
  )

# Create a gt table with spanner headers
summary_table_spanners <- region_summary %>%
  _____ %>%
  _____ (
    label = "Positive cases",
    _____ = c(new_positive, previous_positive)
  )
```

Renaming Table Columns

The column names currently contain unneeded prefixes like “new_” and “previous_”. For better readability, we can rename these using `cols_label()`.

`cols_label()` takes a set of old names to match (on the left side of a tilde, ~) and new names to replace them with (on the right side of the tilde). We can use `contains()` to select columns with “positive” or “negative”:

```
t3 <- t2 %>%
  cols_label(
    contains("positive") ~ "Positive",
    contains("negative") ~ "Negative"
  )

t3
```

HIV Testing in Malawi

Q1 to Q2 2019

	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

This relabels the columns in a cleaner way.

`cols_label()` accepts several column selection helpers like `contains()`, `starts_with()`, `ends_with()` etc. These come from the `tidyselect` package and provide flexibility in renaming.

`cols_label()` has more identification function like `contains()` that work in a similar manner that are identical to the `tidyselect` helpers, these also include :



- `starts_with()`: Starts with an exact prefix.
- `ends_with()`: Ends with an exact suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like x01, x02, x03.

These helpers are useful especially in the case of multiple columns selection.

More on the `cols_label()` function can be found here: https://gt.rstudio.com/reference/cols_label.html

Question 3: column labels

Which function is used to change the labels or names of columns in a `gt` table?

PRACTICE



(in RMD)

- A. ``tab_header()``
- B. ``tab_style()``
- C. ``tab_options()``
- D. ``tab_relabel()``

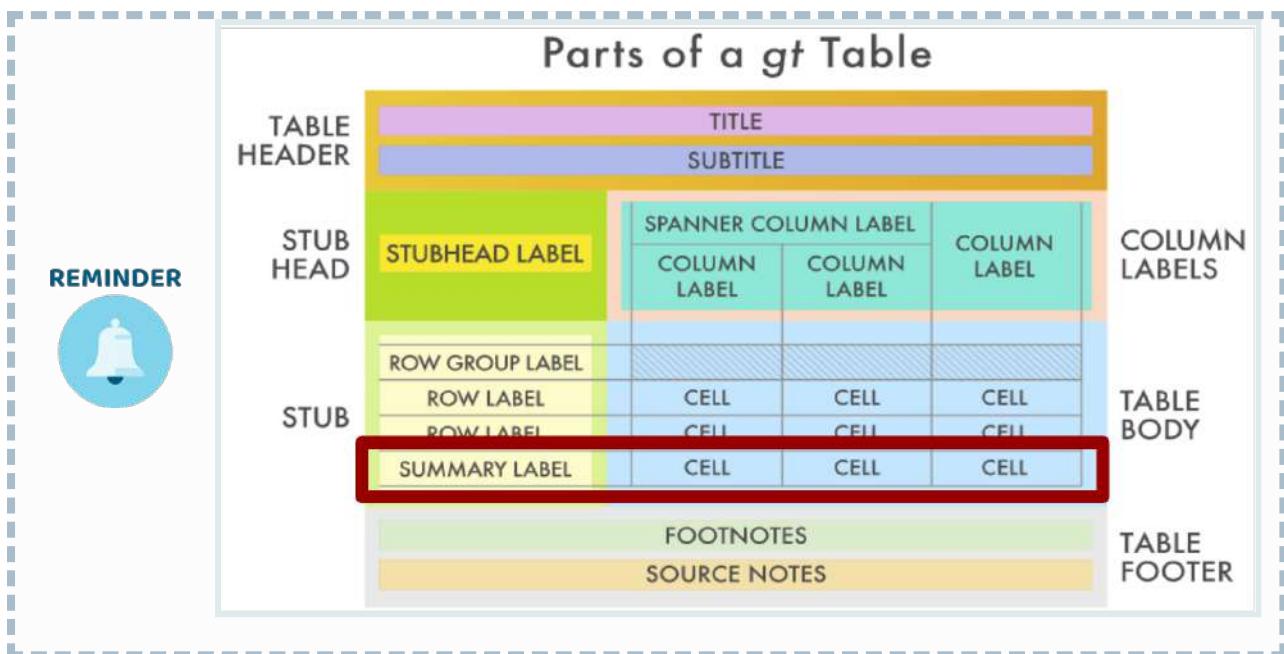
Summary rows

Let's take the same data we started with in the beginning of this lesson and instead of only grouping by period(quarters), let's group by both period and region. We will do this to illustrate the power of summarization features in `gt`: summary tables.

REMINDER



{gt} reminder - Summary Rows This image shows the summary rows component of a `{gt}` table, clearly indicated within a red square. Summary rows, provide aggregated data or statistical summaries of the data contained in the corresponding columns.



First let's recreate the data:

```
summary_data_2 <- hiv_malawi %>%
  group_by(
    # Note the order of the variables we group by.
    region,
    period
  ) %>%
  summarise(
    across(all_of(cols), sum)
  ) %>%
  gt()
```

`summarise()` has grouped output by 'region'. You can override using the ## `.`groups` argument.

```
summary_data_2
```

period	new_positive	previous_positive	new_negative	previous_negative
Central Region				
2019 Q1	2004	3682	123018	2562
2019 Q2	1913	3603	116443	1839
2019 Q3	1916	4002	127799	2645
2019 Q4	1691	3754	124728	1052
Northern Region				
2019 Q1	664	1197	36196	675
2019 Q2	582	1084	35315	590
2019 Q3	570	1191	36850	542
2019 Q4	519	1132	34322	346
Southern Region				
2019 Q1	3531	9937	125480	3358
2019 Q2	3637	10414	130491	3176
2019 Q3	3421	10606	135880	3304
2019 Q4	3436	10814	132572	4895

WATCH OUT



The order in the `group_by()` function affects the row groups in the `gt` table.

Second, let's re-incorporate all the changes we've done previously into this table:

```
# saving the progress to the t4 object

t4 <- summary_data_2 %>%
  tab_header(
    title = "Sum of HIV Tests in Malawi",
    subtitle = "from Q1 2019 to Q4 2019"
  ) %>%
  tab_source_note("Data source: Malawi HIV Program") %>% tab_spanner(
    label = "New tests",
    columns = starts_with("new") # selects columns starting with "new"
  ) %>%
  # creating the first spanner for the Previous tests
  tab_spanner(
    label = "Previous tests",
    columns = starts_with("prev") # selects columns starting with "prev"
  ) %>%
  cols_label(
    # locate #### assign
    contains("positive") ~ "Positive",
    contains("negative") ~ "Negative"
  )
)

t4
```

Sum of HIV Tests in Malawi

from Q1 2019 to Q4 2019

period	New tests		Previous tests	
	Positive	Negative	Positive	Negative
Central Region				
2019 Q1	2004	123018	3682	2562
2019 Q2	1913	116443	3603	1839
2019 Q3	1916	127799	4002	2645
2019 Q4	1691	124728	3754	1052
Northern Region				
2019 Q1	664	36196	1197	675
2019 Q2	582	35315	1084	590
2019 Q3	570	36850	1191	542
2019 Q4	519	34322	1132	346
Southern Region				
2019 Q1	3531	125480	9937	3358
2019 Q2	3637	130491	10414	3176
2019 Q3	3421	135880	10606	3304
2019 Q4	3436	132572	10814	4895

Data source: Malawi HIV Program

Now, what if we want to visualize on the table a summary of each variable for every region group? More precisely we want to see the sum and the mean for the 4 columns we have for each region.

REMINDER



REMINDER

only changed the labels of these columns in the `gt` table and not in the `data.frame` itself, so we can use the names of these columns to tell `gt` where to apply the summary function. Additionally, we already stored the names of these 4 columns in the object `cols` so we will use it again here.

In order to achieve this we will use the handy function `summary_rows` where we explicitly provide the columns that we want summarized, and the functions we want to summarize with, in our case it's `sum` and `mean`. Note that we assign the name of the new row(unquoted) a function name ("quoted").

```
t5 <- t4 %>%
  summary_rows(
    columns = cols, #using columns = 3:6 also works
    fns = list(
      TOTAL = "sum",
      AVERAGE = "mean"
    )
  )
t5
```

Sum of HIV Tests in Malawi

from Q1 2019 to Q4 2019

period	New tests		Previous tests	
	Positive	Negative	Positive	Negative
Central Region				
2019 Q1	2004	123018	3682	2562
2019 Q2	1913	116443	3603	1839
2019 Q3	1916	127799	4002	2645
2019 Q4	1691	124728	3754	1052
sum	—	7524.00	491988.00	15041.00
mean	—	1881.00	122997.00	3760.25
Northern Region				
2019 Q1	664	36196	1197	675
2019 Q2	582	35315	1084	590
2019 Q3	570	36850	1191	542
2019 Q4	519	34322	1132	346
sum	—	2335.00	142683.00	4604.00
mean	—	583.75	35670.75	1151.00
Southern Region				
2019 Q1	3531	125480	9937	3358
2019 Q2	3637	130491	10414	3176
2019 Q3	3421	135880	10606	3304
2019 Q4	3436	132572	10814	4895
sum	—	14025.00	524423.00	41771.00
				14733.00

Sum of HIV Tests in Malawi				
from Q1 2019 to Q4 2019				
period	New tests		Previous tests	
	Positive	Negative	Positive	Negative
				Data source: Malawi HIV Program

Question 4 : summary rows

What is the correct answer (or answers) if you had to summarize the standard deviation of the rows of columns “new_positive” and “previous_negative” only?

- A. Use `summary_rows()` with the `columns` argument set to “new_positive” and “previous_negative” and `fns` argument set to “sd”.

```
# Option A
your_data %>%
  summary_rows(
    columns = c("new_positive", "previous_negative"),
    fns = "sd"
  )
```



- B. Use `summary_rows()` with the `columns` argument set to “new_positive” and “previous_negative” and `fns` argument set to “summarize(sd)”.

```
# Option B
your_data %>%
  summary_rows(
    columns = c("new_positive", "previous_negative"),
    fns = summarize(sd)
  )
```

- C. Use `summary_rows()` with the `columns` argument set to “new_positive” and “previous_negative” and `fns` argument set to `list(SD = "sd")`.

```
# Option C
your_data %>%
  summary_rows(
    columns = c("new_positive", "previous_negative"),
    fns = list(SD = "sd")
  )
```



D. Use `summary_rows()` with the `columns` argument set to “new_positive” and “previous_negative” and `fns` argument set to “standard_deviation”.

```
# Option D
your_data %>%
  summary_rows(
    columns = c("new_positive", "previous_negative"),
    fns = "standard_deviation"
  )
```

Wrap-up

In today’s lesson, we got down to business with data tables in R using `gt`. We started by setting some clear goals, introduced the packages we’ll be using, and met our dataset. Then, we got our hands dirty by creating straightforward tables. We learned how to organize our data neatly using spanner columns and tweaking column labels to make things crystal clear and coherent. Then wrapped up with some nifty table summaries. These are the nuts and bolts of table-making in R and `gt`, and they’ll be super handy as we continue our journey on creating engaging and informative tables in R.

Answer Key

1. B

2.

```
# Solutions are where the numbered lines are

# summarize data first
district_summary <- hiv_malawi %>%
  group_by(region) %>%
  summarize(
    across( #1
      c(new_positive, previous_positive),
      sum #2
    )
  )

# Create a gt table with spanner headers
summary_table_spanners <- district_summary %>%
  gt() %>% #3
  tab_spanner( #4
    label = "Positive cases",
    columns = c(new_positive, previous_positive) #5
  )
```

3. D

4. C

Contributors

The following team members contributed to this lesson:



BENNOEUR HSIN

Data Science Education Officer
Data Visualization enthusiast



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

1. Tom Mock, “The Definite Cookbook of {gt}” (2021), The Mockup Blog, <https://themockup.blog/static/resources/gt-cookbook.html#introduction>.
2. Tom Mock, “The Grammar of Tables” (May 16, 2020), The Mockup Blog, <https://themockup.blog/posts/2020-05-16-gt-a-grammar-of-tables/#add-titles>.
3. RStudio, “Introduction to Creating gt Tables,” Official {gt} Documentation, <https://gt.rstudio.com/articles/intro-creating-gt-tables.html>.
4. Fleming, Jessica A., Alister Munthali, Bagrey Ngwira, John Kadzandira, Monica Jamili-Phiri, Justin R. Ortiz, Philipp Lambach, et al. 2019. “Maternal Immunization in Malawi: A Mixed Methods Study of Community Perceptions, Programmatic Considerations, and Recommendations for Future Planning.” *Vaccine* 37 (32): 4568–75. <https://doi.org/10.1016/j.vaccine.2019.06.020>.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Optimizing gt Tables for Enhanced Visualization

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Learning objectives
Packages
Previously in pt1
Dataset
Themes
Formatting the values in the table
Conditional formatting
Fonts and text
Borders
Answer Key
External resources and packages

Introduction

The previous `gt` lesson focused mainly on the components of the table its structure and how to manipulate it properly. This lesson, presenting the second part of the `gt` series will focus on using the package to polish, style, and customize the visual effects of tables in a way that elevate the quality and efficiency of your reports.

Let's dig in.

Learning objectives

- Cells Formatting
- Conditional coloring
- Format text(font color, bold,etc.)
- Add borders to text

By the conclusion of this lesson, you will have the skills to artfully style your `gt` tables to meet your specific preferences achieving a level of detail similar to this:

HIV Testing in Malawi				
	Q1 to Q2 2019			
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Packages

In this lesson, we will use the following packages:

- `gt`
- `dplyr`, `tidyr`, and `purrr`.
- `janitor`
- `KableExtra`
- `Paletteer`, `ggsci`

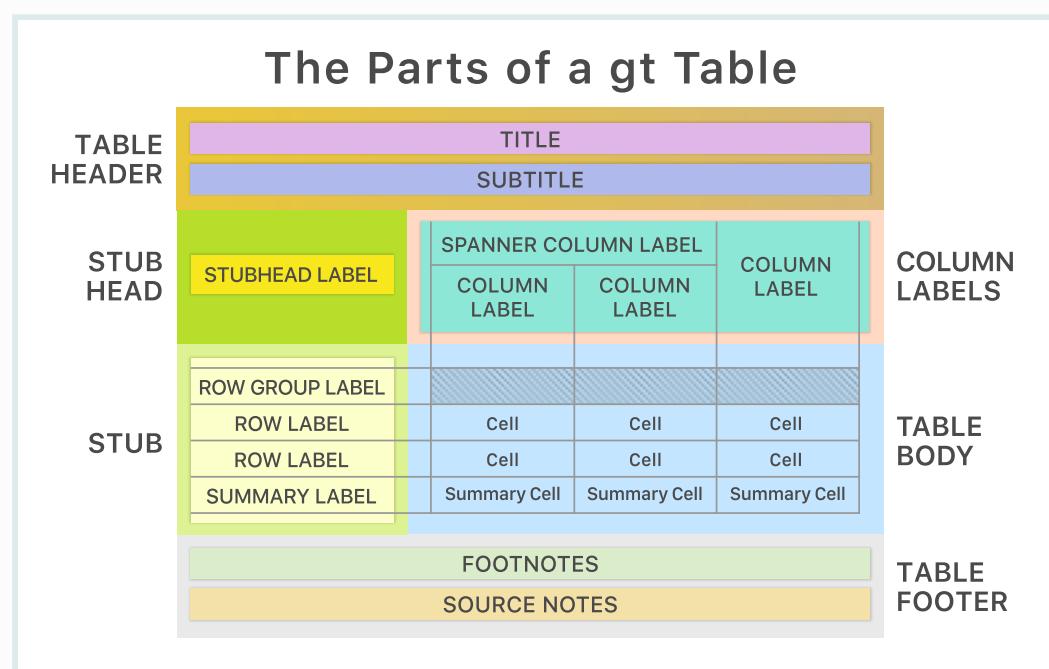
```
pacman::p_load(tidyverse, janitor, gt, here, paletteer, webshot2)
```

Previously in pt1



In the previous `gt` lesson we had the opportunity to :

- Discover the HIV prevalence data of Malawi.
- Discover the grammar of tables and the `gt` package.
- create simple table.
- Add details like title and footnote to the table.
- Group columns into spanners.
- Create Summary rows.



Dataset

In this lesson, we will use the same data from the previous lesson, you can go back for a detailed description of the data and the preparation process we made.



Here's the full details of the columns we will use:



- region: The geographical region or area where the data was collected or is being analyzed.
- period: A specific time period associated with the data, often used for temporal analysis.
- previous_negative: The count or number of individuals with a previous negative test result.
- previous_positive: The count or number of individuals with a previous positive test result.
- new_negative: The count or number of newly diagnosed cases with a negative result.
- new_positive: The count or number of newly diagnosed cases with a positive result.

But for the purposes of this lesson we will use the tables directly, this is the table that we created with the right spanners and column labels, we will base the rest of our lesson on this particular one.

```
hiv_malawi_summary <- read_rds(here::here("data", "clean",  
"malawi_hiv_summary_t3.rds"))
```

```
hiv_malawi_summary
```

HIV Testing in Malawi				
Q1 to Q2 2019				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Themes

Since the objective of this lesson is mainly styling, let's start with using a pre-defined theme to add more visuals and colors to the table and its components. To do so we use the `opt_stylize` function. The function contains multiple pre-defined styles and can accept a color as well. In our case we chose to go with style No.6 and the color 'gray', you can set these arguments to your liking.

```
t1 <- hiv_malawi_summary %>%
  opt_stylize(
    style = 1,
    color = 'cyan'
  ) %>%
  tab_options(
    stub.background.color = '#F4F4F4',
  )

t1
```

HIV Testing in Malawi				
Q1 to Q2 2019				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program



For more sophisticated themes and styling, you can refer to the function `tab_options` (documentations [here](#)) which is basically the equivalent to the `theme` function in `ggplot2`. This function contains arguments and options on every single layer and component of the table. For the purposes of this lesson we won't dive into it.

Formatting the values in the table

Wouldn't it be useful to visualize in colors the difference between values in a specific column? In many reports, these kind of tables are quite useful especially if the number of rows is quite large. Let's do this for our table such that we have the `new_positive` column is formatted red.

We can do this by means of the `data_color` function for which we need two specify tow arguments, `columns` (as in at what column this styling will take place?) and `palette` as the color palette we intend to use.

```
t2 <- t1 %>%
  data_color(
    columns = new_positive, # the column or columns as we will see later
    palette = "ggsci::red_material" # the palette form the ggsci package.
  )
t2
```

HIV Testing in Malawi				
Q1 to Q2 2019				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

SIDE NOTE



`ggsci::red_material` is not the only palette we can use, in fact there are hundereds of palettes that are designed to be used in R. You can find a lot more in the `palatteer` package documentations in here, or in the official `data_color` documentation here.

We can do this for the `previous_negative` column as well. We can use a different kind of palette, I'm using for this case the green palette from the same package:

`ggsci::green_material`, the palette you choose is a matter of convenience and personal taste, you can explore more about this if you refer to the side note above.

```
t2 %>%
  data_color(
    columns = previous_negative,
    palette = "ggsci::green_material"
  )
```

HIV Testing in Malawi				
Q1 to Q2 2019				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Similarly, we can also color multiple columns at once, for example we can style the columns with positive cases in red, and those with negative cases in green. To do this we need to write *two* `data_color` statements one for each color style:

```
t4 <- t1 %>%
  data_color(
    columns = ends_with("positive"), # selecting columns ending with the word
    positive
    palette = "ggsci::red_material" # red palette
  ) %>%
  data_color(
    columns = ends_with("negative"), # selecting columns ending with the word
    negative
    palette = "ggsci::green_material" # green palette
  )
```

```
t4
```

HIV Testing in Malawi

Q1 to Q2 2019

	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

REMINDER



Remember in the previous lesson we used the `tidyselect` functions to select columns, in the code above we used the function `ends_with` to select the columns ending either with the word ‘negative’ or ‘positive’ which is perfect for the purpose of our table.

Again, the column labels in the `gt` table and the actual column names in the `data.frame` can be different, in our case we refer to the names in the data.

Conditional formatting

We can also set up the table to conditionally change the style of a cell given its value. In our case we want to highlight values in the column `previous_positive` according to a threshold (the value 15700). Greater or equal values than the threshold should be in green.

To achieve this we use the `tab_style` function where we specify two arguments:

- `style` : where we specify the color in the `cell_text` function since we intend to manipulate the text within the cells.
- `location` : where we specify the columns and the rows of our manipulation in the `cells_body` since these cells are in the main body of the table.

Let's use the t2 table as an example:

```
t5 <- t2 %>%
  tab_style(
    style = cell_text(
      color = "red",
    ),
    locations = cells_body(
      columns = previous_positive,
      rows = previous_positive >= 15700
    )
  )
t5
```

HIV Testing in Malawi				
Q1 to Q2 2019				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

In the code above, the condition over which the styling will occur is stated in :

WATCH OUT



`locations = cells_body(columns = previous_positive, rows = previous_positive >= 15700)`

Also, note that we can pass more arguments to the `cell_text` function, such as the size and the font of the cells we intend to style.

What if we want to have a two sided condition over the same threshold? Can we have cells with values greater or equal to the threshold styled in green, and simultaneously other cells with values less than the threshold styled in.... cyan?

We absolutely can, we've already done the first part (in the previous code chunk), we just need to add a second condition in a similar manner but in a different `tab_style` statement:

```
t6 <- t5 %>%
  tab_style(
    style = cell_text(
      color = 'cyan'
    ),
    location = cells_body(
      columns = 'previous_positive',
      rows = previous_positive < 15700
    )
  )
t6
```

HIV Testing in Malawi

Q1 to Q2 2019

	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program



Question 1: Conditional Formatting To highlight (in yellow) rows in a `gt` table where the “`hiv_positive`” column exceeds 1,000, which R code snippet should you use?

A.

```

data %>%
  gt() %>%
  tab_style(
    style = cells_body(),
    columns = "Sales",
    conditions = style_number(Sales > 1000, background =
      "yellow")
  )

```

B.

```

data %>%
  gt() %>%
  tab_style(
    style = cells_data(columns = "Sales"),
    conditions = style_number(Sales > 1000, background =
      "yellow")
  )

```

C.



```

data %>%
  gt() %>%
  tab_style(
    style = cell_fill(
      color = "yellow"
    ),
    locations = cells_body(
      columns = "hiv_positive",
      rows = hiv_positive > 1000
    )
  )

```

D.

```

data %>%
  gt() %>%
  tab_style(
    style = cells_data(columns = "Sales"),
    conditions = style_text(Sales > 1000, background =
      "yellow")
  )

```

Question 2: Cell Coloration Fill

Using the `hiv_malawi` data frame, create a `gt` table that displays the total number (`sum`) of “`new_positive`” cases for each “`region`”. Highlight cells with values more than 50 cases in `red` and cells with

less or equal to 50 in green. Complete the missing parts (_____) of this code to achieve this.



```
# Calculate the total_new_pos summary
total_summary <- hiv_malawi %>%
  group_by(_____) %>%
  summarize(total_new_positive = _____)

# Create a gt table and apply cell coloration
summary_table <- total_summary %>%
  gt() %>%
  tab_style(
    style = cell_fill(color = "red"),
    locations = _____ (
      columns = "new_positive",
      rows = _____
    )
  ) %>%
  tab_style(
    style = _____,
    locations = cells_body(
      columns = "new_positive",
      _____ new_positive <= 50
    )
  )
)
```

Fonts and text

Now, we'll enhance the visual appeal of our table's text. For this, we'll use the `gt::tab_style()` function once again.

Let's modify the font and color of the title and the subtitle. We'll select the `Yanone Kaffeesatz` font from Google Fonts, a resource offering a vast array of fonts that can add a unique touch to your table, beyond the standard options in Excel.

To apply these changes, we'll configure the `gt::tab_style()` function as follows:

- The `style` argument is assigned the `cell_text()` function, which houses two other arguments:
 - `font` is assigned the `google_font()` function with our chosen font name.
 - `color` is set to a hexadecimal color code that corresponds to our desired text color.
- The `locations` argument is assigned the `cells_title()` function:

- We specify title and subtitle within the groups argument using vector notation `c(...)`.

To specifically modify the title or subtitle, you can use `locations = cells_title(groups = "title")` or `locations = cells_title(groups = "subtitle")`, respectively, without the need for `c(...)`.

SIDE NOTE



Using lists to pass arguments in gt: Lists in R are an integral part of the language and are extremely versatile. A list can contain elements of different types (numbers, strings, vectors, and even other lists) and each element can be accessed by its index. In the context of our {gt} table, we use lists to group together style properties (with the style argument) and to specify multiple locations in the table where these styles should be applied (with the locations argument).

Using Hexadecimal Color Codes: Colors in many programming languages, including R, can be specified using hexadecimal color codes. These codes start with a hash symbol (#) and are followed by six hexadecimal digits. The first two digits represent the red component, the next two represent the green component, and the last two represent the blue component. So, when we set `color = "#00353f"`, we're specifying a color that has no red, a bit of green, and a good amount of blue, which results in a deep blue color. This allows us to have precise control over the colors we use in our tables.

```
t7 <- t4 %>%
  tab_style(
    style = cell_text(
      font = google_font(name = 'Yanone Kaffeesatz'),
      color = "#00353f"
    ),
    locations = cells_title(groups = c("title", "subtitle"))
  )
t7
```

HIV Testing in Malawi				
	Q1 to Q2 2019			
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

We can extend our customization to include the labels for columns, spanners, and stubs, as well as the source note. Within the `locations` argument, we'll supply a list indicating the specific locations for these changes. For a comprehensive understanding of the locations, please refer to Appendix (List 1).

```
t8 <- t7 %>%
  tab_style(
    style = list(
      cell_text(
        font = google_font(name = "Montserrat"),
        color = "#00353f"
      )
    ),
    locations = list(
      cells_column_labels(columns = everything()), # select every column
      cells_column_spanners(spanners = everything()), # select all spanners
      cells_source_notes(),
      cells_stub()
    )
  )
t8
```

HIV Testing in Malawi				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

If you want to change the fill background of the title, you can do so by adjusting the `locations` argument to point at `cells_title(groups = "title")`. Here's how you could do it:

SIDE NOTE



```
t9 <- t7 %>%
  tab_style(
    style = cell_fill(background = "#ffffff"),
    locations = cells_title(groups = "title")
  )
t9
```

In this code, `cell_fill(background = "#ffffff")` changes the background color to white, and `locations = cells_title(groups = "title")` applies this change specifically to the title of the table.

PRACTICE



Question 2: Fonts and Text Which R code snippet allows you to change the font size of the footnote text in a `gt` table?

A.

```
data %>%
  gt() %>%
  tab_header(font.size = px(16))
```

B.

```
data %>%
  gt() %>%
  tab_style(
    style = cell_text(
      size = 16
    ),
    locations = cells_footnotes()
  )
```

C.



```
data %>%
  gt() %>%
  tab_style(
    style = cells_header(),
    css = "font-size: 16px;"
  )
```

D.

```
data %>%
  gt() %>%
  tab_style(
    style = cells_header(),
    css = "font-size: 16;"
  )
```

Borders

In `gt` it's also possible to draw borders in the tables to help the end user focus on specific area in the table. In order to add borders to a `gt` table we will use, again the, `tab_style` function and, again, specify the `style` and `locations` argument. The only difference now is that we will use the `cell_borders` helper function and assign it to the `style` argument. Here's how:

Let's first add a vertical line:

```
t10 <- t8 %>%
  tab_style(
    style = cell_borders( # we are adding a border
      sides = "left",     # to the left of the selected location
      color = "#45785e",   # with a dark green color
      weight = px(5)       # and five pixels of thickness
    ),
    locations = cells_body(columns = 2) # add this border line to the left of
                                         column 2
  )
t10
```

HIV Testing in Malawi				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Now let's add another pink horizontal border line:

```
t11 <- t10 %>%
  tab_style(
    style = cell_borders( # we are adding a border line
      sides = "bottom",    # to the bottom of the selected location
      color = "#45785e",   # with a pink color
      weight = px(5)       # and five pixels of thickness
    ),
    locations = list(
      cells_column_labels(columns = everything()), # add this border line to
                                                 the bottom of the column labels
      cells_stubhead()                           # and to the stubhead
    )
  )
t11
```

HIV Testing in Malawi				
	New tests		Previous tests	
	Positive	Negative	Positive	Negative
2019 Q1	6199	284694	14816	6595
2019 Q2	6132	282249	15101	5605
2019 Q3	5907	300529	15799	6491
2019 Q4	5646	291622	15700	6293

Source: Malawi HIV Program

Question 4: Borders To add a solid border around the entire `gt` table, which R code snippet should you use?

Hint : we can use a function that sets options for the entirety of the table, just like the `theme` function for the `ggplot` package.

A.

```
data %>%
  gt() %>%
  tab_options(table.border.top.style = "solid")
```

CHALLENGE



B.

```
data %>%
  gt() %>%
  tab_options(table.border.style = "solid")
```

C.

```
data %>%
  gt() %>%
  tab_style(
    style = cells_table(),
    css = "border: 1px solid black;"
  )
```

D.

CHALLENGE



```
data %>%
  gt() %>%
  tab_style(
    style = cells_body(),
    css = "border: 1px solid black;"
  )
```

Answer Key

1.C

2.

```
# Solutions are where the numbered lines are

# Calculate the total_new_pos summary
total_summary <- hiv_malawi %>%
  group_by(region) %>% ##1
  summarize(total_new_positive = new_positive) ##2

# Create a gt table and apply cell coloration
summary_table <- total_summary %>%
  gt() %>% ##3
  tab_style(
    style = cell_fill(color = "red"),
    locations = cells_body( ##4
      columns = "new_positive",
      rows = new_positive >= 50 ##5
    )
  ) %>%
  tab_style(
    style = cell_fill(color = "green"), ##6
    locations = cells_body(
      columns = "new_positive",
      rows = new_positive < 50 ##7
    )
  )
```

3.B

4.B

Contributors

The following team members contributed to this lesson:



BENNOUR HSIN

Data Science Education Officer
Data Visualization enthusiast



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

External resources and packages

- The definite cookbook of `gt` by Tom Mock : <https://themockup.blog/static/resources/gt-cookbook.html#introduction>
- the Grammar of Table article : <https://themockup.blog/posts/2020-05-16-gt-a-grammar-of-tables/#add-titles>
- official `gt` documentation page : <https://gt.rstudio.com/articles/intro-creating-gt-tables.html>
- Create Awesome HTML Table with `knitr::kable` and `kableExtra` book by Hao Zhu : https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html#Overview

Appendix

The `{gt}` package in R provides a variety of functions to specify locations within a table where certain styles or formatting should be applied. Here are some of them:

- `cells_body()`: This function targets cells within the body of the table. You can further specify rows and columns to target a subset of the body.
- `cells_column_labels()`: This function targets the cells that contain the column labels.

- `cells_column_spanners()`: This function targets cells that span multiple columns.
- `cells_footnotes()`: This function targets cells that contain footnotes.
- `cells_grand_summary()`: This function targets cells that contain grand summary rows.
- `cells_group()`: This function targets cells that contain group label rows.
- `cells_row_groups()`: This function targets cells that contain row group label rows.
- `cells_source_notes()`: This function targets cells that contain source notes.
- `cells_stub()`: This function targets cells in the table stub (the labels in the first column of the table).
- `cells_stubhead()`: This function targets the cell that contains the stubhead.
- `cells_stub_summary()`: This function targets cells that contain stub summary rows.
- `cells_title()` : This function targets cells that contain the table title and subtitle.
- `cells_summary()`: This function targets cells that contain summary rows.

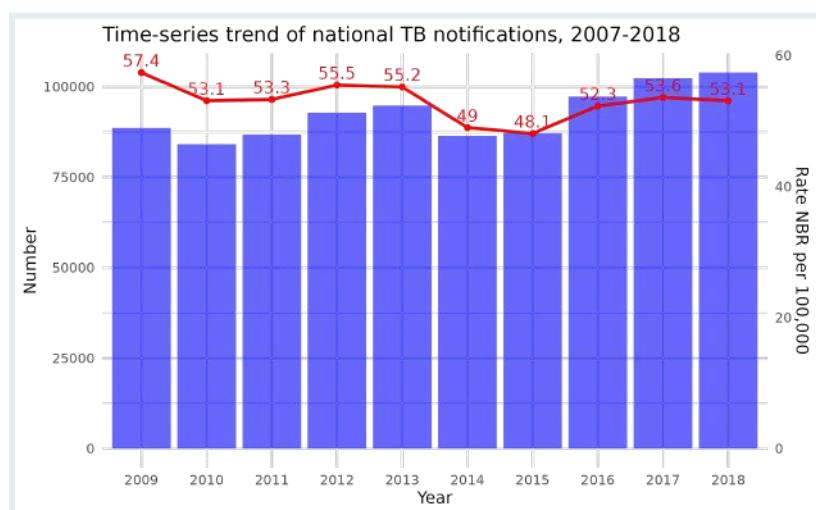
These functions can be used in the `locations` argument of the `tab_style()` function to apply specific styles to different parts of the table.

Epidemiological Time Series Visualization

Introduction
Learning Objectives
Packages
Intro to Line Graphs for Time Series Data
Data Preparation: Aggregating and Pivoting
A Basic Grouped Line Graph
Aesthetic Improvements to Line Graphs
Reducing Label Frequency
Alternating Labels
<code>ggrepel::geom_text_repel()</code>
Customizing the Color Palette
Adding Plot Annotations
Plotting Confidence Intervals with <code>geom_ribbon()</code>
Smoothing Noisy Trends
Creating an Incidence Table from a Linelist
Smoothing with <code>geom_smooth()</code>
Smoothing by Aggregating
Smoothing with Rolling Averages
Secondary Axes
Understanding the Concept of a Secondary Y-Axis
Creating a Plot with a Secondary Y-Axis
Wrap up
Answer Key
Contributors

Introduction

By analyzing time series data—observations made sequentially over time—epidemiologists can spot trends and patterns in disease outbreaks, and inform decision-making for improved health outcomes. In this lesson, we explore using `ggplot` and the `tidyverse` to visualize time series data and effectively communicate insights.



Learning Objectives

By the end of this lesson you will be able to:

- Reshape time series data for plotting with `pivot_longer()`
- Create line graphs in `ggplot2` mapping time to `x` and values to `y`
- Enhance line graph aesthetics with techniques like custom labels, color palettes, annotations
- Visualize confidence intervals with `geom_ribbon()`
- Highlight patterns in noisy data using smoothing and aggregation
- Compare time series with distinct scales using dual axes and `sec_axis()`

Packages

Install and load the necessary packages with the following code chunk:

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(dplyr, ggplot2, tidyr, lubridate, outbreaks, scales, ggrepel,
               ggthemes, zoo, here)
options(scipen=999)
```



Setting `options(scipen = 999)` prevents the use of scientific notation in our plots, making long numbers easier to read and interpret.

Intro to Line Graphs for Time Series Data

To get started with visualizing time series data, we'll examine the dynamics of tuberculosis (TB) notifications in Australia over time, comparing notifications in urban and rural areas. The source dataset can be accessed [here](#)

VOCAB



Notifications are a technical term for the number of cases of a disease that are reported to public health authorities.

Data Preparation: Aggregating and Pivoting

Let's start by loading and inspecting the data:

```
tb_data_aus <- read_csv(here::here("data/aus_tb_notifs.csv"))
head(tb_data_aus)
```

```
## # A tibble: 6 × 3
##   period rural urban
##   <chr>   <dbl> <dbl>
## 1 1993Q1     6     51
## 2 1993Q2    11     52
## 3 1993Q3    13     67
## 4 1993Q4    14     82
## 5 1994Q1    16     63
## 6 1994Q2    15     65
```

This dataset includes the columns `period` (time in quarterly format, e.g., '1993Q1'), `rural` (cases in rural areas), and `urban` (cases in urban areas).

We would like to visualize the number of *annual* TB notifications in urban and rural areas, but the data is currently in a quarterly format. So, we need to aggregate the data by year.

Let's start by extracting the year from the `period` column. We do this using the `str_sub()` function from the `stringr` package:

```
tb_data_aus %>%
  mutate(year = str_sub(period, 1, 4)) %>%
  # convert back to numeric
  mutate(year = as.numeric(year))
```

```
## # A tibble: 120 × 4
##   period rural urban   year
##   <chr>   <dbl> <dbl> <dbl>
## 1 1993Q1     6     51  1993
## 2 1993Q2    11     52  1993
## 3 1993Q3    13     67  1993
## 4 1993Q4    14     82  1993
## 5 1994Q1    16     63  1994
## 6 1994Q2    15     65  1994
## 7 1994Q3    18     60  1994
## 8 1994Q4    25     71  1994
## 9 1995Q1     7     77  1995
## 10 1995Q2    9     52  1995
## # i 110 more rows
```

The `str_sub()` function takes three arguments: the string we want to extract from, the starting position, and the ending position. In this case, we want to extract the first

four characters from the `period` column, which correspond to the year.

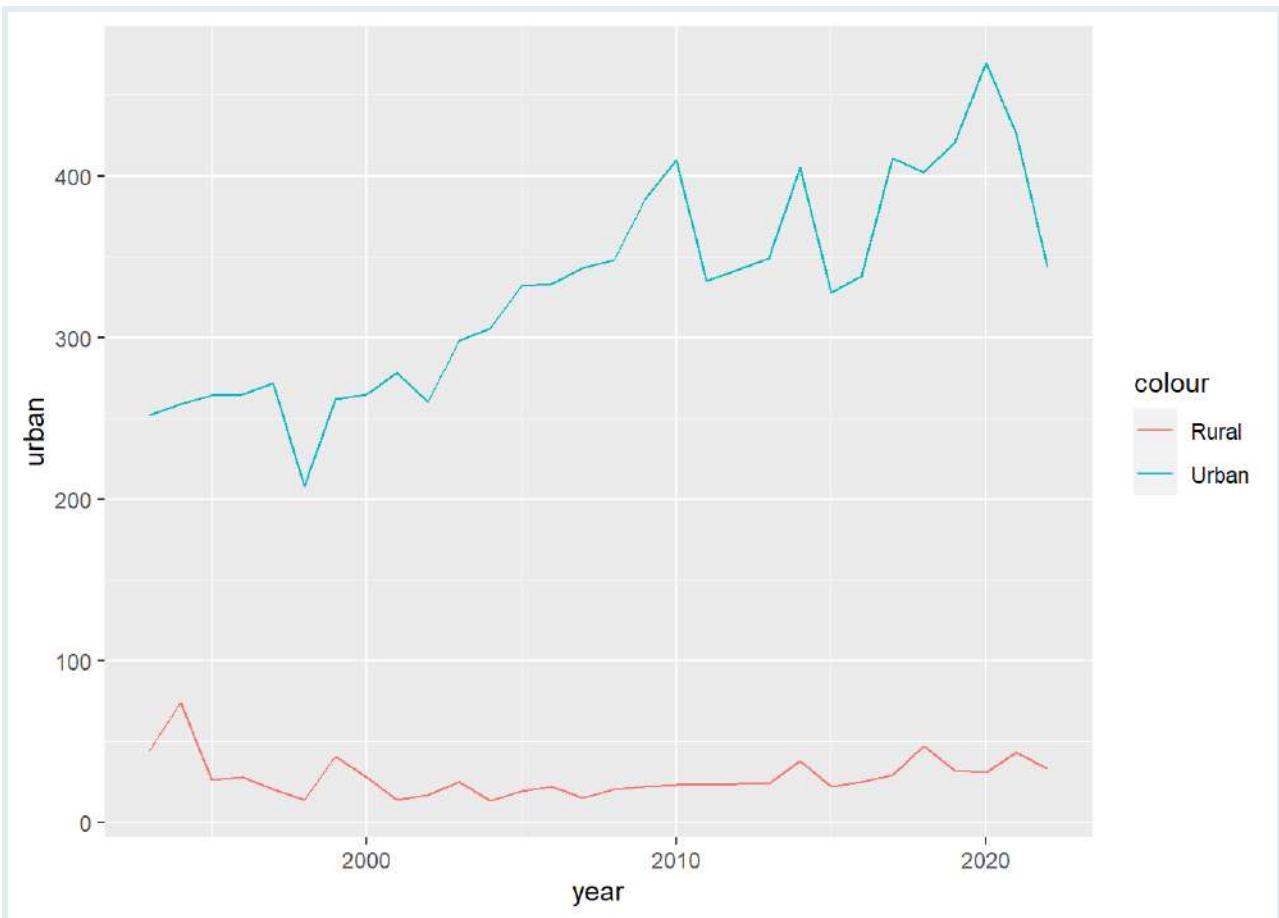
Now, let's aggregate the data by year. We can do this using the `group_by()` and `summarise()` functions:

```
annual_data_aus <- tb_data_aus %>%
  mutate(year = str_sub(period, 1, 4)) %>%
  mutate(year = as.numeric(year)) %>%
  # group by year
  group_by(year) %>%
  # sum the number of cases in each year
  summarise(rural = sum(rural),
            urban = sum(urban))
annual_data_aus
```

```
## # A tibble: 30 × 3
##       year rural urban
##   <dbl> <dbl> <dbl>
## 1 1993     44    252
## 2 1994     74    259
## 3 1995     26    264
## 4 1996     28    265
## 5 1997     20    272
## 6 1998     14    208
## 7 1999     41    262
## 8 2000     28    265
## 9 2001     14    278
## 10 2002    17    260
## # i 20 more rows
```

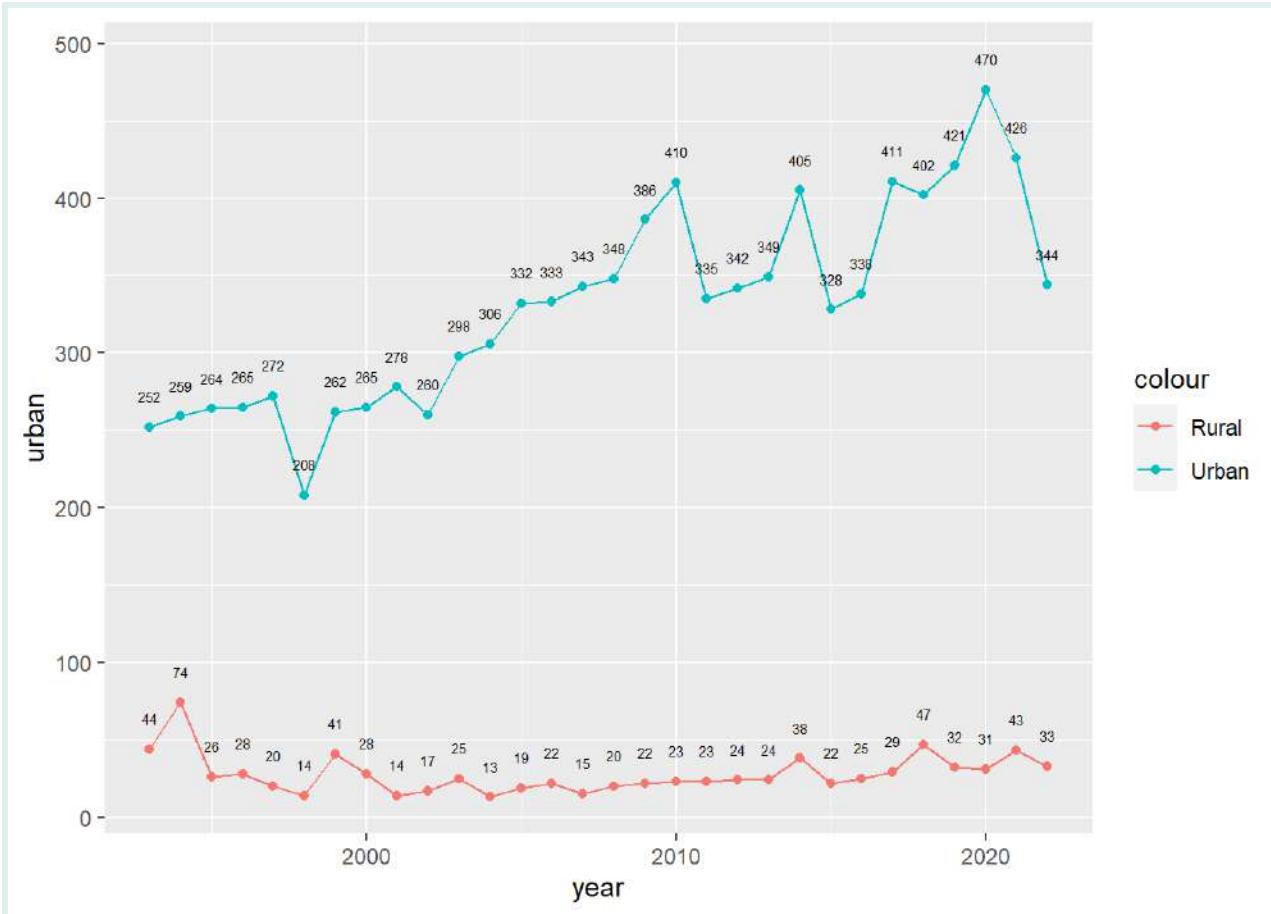
Now that we seem to have the data in the format we want, let's make an initial line plot:

```
ggplot(annual_data_aus, aes(x = year)) +
  geom_line(aes(y = urban, colour = "Urban")) +
  geom_line(aes(y = rural, colour = "Rural"))
```



This is an informative plot, however, there is some unnecessary code duplication, though you may not yet realize it. This will become clearer if we try to add additional geoms, such as points, or text:

```
ggplot(annual_data_aus, aes(x = year)) +  
  geom_line(aes(y = urban, colour = "Urban")) +  
  geom_line(aes(y = rural, colour = "Rural")) +  
  geom_point(aes(y = urban, colour = "Urban")) +  
  geom_point(aes(y = rural, colour = "Rural")) +  
  geom_text(aes(y = urban, label = urban), size = 2, nudge_y = 20) +  
  geom_text(aes(y = rural, label = rural), size = 2, nudge_y = 20)
```



As you can see, we have to repeat the same lines of code for each geom. This is not only tedious, but also makes the code more difficult to read and interpret. If we had more than two categories, as often happens, it would be even more cumbersome.

Fortunately, there is a better way. We can use the `pivot_longer()` function from the `{tidyverse}` package to reshape the data into a format that is more suitable for plotting:

```
# Using tidyverse's `pivot_longer` to reshape the data
annual_data_aus %>%
  pivot_longer(cols = c("urban", "rural"))
```

```
## # A tibble: 60 × 3
##       year name   value
##   <dbl> <chr> <dbl>
## 1 1993  urban  252
## 2 1993  rural   44
## 3 1994  urban  259
## 4 1994  rural   74
## 5 1995  urban  264
## 6 1995  rural   26
## 7 1996  urban  265
## 8 1996  rural   28
## 9 1997  urban  272
```

```
## 10 1997 rural 20
## # i 50 more rows
```

The code above has converted the data from a “wide” format to a “long” format. This is a more suitable format for plotting, as it allows us to map a specific column to the colour aesthetic.

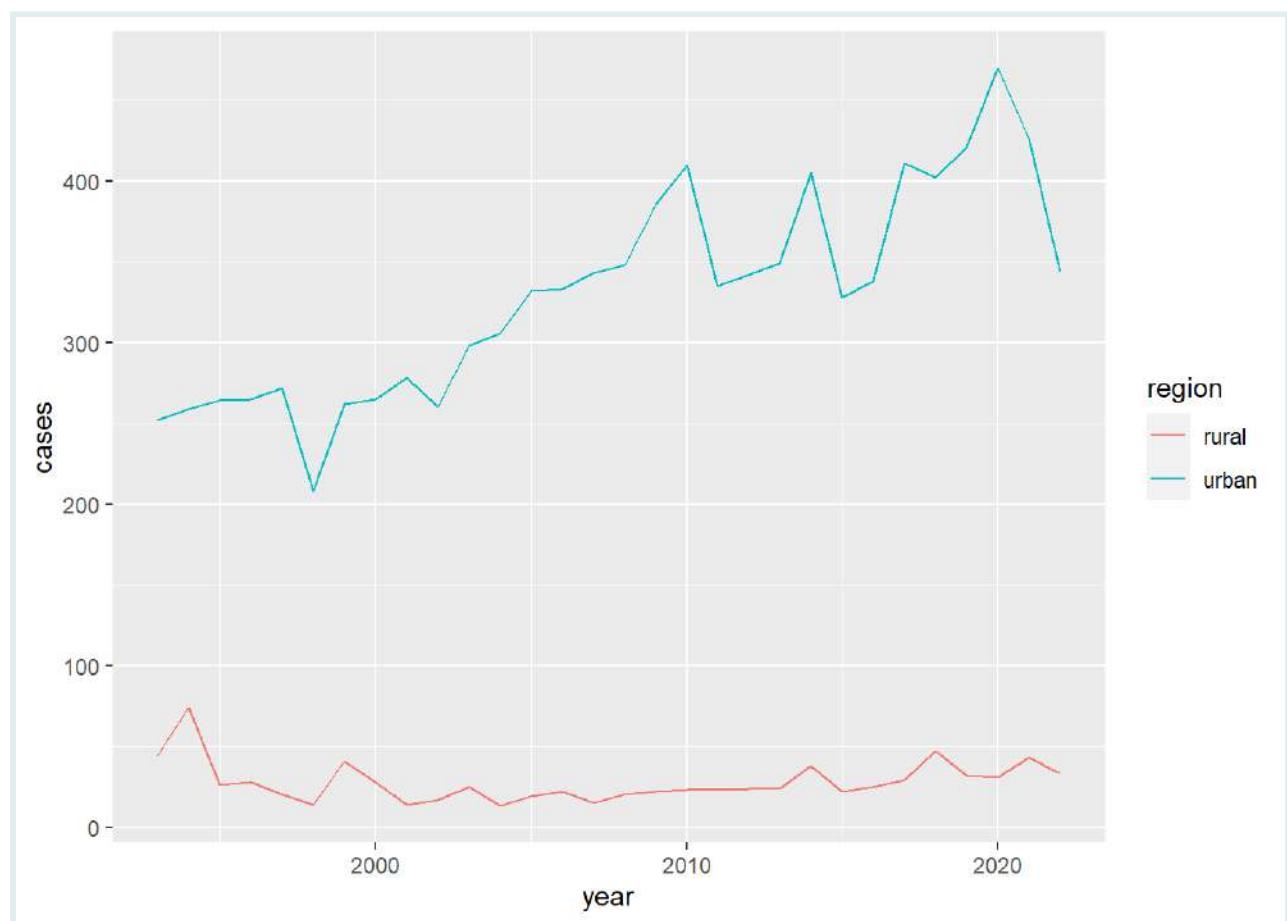
Before we plot this long dataset, let’s rename the columns to make them more informative:

```
aus_long <- annual_data_aus %>%
  pivot_longer(cols = c("urban", "rural")) %>%
  rename(region = name, cases = value)
```

A Basic Grouped Line Graph

We’re ready to plot the data again. We map the colour and group aesthetics to the `region` column, which contains the two categories of interest: urban and rural.

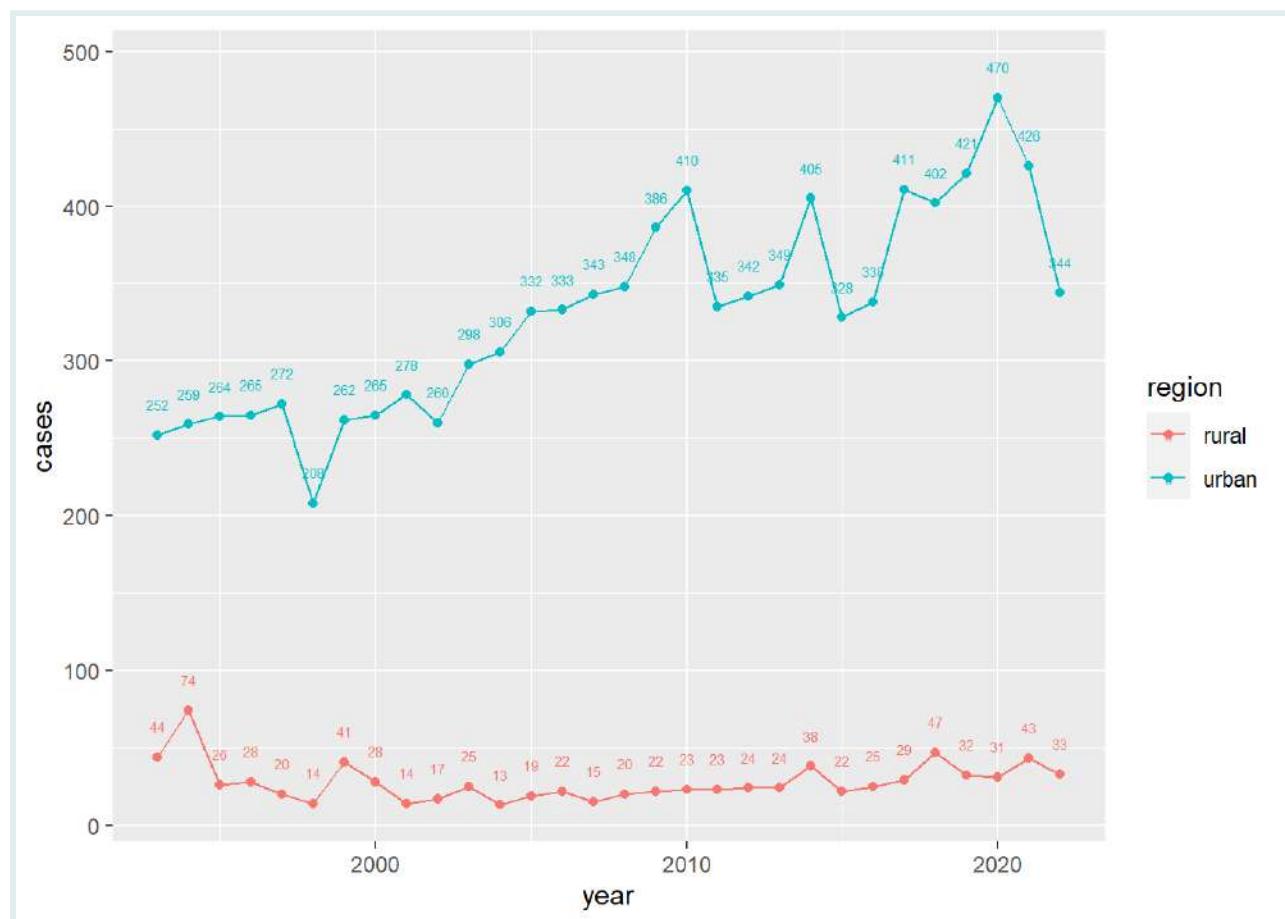
```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line()
```



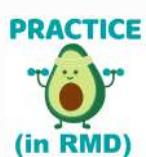
The plotting code is now more concise, thanks to the pivoting operation executed previously.

We can now also add points and text labels with significantly less code:

```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +  
  geom_line() +  
  geom_point() +  
  geom_text(aes(label = cases), size = 2, nudge_y = 20)
```



Great! We now have a clear view of trends in annual TB case notifications in rural and urban areas over time. However, there are still some aesthetic improvements we can make; we will cover these in the next section.



Q: Reshaping and Plotting TB Data

Consider the Benin dataset shown below, which contains information about bacteriologically confirmed and clinically diagnosed TB cases for several years in Benin. (The data was sourced from a paper [here](#))

```
tb_data_benin <- read_csv(here("data/benin_tb_notifs.csv"))
tb_data_benin
```

```
## # A tibble: 15 × 3
##   year new_clindx new_labconf
##   <dbl>     <dbl>      <dbl>
## 1 2000        289      2280
## 2 2001        286      2289
## 3 2002        338      2428
## 4 2003        350      2449
## 5 2004        330      2577
## 6 2005        346      2731
## 7 2006        261      2950
## 8 2007        294      2755
## 9 2008        239      2983
## 10 2009       279      2950
## 11 2010       307      2958
## 12 2011       346      3326
## 13 2012       277      3086
## 14 2013       285      3219
## 15 2014       318      3062
```



Reshape the dataset using `pivot_longer()`, then create a plot with two lines, one for each type of TB case diagnosis. Add points and text labels to the plot.

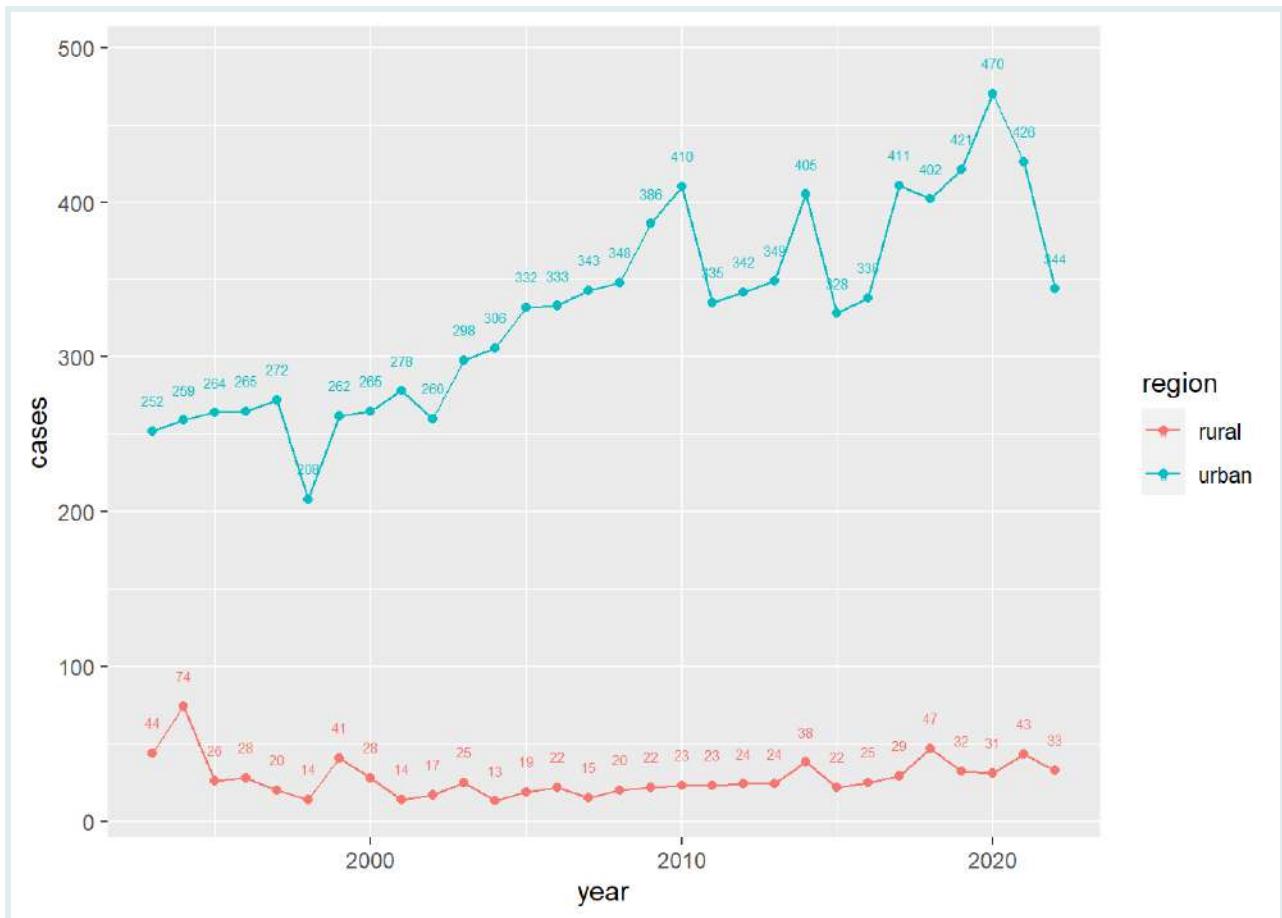
Aesthetic Improvements to Line Graphs

In this section, we will focus on improving the aesthetics of time series line graphs to enhance their clarity and visual appeal.

Reducing Label Frequency

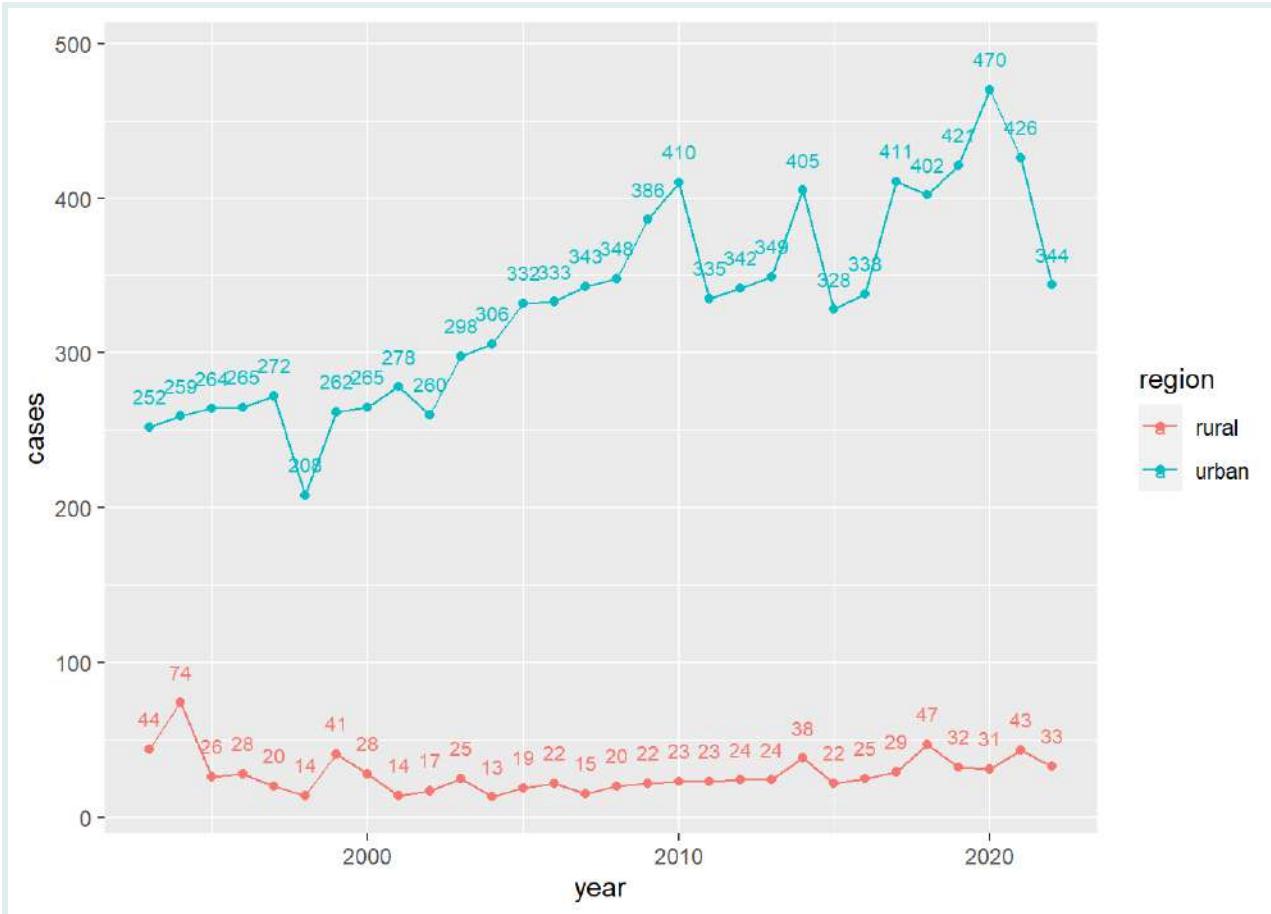
Where we last left off, we had a plot that looked like this:

```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text(aes(label = cases), size = 2, nudge_y = 20)
```



One problem with this plot is that the text labels are a bit too small. Such tiny labels are not ideal for a public-facing plot, as they are difficult to read. However, if we increase the label size, the labels will start to overlap, as shown below:

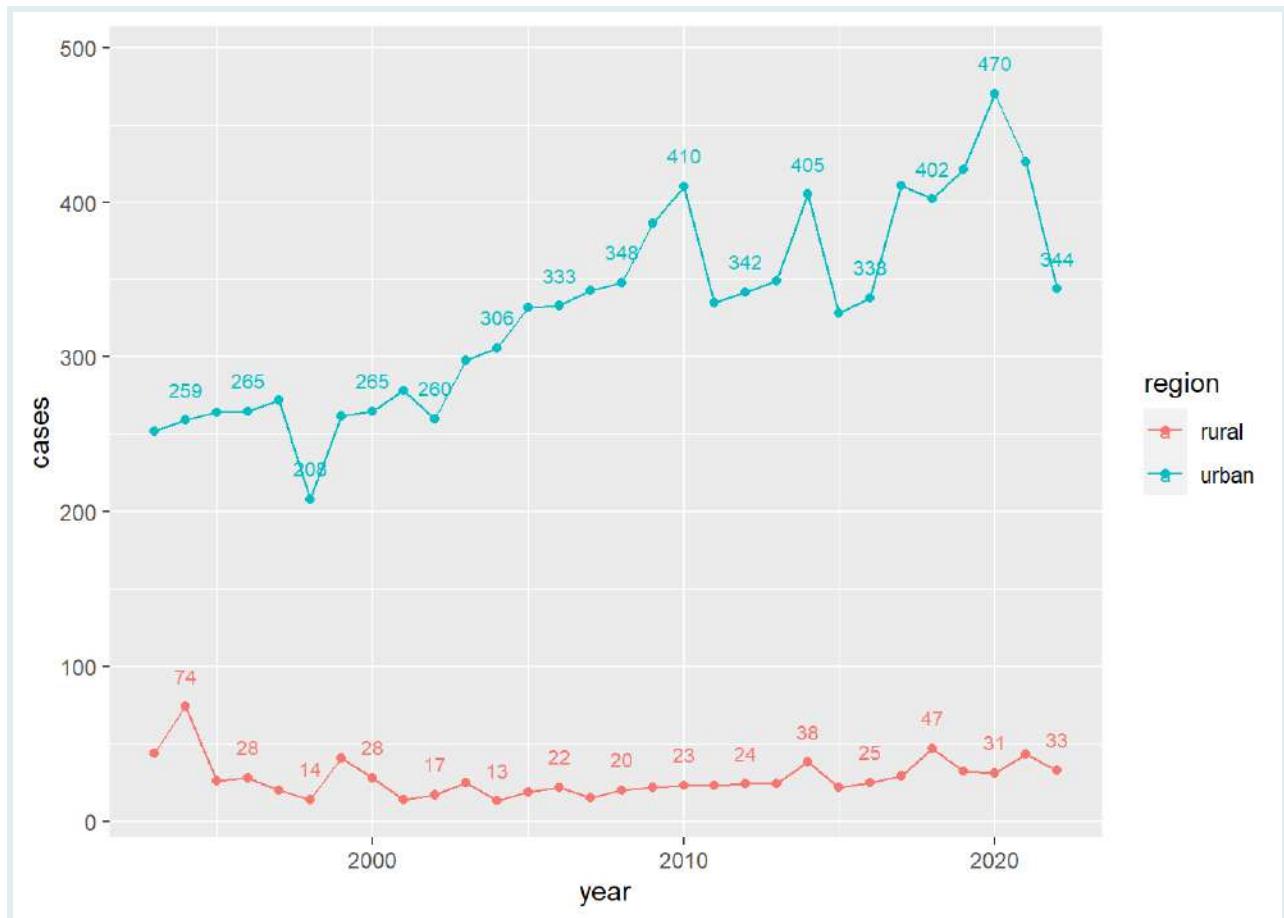
```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text(aes(label = cases), size = 2.8, nudge_y = 20)
```



To avoid this clutter, a handy technique is to display labels for only certain years. To do this, we can give a custom dataset to the `geom_text()` function. In this case, we will create a dataset that contains only the even years:

```
even_years <- aus_long %>%
  filter(year %% 2 == 0) # Keep only years that are multiples of 2

ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text(data = even_years, aes(label = cases),
            size = 2.8, nudge_y = 20)
```



Great, now we have larger labels and they do not overlap.

Alternating Labels

While the plot above is an improvement, it would be even better if we could display the labels for *all* years. We can do this by displaying the labels for the even years above the data points, and the labels for the odd years below the data points.

Including many data points (within reason) in your plots is helpful for public health officials; as they can pull quick numbers from the plot when trying to make decisions, without needing to look at the reference datasets.

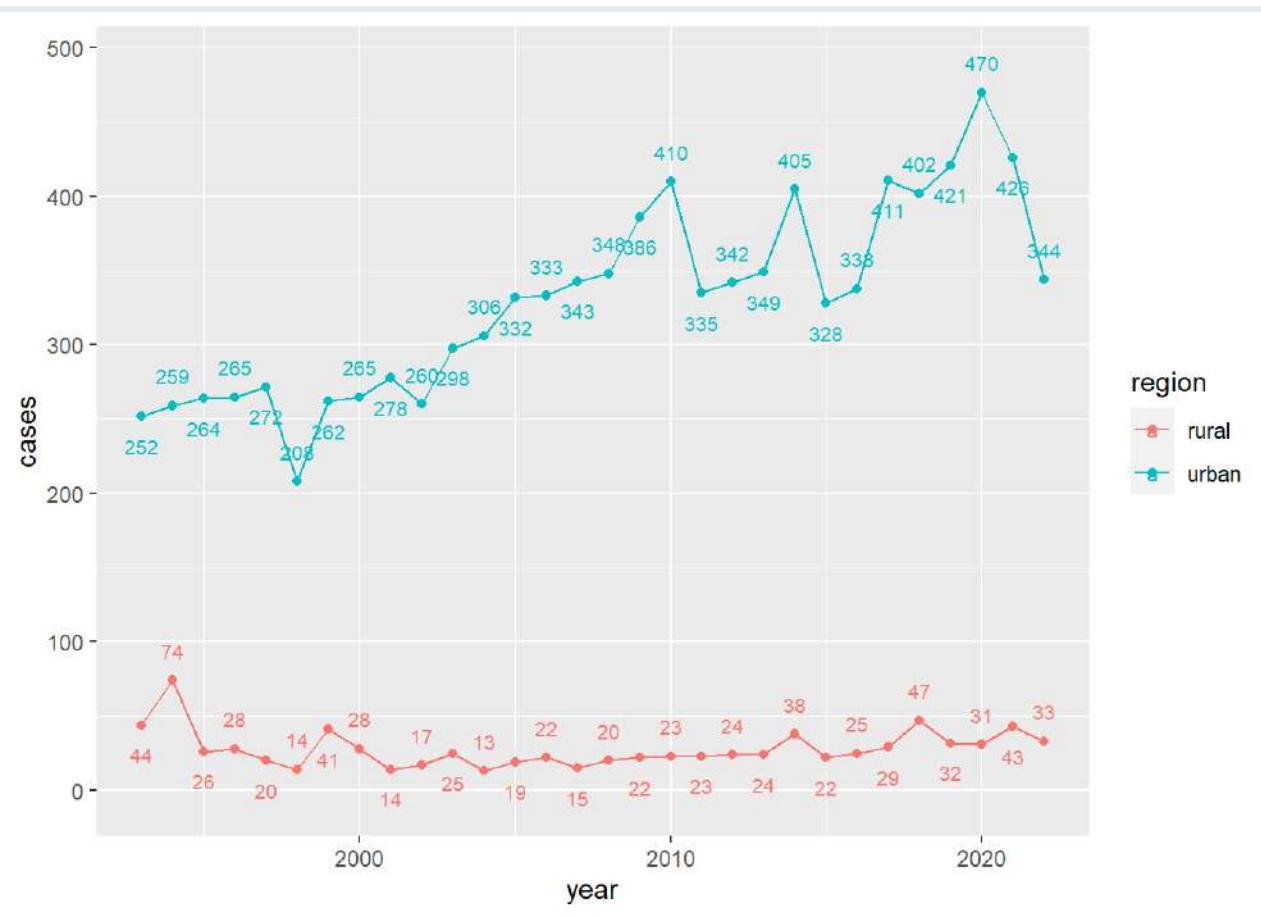
To address this, let's create a filtered dataset for odd years, and then use `geom_text()` twice, once for each filtered dataset.

```

odd_years <- aus_long %>%
  filter(year %% 2 != 0) # Keep only years that are NOT multiples of 2

ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text(data = even_years, aes(label = cases),
            nudge_y = 20, size = 2.8) +
  geom_text(data = odd_years, aes(label = cases),
            nudge_y = -20, size = 2.8)

```



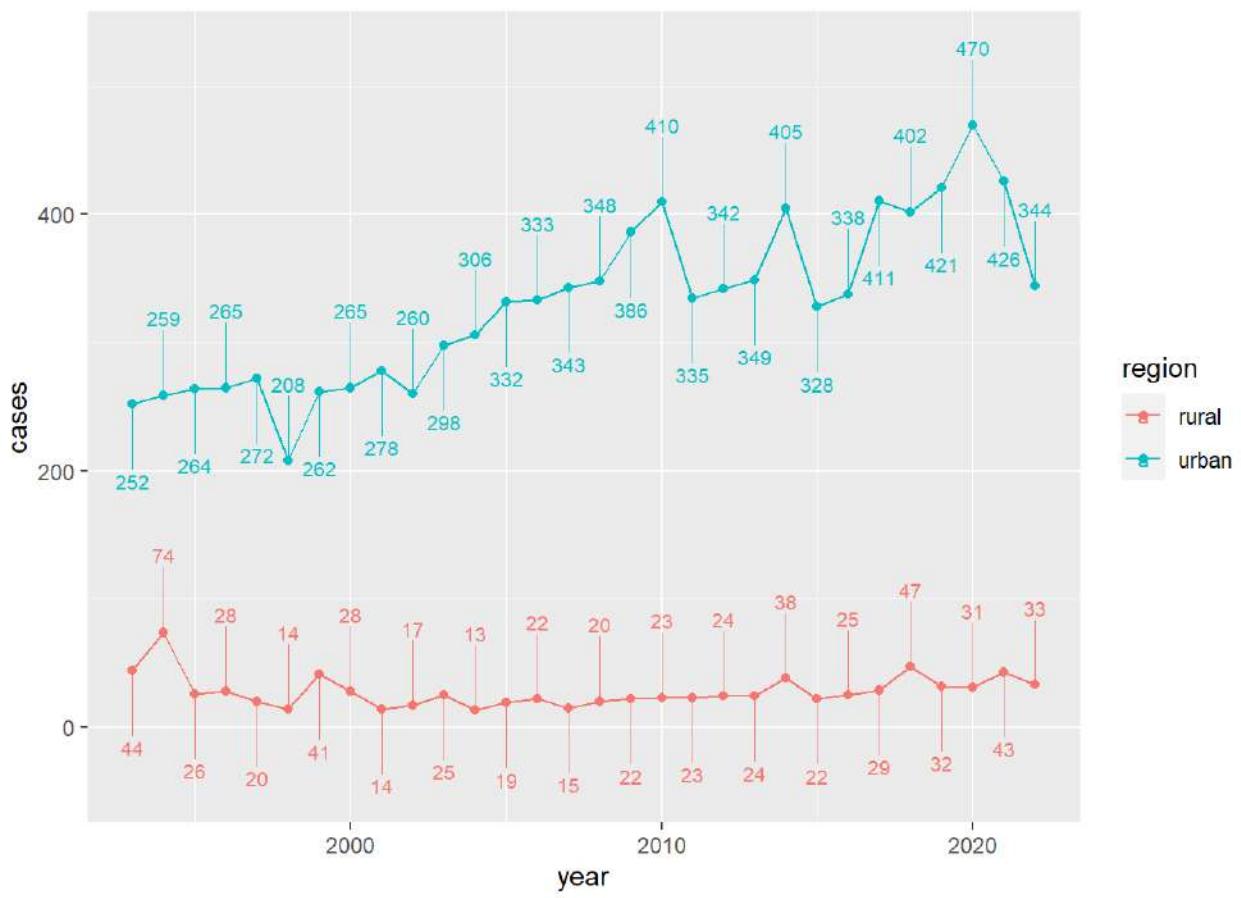
`ggrepel::geom_text_repel()`

The plot above is clear, but there is still some overlap between the labels and the line.

To further enhance clarity, we can use the `geom_text_repel()` from the `{ggrepel}` package.

This function nudges individual labels to prevent overlap, and connects labels to their data points with lines, making it easier to see which label corresponds to which data point, and allowing us to increase the distance between the labels and the data points.

```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text_repel(data = even_years, aes(label = cases),
                  nudge_y = 60, size = 2.8, segment.size = 0.1) +
  geom_text_repel(data = odd_years, aes(label = cases),
                  nudge_y = -60, size = 2.8, segment.size = 0.1)
```



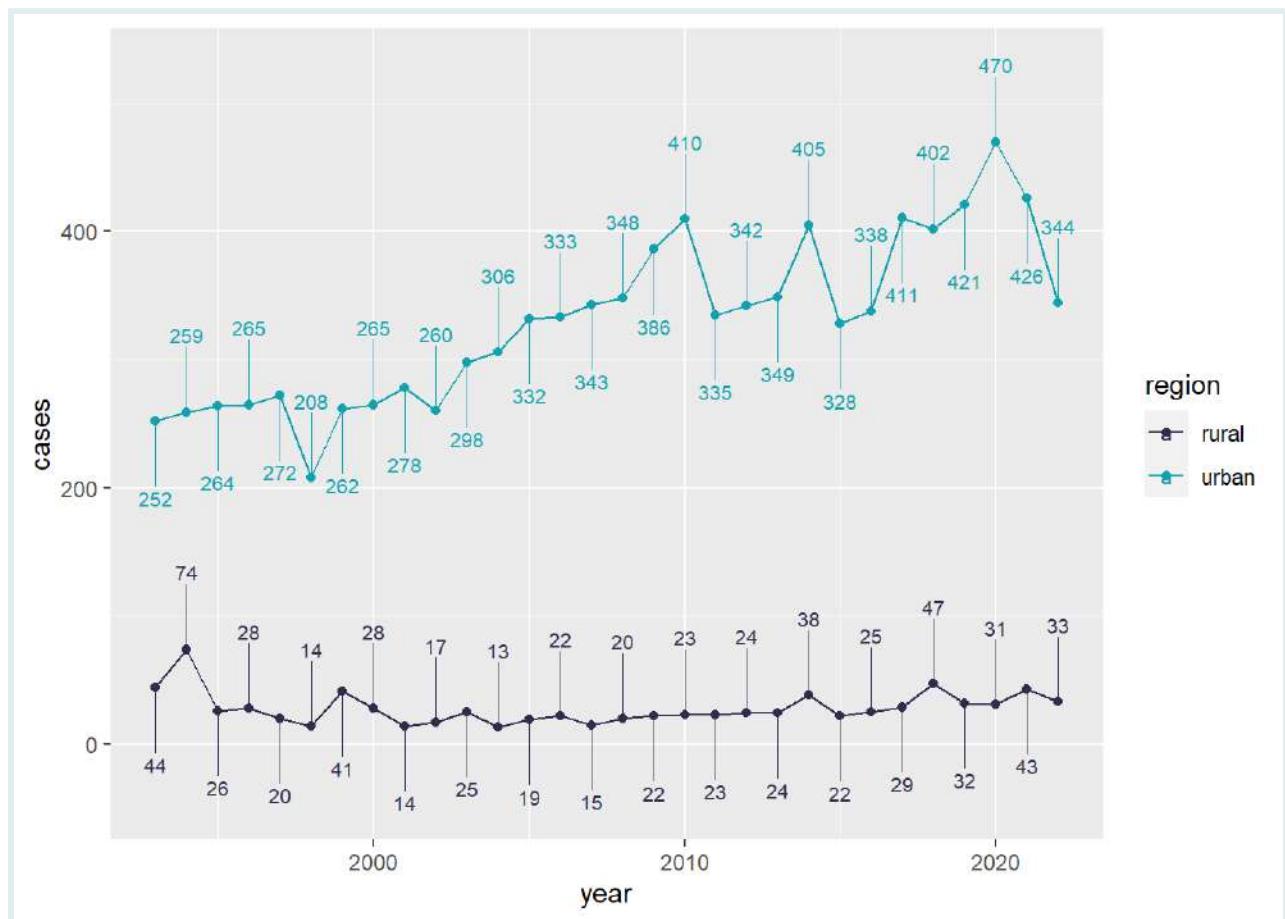
As you can see, the function `geom_text_repel()` takes basically the same arguments as `geom_text()`. The extra argument, `segment.size`, controls the width of the lines connecting the labels to the data points.

Customizing the Color Palette

It is often useful to customize the color palette of your plots, so that they match, for example, the color scheme of your organization.

We can customize the colors of the lines using the `scale_color_manual()` function. Below, we specify two colors, one for each region:

```
ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line() +
  geom_point() +
  geom_text_repel(data = even_years, aes(label = cases),
                  nudge_y = 60, size = 2.8, segment.size = 0.1) +
  geom_text_repel(data = odd_years, aes(label = cases),
                  nudge_y = -60, size = 2.8, segment.size = 0.1) +
  scale_color_manual(values = c("urban" = "#0fa3b1",
                               "rural" = "#2F2C4E"))
```



Success!

Adding Plot Annotations

Finally, let's add a set of finish touches. We'll annotate the plot with appropriate titles, axis labels, and captions, and modify the theme:

```

ggplot(aus_long, aes(x = year, y = cases, colour = region, group = region)) +
  geom_line(linewidth = 1) +
  geom_text_repel(data = even_years, aes(label = cases),
                  nudge_y = 60, size = 2.8, segment.size = 0.08) +
  geom_text_repel(data = odd_years, aes(label = cases),
                  nudge_y = -50, size = 2.8, segment.size = 0.08) +
  scale_color_manual(values = c("urban" = "#0fa3b1", "rural" = "#2F2C4E")) +
  labs(title = "Tuberculosis Notifications in Australia",
       subtitle = "1993-2022",
       caption = "Source: Victoria state Government Department of Health",
       x = "Year",
       color = "Region") +
  ggthemes::theme_few() +
  theme(legend.position = "right")

```

Tuberculosis Notifications in Australia

1993-2022



Source: Victoria state Government Department of Health

This covers some options for improving line chart aesthetics! Feel free to further tweak and adjust visuals based on your specific analysis needs.



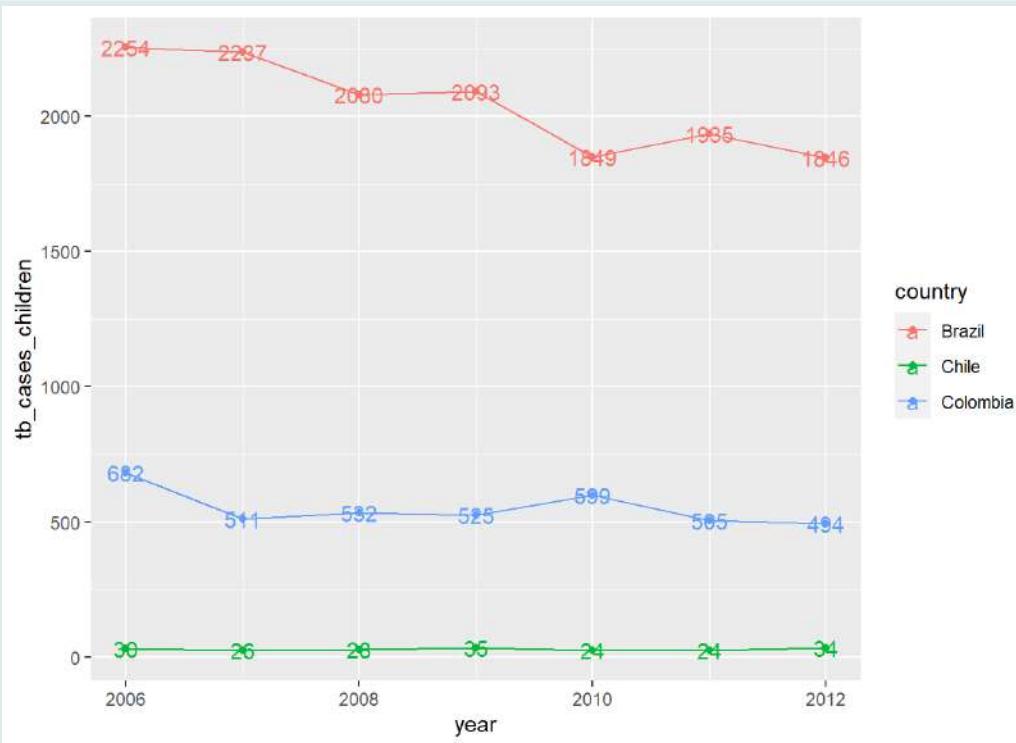
We've transformed our plot into a visually appealing, easy-to-read representation of TB notification trends in Australia. We balanced the need for detailed information with a clear presentation, making our plot both informative and accessible.

Q: Aesthetic improvements

Consider the following plot, which shows the number of child TB cases in three countries over time:

```
tb_child_cases_southam <- tidyverse::who2 %>%
  transmute(country, year,
            tb_cases_children = sp_m_014 + sp_f_014 + sn_m_014 +
            sn_f_014) %>%
  filter(country %in% c("Brazil", "Colombia", "Chile")) %>%
  filter(!is.na(tb_cases_children))

tb_child_cases_southam %>%
  ggplot(aes(x = year, y = tb_cases_children, color = country)) +
  geom_line() +
  geom_point() +
  geom_text(aes(label = tb_cases_children))
```



Build on this plot, implementing the following improvements:

- Set the `geom_text` labels to alternate above and below the lines, similar to the example we saw above.



- Use the following color palette `c("#212738", "#F97068", "#067BC2")`
- Apply `theme_classic()`
- Add a title, subtitle, and caption to provide context and information about the data. (You can type `?tidyverse::who2` into the console to learn more about the data source.)

Plotting Confidence Intervals with `geom_ribbon()`

In time series visualizations, it is often important to plot confidence intervals to indicate the level of uncertainty in your data.

We will demonstrate how to do this using a dataset on new HIV infections in Brazil, which includes estimated numbers for male and female cases along with confidence intervals. The dataset is sourced from the World Health Organization (WHO) and can be accessed [here](#).

Let's start by loading and inspecting the dataset:

```
hiv_data_brazil <-  
  rio::import(here("data/new_hiv_infections_gho.xlsx"),  
             sheet = "Brazil") %>%  
  as_tibble() %>%  
  janitor::clean_names()  
hiv_data_brazil
```

```
## # A tibble: 89 × 5  
##   continent country year sex      new_hiv_cases  
##   <chr>     <chr>  <dbl> <chr>    <chr>  
## 1 Americas   Brazil   2022 Female  13 000 [12 000 – 15 0...  
## 2 Americas   Brazil   2022 Male    37 000 [35 000 – 40 0...  
## 3 Americas   Brazil   2022 Both sexes 51 000 [47 000 – 54 0...  
## 4 Americas   Brazil   2021 Female  14 000 [12 000 – 15 0...  
## 5 Americas   Brazil   2021 Male    37 000 [34 000 – 39 0...  
## 6 Americas   Brazil   2021 Both sexes 50 000 [47 000 – 53 0...  
## 7 Americas   Brazil   2020 Female  14 000 [13 000 – 15 0...  
## 8 Americas   Brazil   2020 Male    35 000 [32 000 – 37 0...  
## 9 Americas   Brazil   2020 Both sexes 49 000 [46 000 – 52 0...  
## 10 Americas  Brazil   2019 Female  14 000 [13 000 – 15 0...  
## # i 79 more rows
```

We can see that the `new_hiv_cases` column contains both the number of cases and the corresponding confidence intervals in square brackets. This format cannot be directly used for plotting, so we will need to extract them into pure numeric forms.

First, to separate these values, we can use the `separate()` function from the `{tidyverse}` package:

```
hiv_data_brazil %>%  
  separate(new_hiv_cases,  
    into = c("cases", "cases_lower", "cases_upper"),  
    sep = "\\[| -")
```

```
## # A tibble: 89 × 7  
##   continent country  year sex      cases  cases_lower  
##   <chr>     <chr>   <dbl> <chr>    <chr>    <chr>  
## 1 Americas  Brazil    2022 Female  "13 000 " "12 000 "  
## 2 Americas  Brazil    2022 Male    "37 000 " "35 000 "  
## 3 Americas  Brazil    2022 Both sexes "51 000 " "47 000 "  
## 4 Americas  Brazil    2021 Female  "14 000 " "12 000 "  
## 5 Americas  Brazil    2021 Male    "37 000 " "34 000 "  
## 6 Americas  Brazil    2021 Both sexes "50 000 " "47 000 "  
## 7 Americas  Brazil    2020 Female  "14 000 " "13 000 "  
## 8 Americas  Brazil    2020 Male    "35 000 " "32 000 "  
## 9 Americas  Brazil    2020 Both sexes "49 000 " "46 000 "  
## 10 Americas Brazil    2019 Female  "14 000 " "13 000 "  
## # i 79 more rows  
## # i 1 more variable: cases_upper <chr>
```

In the code above, we split the `new_hiv_cases` column into three new columns: `cases`, `cases_lower`, and `cases_upper`. We use the `[` and `-` as separators. The double backslash `\\"` is used to escape the square bracket, which has a special meaning in regular expressions. And the `|` is used to indicate that either the `[` or the `-` can be used as a separator.



Large Language Models like ChatGPT are excellent at regular expression understanding. If you're ever stuck with code like `sep = "\\[| -"` and want to understand what it does, you can ask ChatGPT to explain it to you. And if you need to generate such expressions yourself, you can ask ChatGPT to generate them for you.

Next, we need to convert these string values into numeric values, removing any non-numeric characters.

```

hiv_data_brazil_clean <-
  hiv_data_brazil %>%
  separate(new_hiv_cases,
    into = c("cases", "cases_lower", "cases_upper"),
    sep = "\\\\[| -") %>%
  mutate(across(c("cases", "cases_lower", "cases_upper"),
    ~ str_replace_all(.x, "[^0-9]", "") %>%
      as.numeric())))

```

`hiv_data_brazil_clean`

```

## # A tibble: 89 × 7
##   continent country year sex     cases cases_lower
##   <chr>     <chr> <dbl> <chr>     <dbl>     <dbl>
## 1 Americas  Brazil  2022 Female    13000    12000
## 2 Americas  Brazil  2022 Male     37000    35000
## 3 Americas  Brazil  2022 Both sexes 51000    47000
## 4 Americas  Brazil  2021 Female    14000    12000
## 5 Americas  Brazil  2021 Male     37000    34000
## 6 Americas  Brazil  2021 Both sexes 50000    47000
## 7 Americas  Brazil  2020 Female    14000    13000
## 8 Americas  Brazil  2020 Male     35000    32000
## 9 Americas  Brazil  2020 Both sexes 49000    46000
## 10 Americas Brazil  2019 Female   14000    13000
## # ... 79 more rows
## # ... 1 more variable: cases_upper <dbl>

```

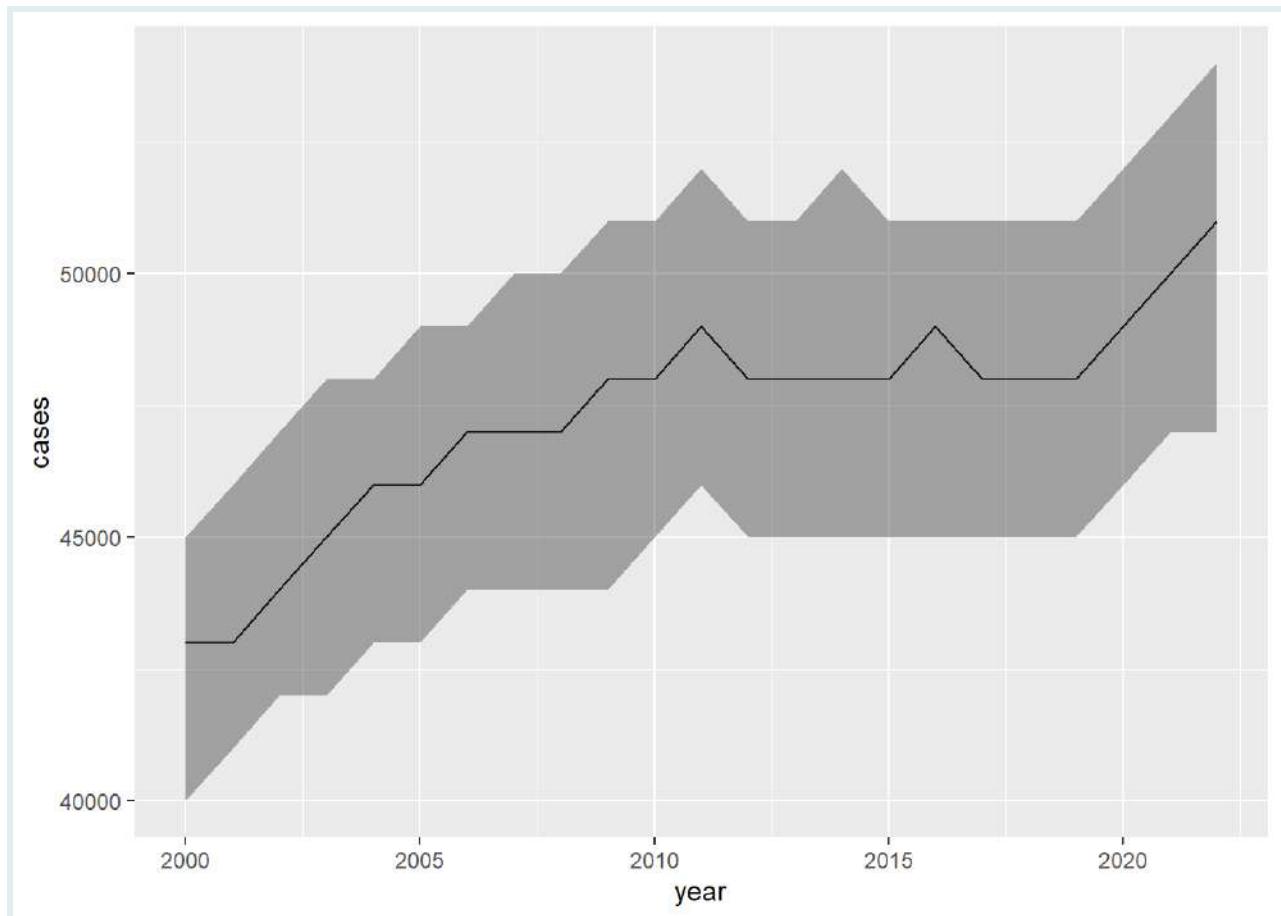
The code above looks complex, but essentially, it cleans the data by keeping only numeric characters and then converts these numbers to actual numeric values. See our lesson on the `across()` function for further details.

We're finally ready to plot the data. We'll use ggplot's `geom_ribbon()` to display the confidence intervals:

```

hiv_data_brazil_clean %>%
  filter(sex == "Both sexes") %>%
  ggplot(aes(x = year, y = cases)) +
  geom_line() +
  geom_ribbon(aes(ymin = cases_lower, ymax = cases_upper), alpha = 0.4)

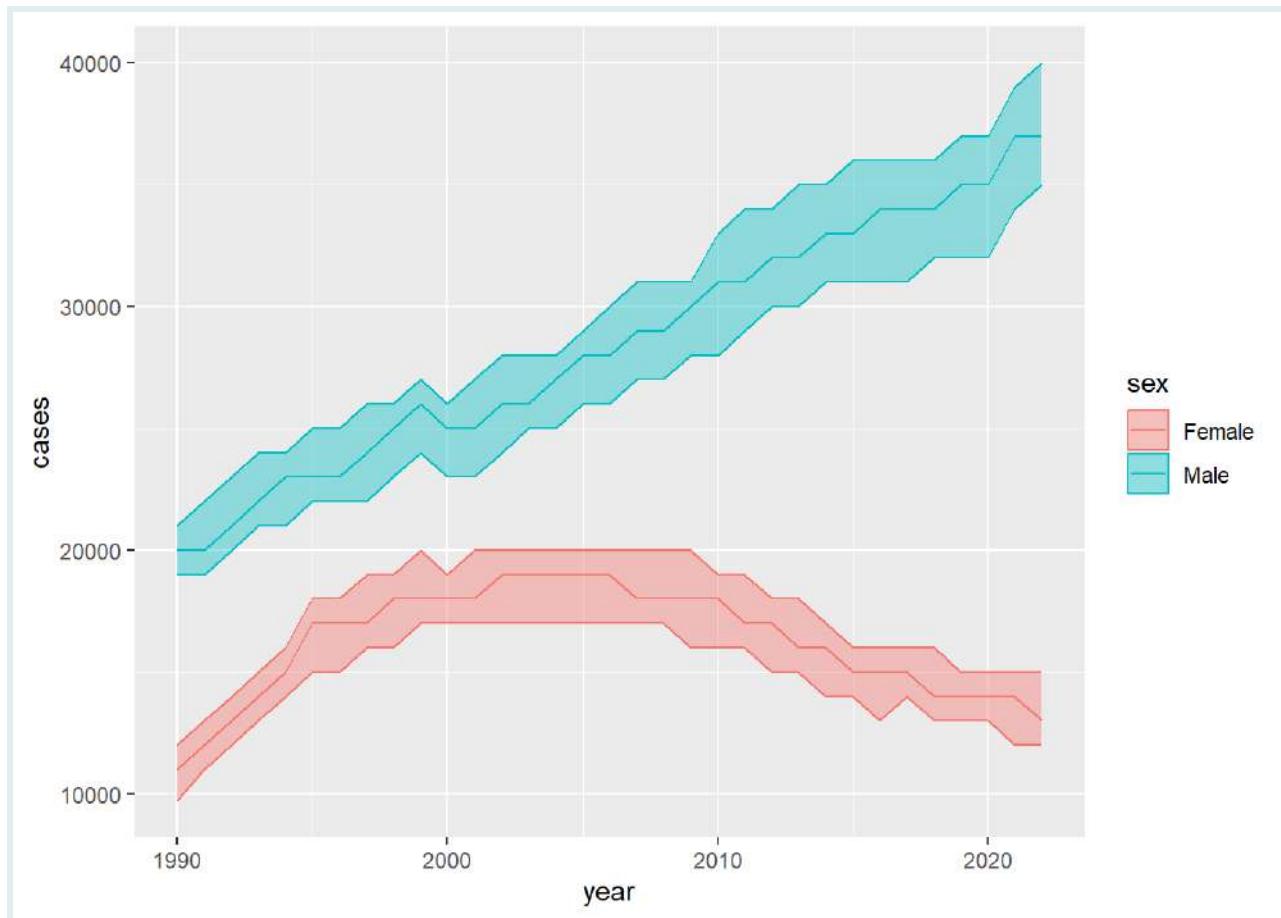
```



The `geom_ribbon()` function takes the `x` and `y` aesthetics like `geom_line()`, but it also takes in `ymin` and `ymax` aesthetics, to determine the vertical extent of the ribbon. We also set the transparency of the ribbon using the `alpha` argument.

We can create a separate ribbon for men and women to compare their infection trends.

```
hiv_data_brazil_clean %>%
  filter(sex != "Both sexes") %>%
  ggplot(aes(x = year, y = cases, color = sex, fill = sex)) +
  geom_line() +
  geom_ribbon(aes(ymin = cases_lower, ymax = cases_upper), alpha = 0.4)
```



Notably HIV infection rates among women have been falling in recent years, but those among men have been rising.



Q: Plotting confidence intervals

Consider the following dataset that shows the number of annual malaria cases in Kenya and Nigeria. The data is sourced from the WHO Global Health Observatory data repository and can be accessed [here](#).

```
nig_ken_mal <- read_csv("data/nigeria_kenya_malaria.csv")
nig_ken_mal
```

```
## # A tibble: 44 × 3
##   country  year malaria_cases
##   <chr>    <dbl> <chr>
## 1 Kenya     2021 3 419 698 (2 478 000 to 4 641 000)
## 2 Nigeria   2021 65 399 501 (47 520 000 to 89 000 000)
## 3 Kenya     2020 3 302 189 (2 385 000 to 4 466 000)
```



```
## 4 Nigeria 2020 65 133 759 (46 800 000 to 88 650 000)
## 5 Kenya 2019 3 037 541 (2 168 000 to 4 130 000)
## 6 Nigeria 2019 61 379 283 (47 440 000 to 78 810 000)
## 7 Kenya 2018 3 068 062 (2 148 000 to 4 260 000)
## 8 Nigeria 2018 59 652 248 (46 930 000 to 75 230 000)
## 9 Kenya 2017 3 155 636 (2 217 000 to 4 375 000)
## 10 Nigeria 2017 57 869 533 (45 870 000 to 72 050 000)
## # i 34 more rows
```

Write code to extract the confidence intervals from the “malaria_cases” column and create a plot with confidence intervals using `geom_ribbon()`. Use a different color for each country.

Smoothing Noisy Trends

When analyzing time series data, it is common for daily or granular measurements to show a lot of noise and variability, and this can hide the important trends we are actually interested in. Smoothing techniques can help highlight these trends and patterns. We'll explore several techniques for this in the sections below.

First though, let's do some data preparation!

Creating an Incidence Table from a Linelist

Consider the following linelist of pediatric malaria admissions in four hospitals in Mozambique ([Data source](#)):

```
mal <-  
  rio::import(here("data/pediatric_malaria_data_joao_2021.xlsx")) %>%  
  as_tibble() %>%  
  mutate(date_positive_test = as.Date(date_positive_test)) %>%  
  # Keep data from 2019-2020  
  filter(date_positive_test >= as.Date("2019-01-01"),  
         date_positive_test <= as.Date("2020-12-31"))  
mal
```

```
## # A tibble: 20,939 × 4  
##   date_positive_test neighbourhood sex     age  
##   <date>              <chr>       <chr> <chr>  
## 1 2020-01-22          25 de Junho  1 M    6-11 meses  
## 2 2020-01-22          Chicueu      F    5-14 anos  
## 3 2020-01-22          Mussessa     M    5-14 anos  
## 4 2020-01-22          Nhamizara    M    12-23 meses  
## 5 2020-01-22          Nhamizara    F    12-23 meses  
## 6 2020-01-22          Unidade      F    5-14 anos  
## 7 2020-01-22          Bapua        F    12-23 meses
```

```

## 8 2020-01-22      Bapua      F      5-14 anos
## 9 2020-01-22      7 de Abril   M     12-23 meses
## 10 2020-01-22     Nhanguzue   F      5-14 anos
## # i 20,929 more rows

```

Each row corresponds to a single malaria case, and the `date_positive_test` column indicates the date when the child tested positive for malaria.

To get a count of cases per day—that is, an incidence table—we can simply use `count()` to aggregate the cases by date of positive test:

```

mal %>%
  count(date_positive_test, name = "cases")

```

```

## # A tibble: 235 x 2
##   date_positive_test cases
##   <date>           <int>
## 1 2019-01-01         67
## 2 2019-01-02        120
## 3 2019-01-03        112
## 4 2019-01-04        203
## 5 2019-01-05         85
## 6 2019-01-07        115
## 7 2019-01-08        196
## 8 2019-01-10         89
## 9 2019-01-11         55
## 10 2019-01-12        69
## # i 225 more rows

```

There are many dates missing though—days when no children were admitted. To create a complete incidence table, we should use `complete()` to insert missing dates and then fill in the missing values with 0:

```

mal_notif_count <- mal %>%
  count(date_positive_test, name = "cases") %>%
  complete(date_positive_test = seq.Date(min(date_positive_test),
                                           max(date_positive_test),
                                           by = "day"),
            fill = list(cases = 0))

```

`mal_notif_count`

```

## # A tibble: 406 x 2
##   date_positive_test cases
##   <date>           <int>
## 1 2019-01-01         67
## 2 2019-01-02        120
## 3 2019-01-03        112
## 4 2019-01-04        203
## 5 2019-01-05         85
## 6 2019-01-06          0

```

```

## 7 2019-01-07      115
## 8 2019-01-08      196
## 9 2019-01-09       0
## 10 2019-01-10      89
## # i 396 more rows

```

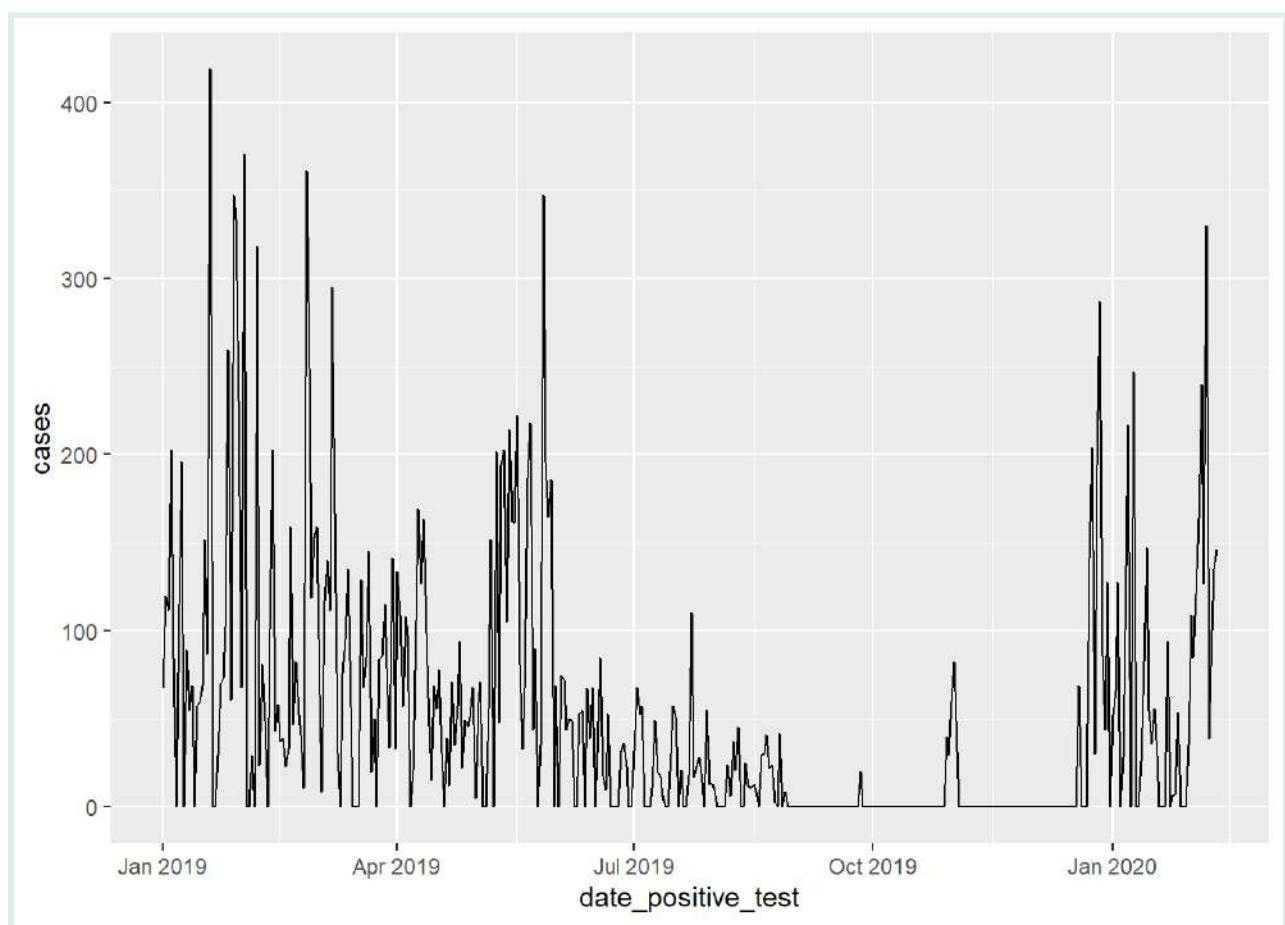
Now we have a complete incidence table with the number of cases on 406 consecutive days.

We can now plot the data to see the overall trend:

```

# Create a basic epicurve using ggplot2
ggplot(mal_notif_count, aes(x = date_positive_test, y = cases)) +
  geom_line()

```

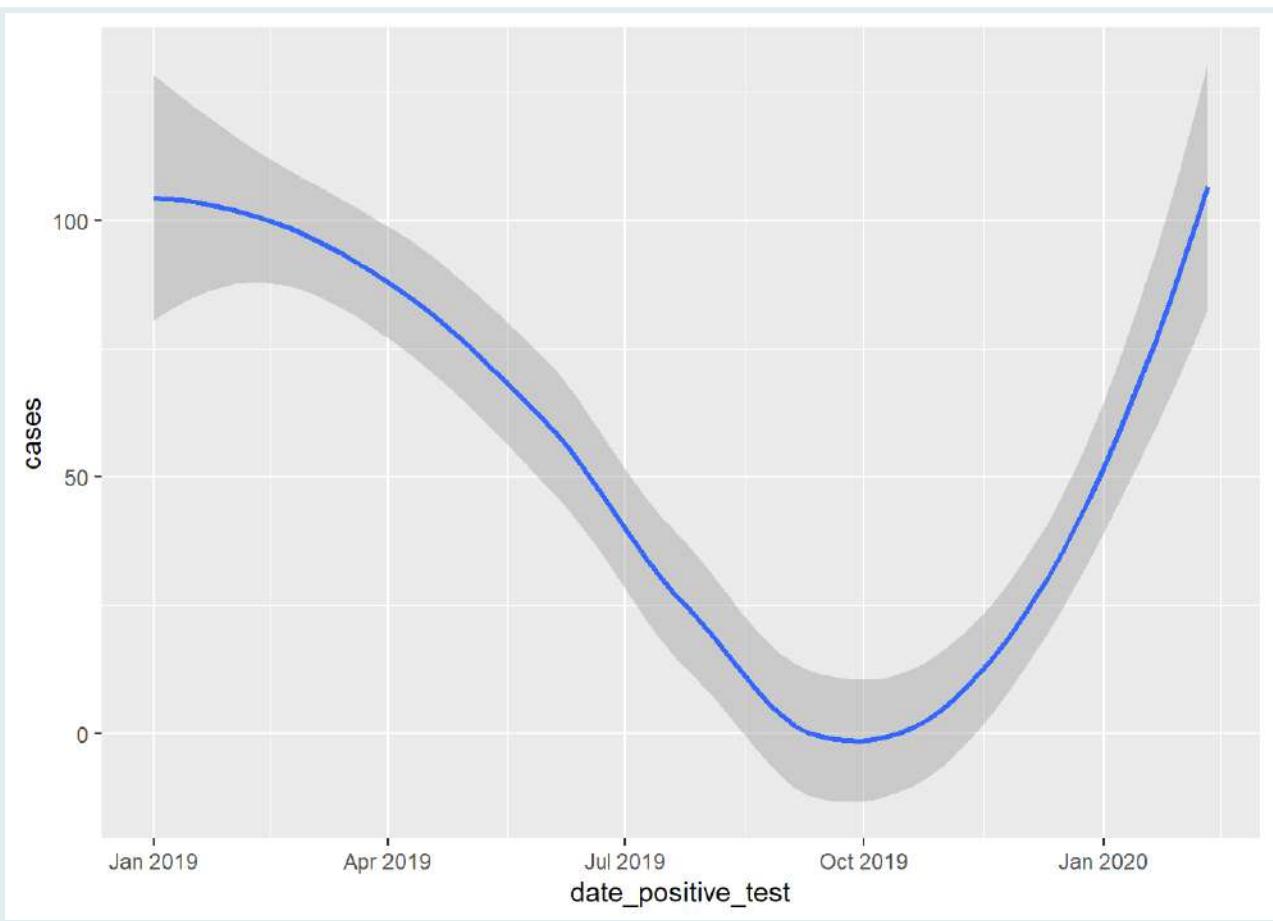


We have a valid epicurve, but as you may notice, the daily variability makes it hard to see the overall trend. Let's smooth things out.

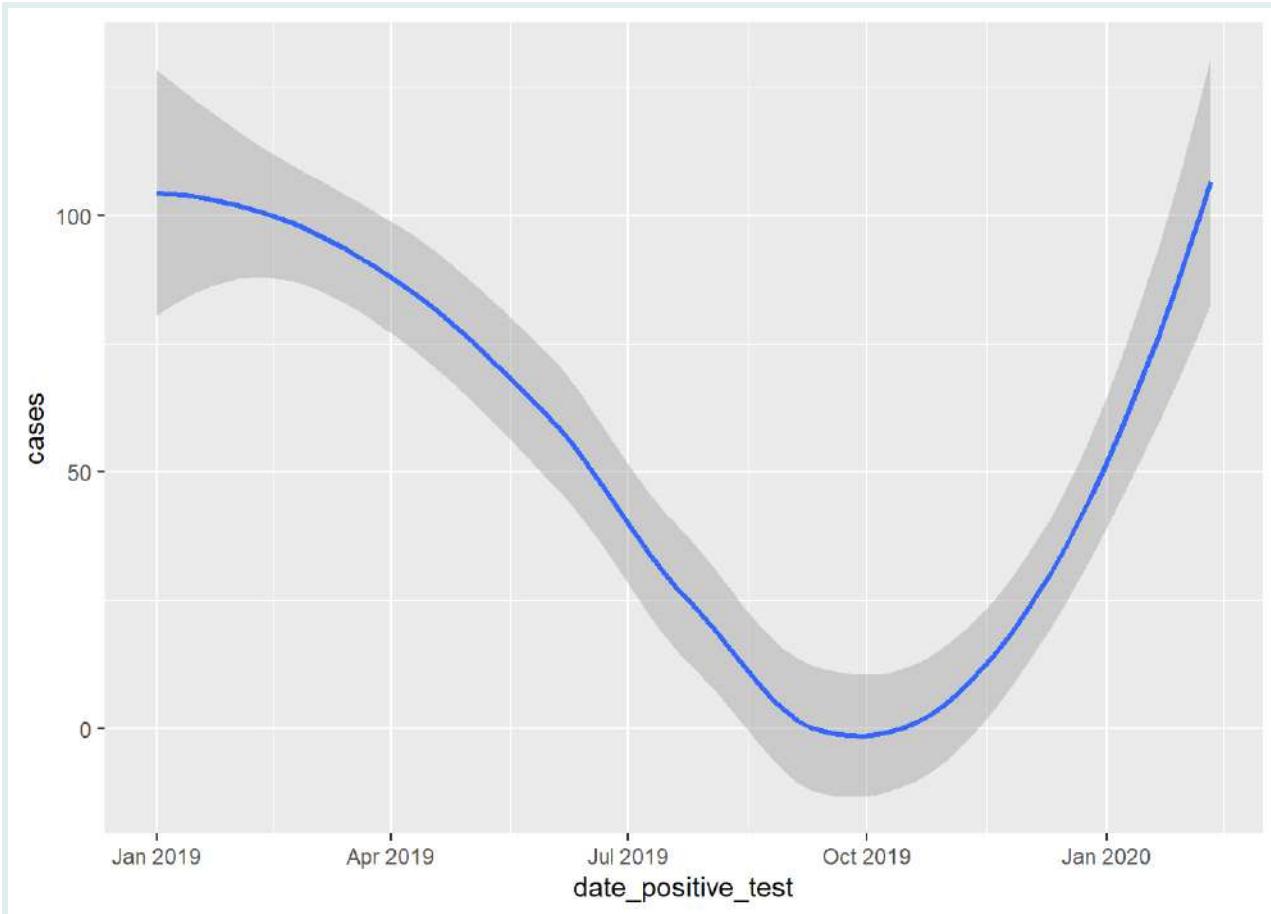
Smoothing with `geom_smooth()`

One option for smoothing is the `geom_smooth()` function, which can perform local regression with `loess` to smooth out the time series. Let's try it out:

```
ggplot(mal_notif_count, aes(x = date_positive_test, y = cases)) +  
  geom_smooth()
```



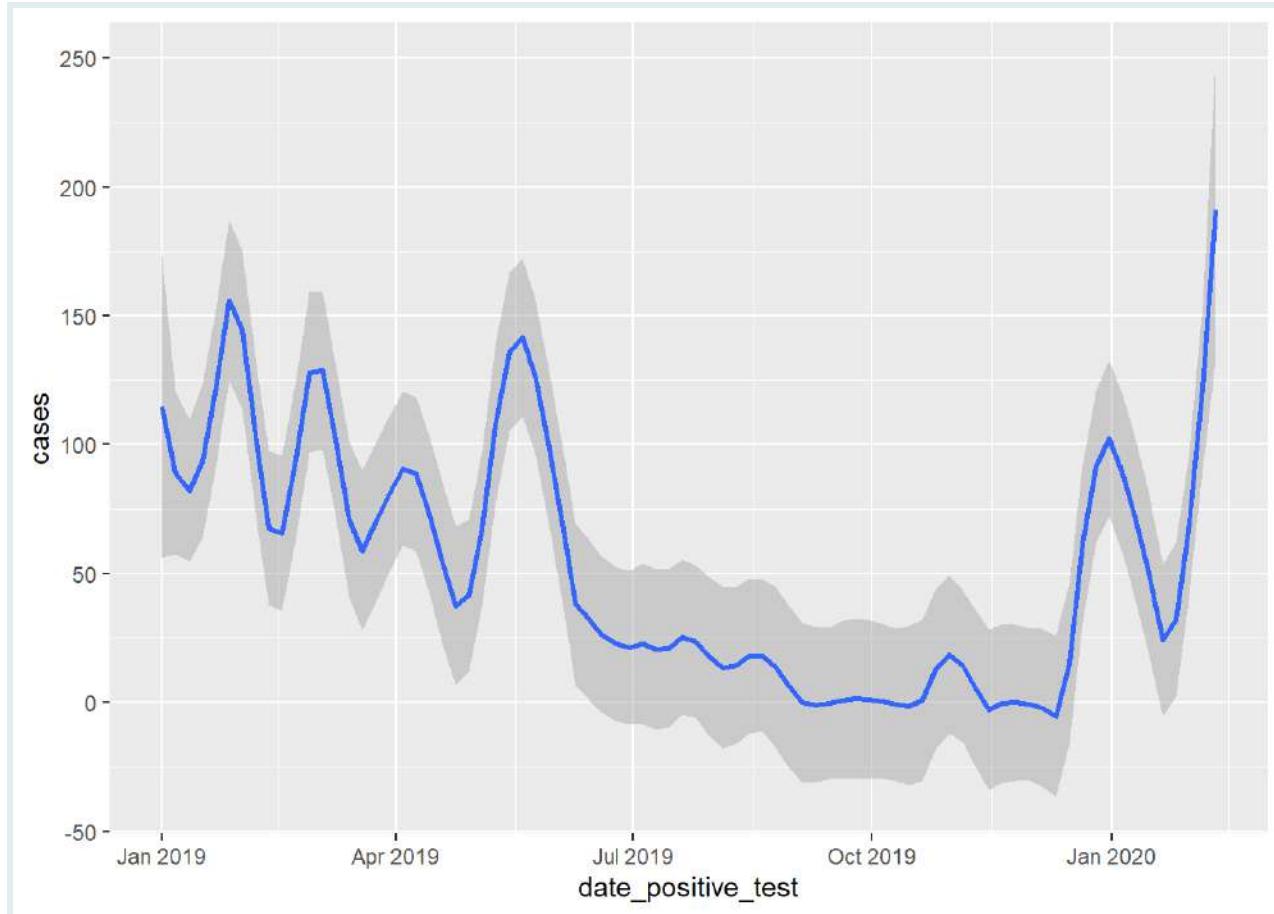
```
# Or we can specify the method explicitly  
ggplot(mal_notif_count, aes(x = date_positive_test, y = cases)) +  
  geom_smooth(method = "loess")
```



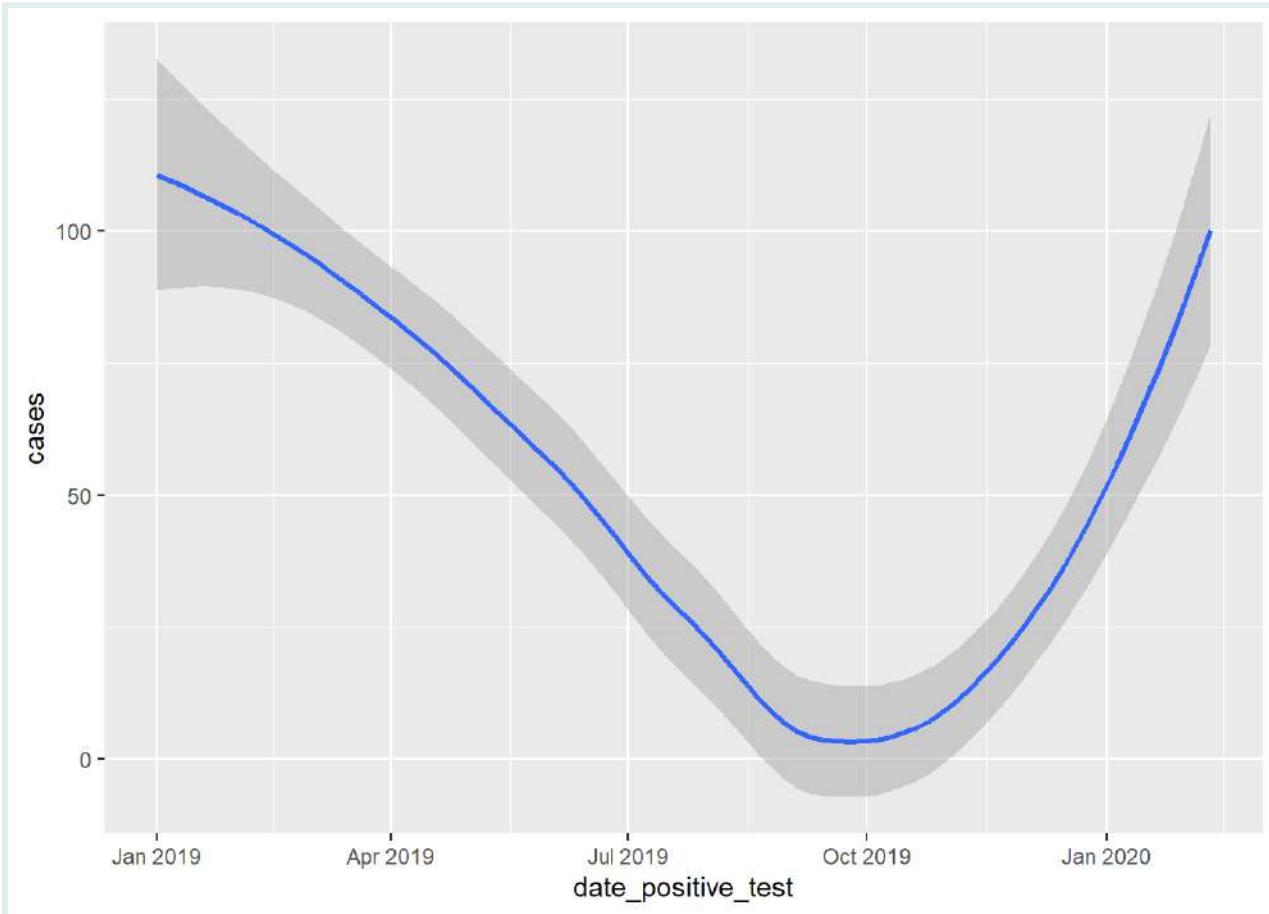
The `loess` methods, which stands for locally weighted scatterplot smoothing, fits a smooth curve to the data by calculating weighted averages for nearby points.

You can adjust the sensitivity of the smoothing by modifying the `span` argument. A span of 0.1 will result in a more sensitive smoothing, while a span of 0.9 will result in a less sensitive smoothing.

```
# Adjust the sensitivity of the smoothing
ggplot(mal_notif_count, aes(x = date_positive_test, y = cases)) +
  geom_smooth(method = "loess", span = 0.1)
```



```
ggplot(mal_notif_count, aes(x = date_positive_test, y = cases)) +  
  geom_smooth(method = "loess", span = 0.9)
```



Smoothing by Aggregating

Another way to smooth data is by aggregating it—grouping the data into larger time intervals and calculating summary statistics for each interval.

We already saw this at the start of the lesson, when we aggregated quarterly data to yearly data.

Let's apply it again, this time aggregating daily malaria incidence to monthly incidence. To do this, we employ the `floor_date()` function from the `lubridate` package to round the dates down to the nearest month:

```
mal_notif_count %>%
  mutate(month = floor_date(date_positive_test, unit = "month"))
```

```
## # A tibble: 406 x 3
##   date_positive_test cases month
##   <date>           <int> <date>
## 1 2019-01-01          67 2019-01-01
## 2 2019-01-02         120 2019-01-01
## 3 2019-01-03         112 2019-01-01
## 4 2019-01-04         203 2019-01-01
## 5 2019-01-05          85 2019-01-01
```

```

## 6 2019-01-06          0 2019-01-01
## 7 2019-01-07        115 2019-01-01
## 8 2019-01-08        196 2019-01-01
## 9 2019-01-09          0 2019-01-01
## 10 2019-01-10         89 2019-01-01
## # i 396 more rows

```

We can then use `group_by()` and `summarize()` to calculate the total number of cases per month:

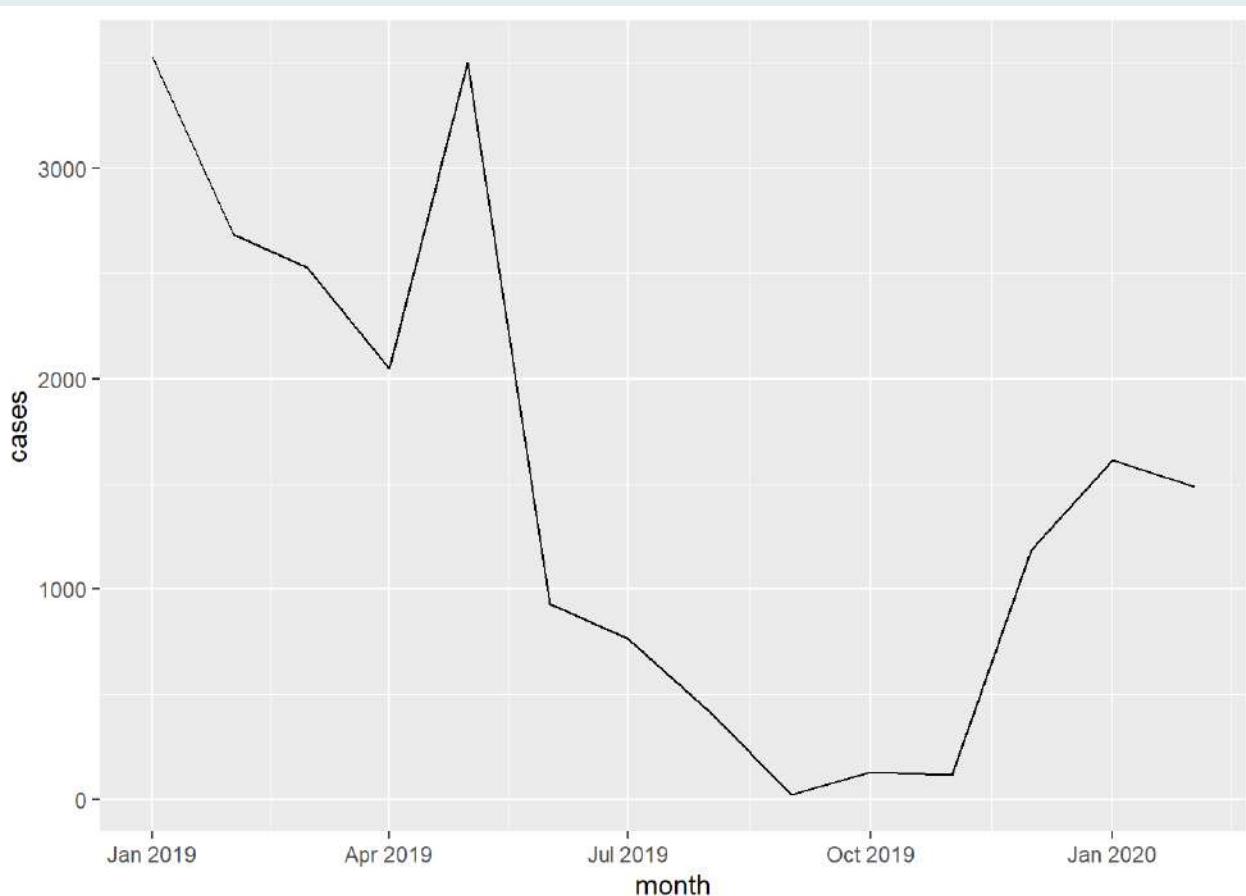
```

mal_monthly <-
  mal_notif_count %>%
  mutate(month = floor_date(date_positive_test, unit = "month")) %>%
  group_by(month) %>%
  summarize(cases = sum(cases))

```

This gives us a monthly incidence table, which we can plot to see the overall trend:

```
ggplot(mal_monthly, aes(x = month, y = cases)) +
  geom_line()
```



Voila! A much clearer picture.

PRACTICE

Q: Smoothing HIV Death Data in Colombia

Consider this dataset of individuals who died from HIV in Colombia between 2010 and 2016, sourced from [this URL](#).

```
colom_hiv_deaths <-  
  read_csv(here("data/colombia_hiv_deaths_2010_to_2016.csv")) %>%  
  mutate(date_death = ymd(paste(death_year, death_month,  
    death_day, sep = "-")))  
colom_hiv_deaths
```

```
## # A tibble: 445 × 26  
##   municipality_type death_location birth_date birth_year  
##   <chr>              <chr>          <date>        <dbl>  
## 1 Municipal head   Hospital/clinic 1956-05-26  1956  
## 2 Municipal head   Hospital/clinic 1983-10-10  1983  
## 3 Municipal head   Hospital/clinic 1967-11-22  1967  
## 4 Municipal head   Home/address   1964-03-14  1964  
## 5 Municipal head   Hospital/clinic 1960-06-27  1960  
## 6 Municipal head   Hospital/clinic 1982-03-23  1982  
## 7 Municipal head   Hospital/clinic 1964-12-09  1964  
## 8 Municipal head   Hospital/clinic 1975-01-15  1975  
## 9 Municipal head   Hospital/clinic 1988-02-15  1988  
## 10 Municipal head  Hospital/clinic NA            NA  
## # i 435 more rows  
## # i 22 more variables: birth_month <chr>, ...
```



Using the steps taught above:

1. Create a table that counts HIV-related deaths per month.
2. Plot an epicurve of the deaths per month
3. Apply `geom_smooth` to the epicurve for a smoother visualization.
Ensure you choose an appropriate span for smoothing.

Smoothing with Rolling Averages

Another technique to smooth noisy time series data is to calculate **rolling averages**. This takes the average of a fixed number of points, centered around each data point.

The `rollmean()` function from the `{zoo}` package will be your primary work-horse for calculating rolling means.

Let's apply a 14 day rolling average to smooth our daily malaria case data:

```
mal_notif_count <- mal_notif_count %>%
  mutate(roll_cases = rollmean(cases, k = 14, fill = NA))
mal_notif_count
```

```
## # A tibble: 406 x 3
##   date_positive_test cases roll_cases
##   <date>           <int>     <dbl>
## 1 2019-01-01          67      NA
## 2 2019-01-02         120      NA
## 3 2019-01-03         112      NA
## 4 2019-01-04         203      NA
## 5 2019-01-05          85      NA
## 6 2019-01-06          0       NA
## 7 2019-01-07         115     83.4
## 8 2019-01-08         196     82.9
## 9 2019-01-09          0      79.3
## 10 2019-01-10         89      82.1
## # ... i 396 more rows
```

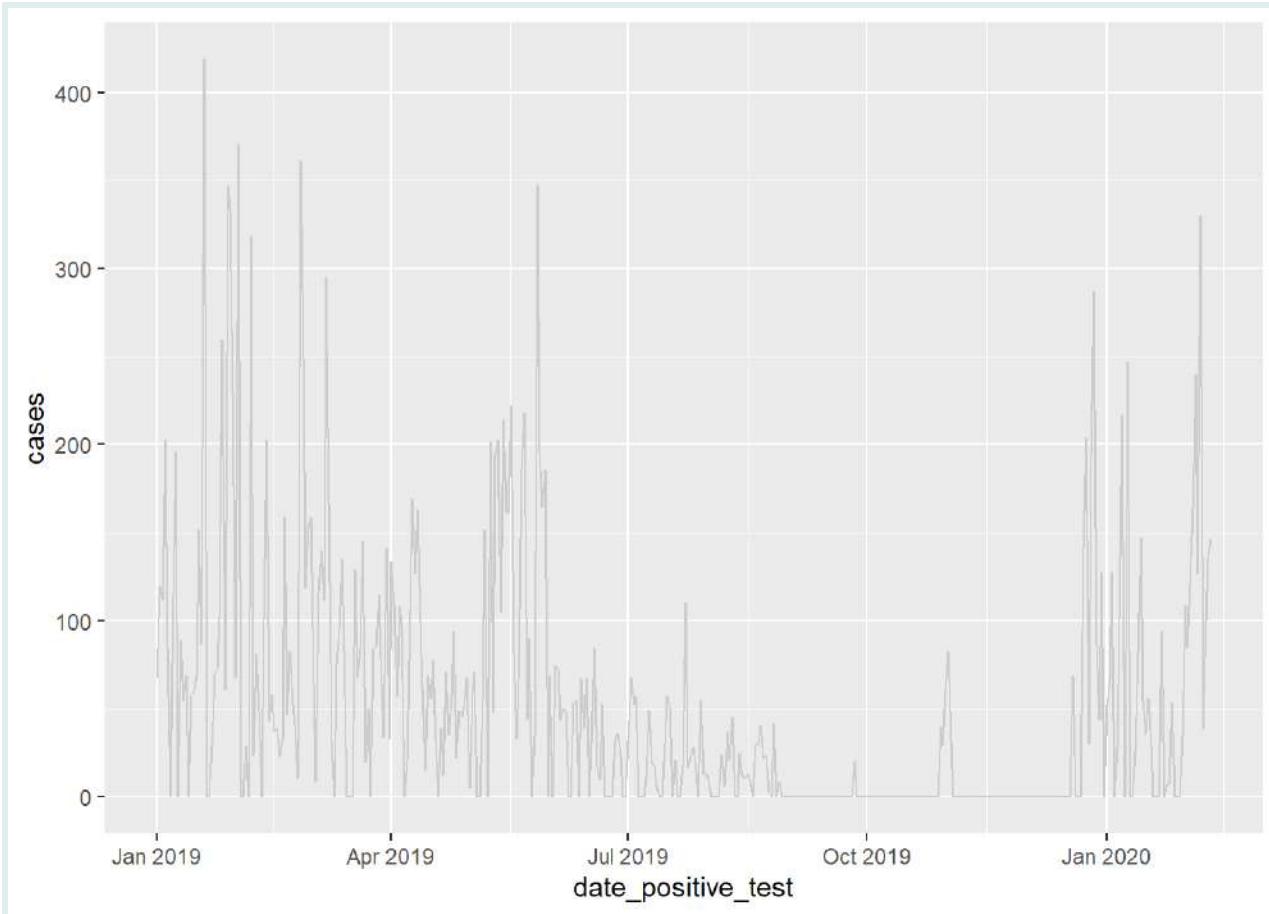
The key arguments are:

- **x**: The time series to smooth
- **k**: The number of points before and after to average
- **fill**: How to handle missing data within each window

This calculates the 14-day moving average, leaving missing data as NA. Notice that the first 6 days are NA, since there are not enough points to average over (with a **k** of 14, we need 7 days before and 7 days after each point to calculate the rolling average).

Let's plot it:

```
mal_notif_count %>%
  ggplot(aes(x = date_positive_test, y = cases)) +
  geom_line(color = "gray80")
```



Commonly, you will be asked to plot a rolling average of the *past* 1 or 2 weeks. For this, you must set the `align` argument to "right":

```
mal_notif_count_right <-
  mal_notif_count %>%
  mutate(roll_cases = rollmean(cases, k = 14, fill = NA, align = "right"))

head(mal_notif_count_right, 15)
```

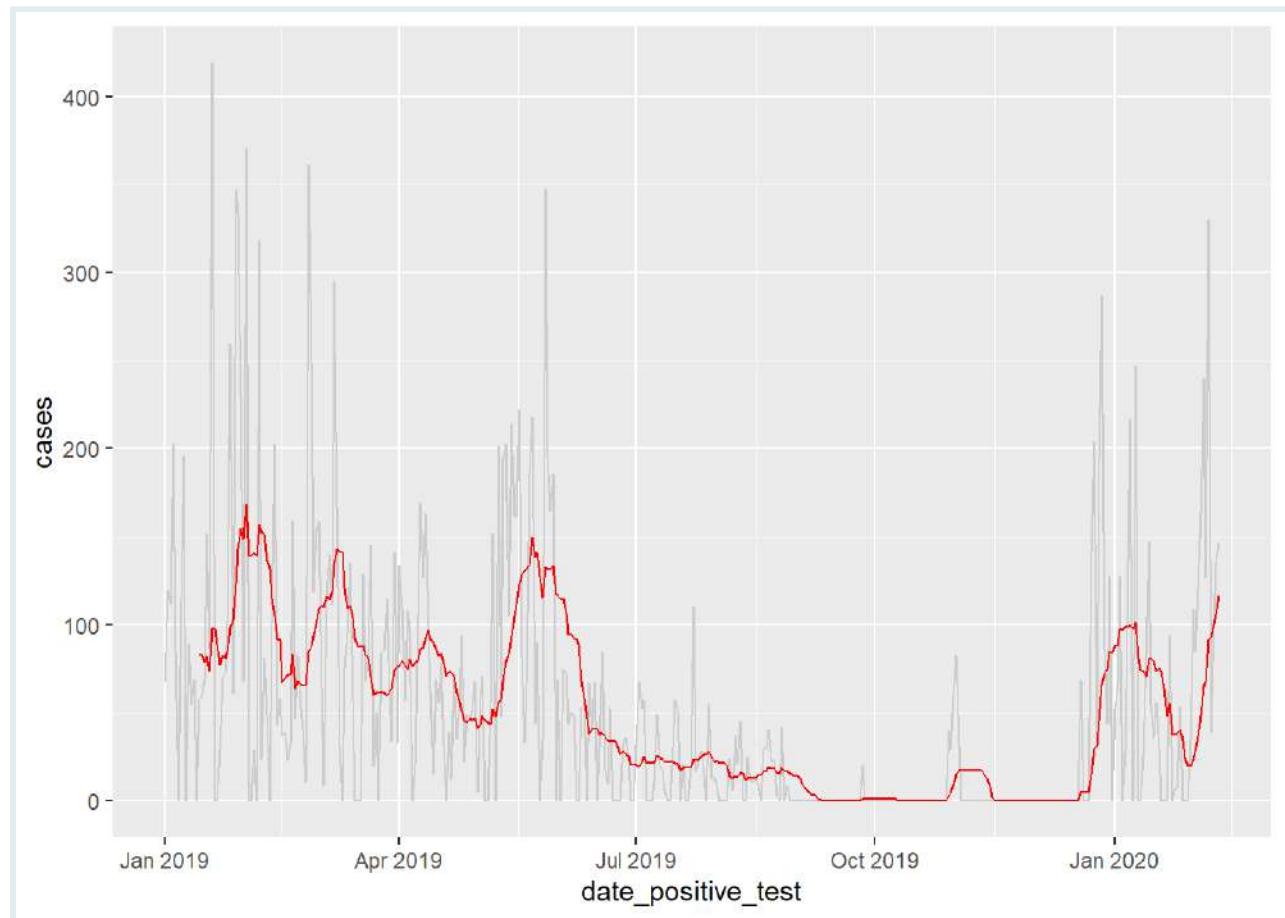
```
## # A tibble: 15 × 3
##   date_positive_test cases roll_cases
##   <date>           <int>     <dbl>
## 1 2019-01-01          67      NA
## 2 2019-01-02         120      NA
## 3 2019-01-03         112      NA
## 4 2019-01-04         203      NA
## 5 2019-01-05          85      NA
## 6 2019-01-06           0      NA
## 7 2019-01-07         115      NA
## 8 2019-01-08         196      NA
## 9 2019-01-09           0      NA
## 10 2019-01-10          89      NA
## 11 2019-01-11          55      NA
## 12 2019-01-12          69      NA
## 13 2019-01-13           0      NA
```

```
## 14 2019-01-14      57      83.4
## 15 2019-01-15      60      82.9
```

Notice that now the first 13 days are NA, since there are not enough points to average over. This is because we are calculating the average of the *past* 14 days, and the first 13 days do not have 14 days before them.

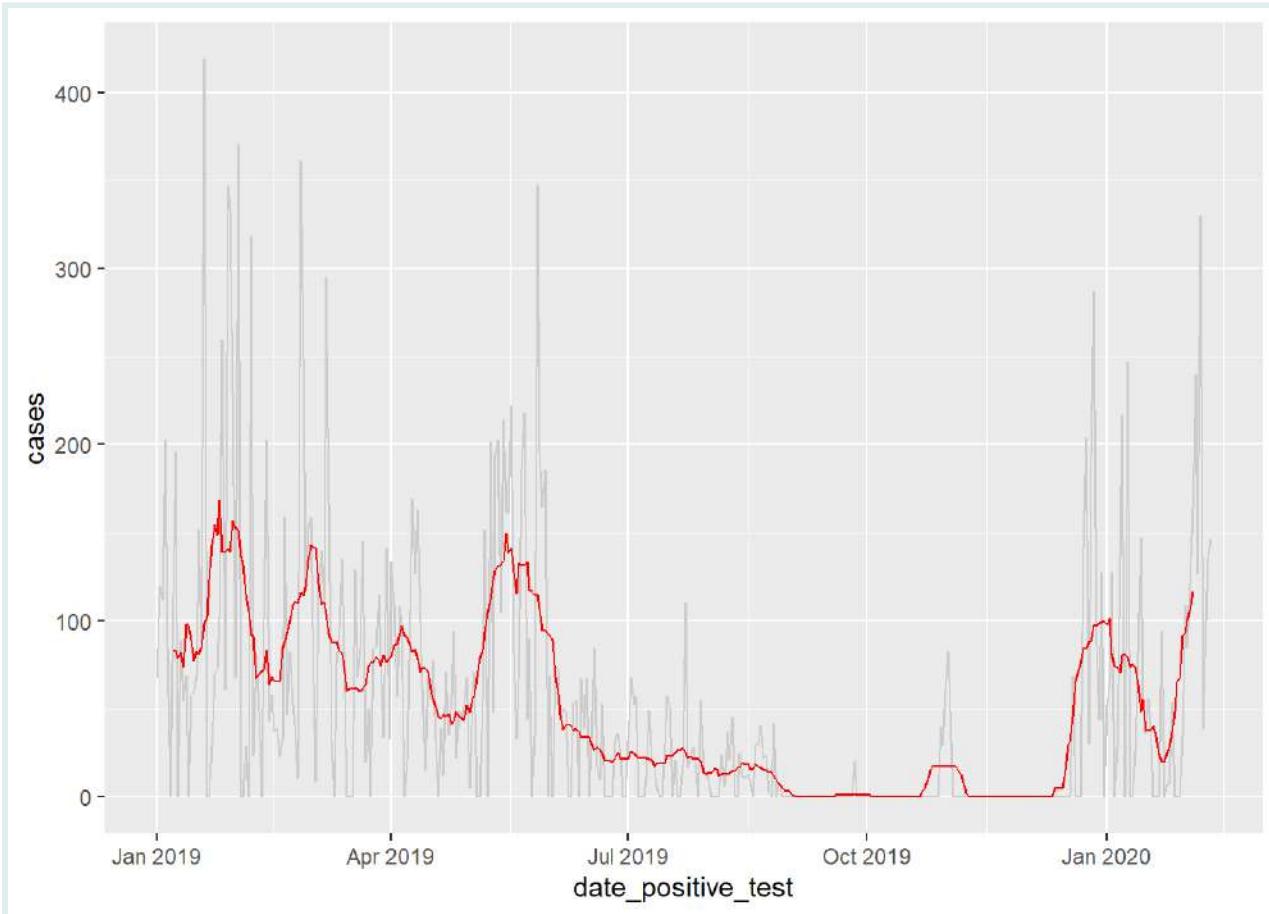
The output does not change much in this case:

```
ggplot(mal_notif_count_right, aes(x = date_positive_test, y = cases)) +
  geom_line(color = "gray80") +
  geom_line(aes(y = roll_cases), color = "red")
```



Finally, let's plot both the original and smoothed data:

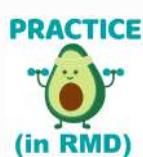
```
mal_notif_count %>%
  ggplot(aes(x = date_positive_test, y = cases)) +
  geom_line(color = "gray80") +
  geom_line(aes(y = roll_cases), color = "red")
```



In summary, the `rollmean()` function lets us easily compute a rolling average over a fixed window to smooth and highlight patterns in noisy time series data.

Q: Smoothing with rolling averages

Consider again the dataset of HIV patient deaths in Colombia:



`colom_hiv_deaths`

```
## # A tibble: 445 x 26
##   municipality_type death_location birth_date birth_year
##   <chr>              <chr>          <date>        <dbl>
## 1 Municipal head    Hospital/clinic 1956-05-26  1956
## 2 Municipal head    Hospital/clinic 1983-10-10  1983
## 3 Municipal head    Hospital/clinic 1967-11-22  1967
## 4 Municipal head    Home/address    1964-03-14  1964
## 5 Municipal head    Hospital/clinic 1960-06-27  1960
## 6 Municipal head    Hospital/clinic 1982-03-23  1982
## 7 Municipal head    Hospital/clinic 1964-12-09  1964
## 8 Municipal head    Hospital/clinic 1975-01-15  1975
```

```

## 9 Municipal head      Hospital/clinic 1988-02-15      1988
## 10 Municipal head     Hospital/clinic NA             NA
## # i 435 more rows
## # i 22 more variables: birth_month <chr>, ...

```

The following code calculates the number of deaths per day:

```

colom_hiv_deaths_per_day <-
  colom_hiv_deaths %>%
  group_by(date_death) %>%
  summarize(deaths = n()) %>%
  complete(date_death = seq.Date(min(date_death),
                                 max(date_death),
                                 by = "day"),
           fill = list(deaths = 0))

colom_hiv_deaths_per_day

```



```

## # A tibble: 2,543 × 2
##   date_death   deaths
##   <date>       <int>
## 1 2010-01-05     1
## 2 2010-01-06     0
## 3 2010-01-07     0
## 4 2010-01-08     0
## 5 2010-01-09     1
## 6 2010-01-10     0
## 7 2010-01-11     1
## 8 2010-01-12     0
## 9 2010-01-13     0
## 10 2010-01-14    0
## # i 2,533 more rows

```

Your task is to create a new column that calculates the rolling average of deaths per day over a 14-day period. Then, plot this rolling average alongside the raw data.

Secondary Axes

Understanding the Concept of a Secondary Y-Axis

A secondary y-axis is helpful when visualizing two different measures with distinct scales on the same plot. This approach is useful when the variables have different units or magnitudes, making direct comparison on a single scale challenging.

While some data visualization experts caution against using secondary axes, public health decision-makers often appreciate these plots.

Creating a Plot with a Secondary Y-Axis

Let's demonstrate how to create a plot with a secondary y-axis using our dataset of malaria notifications:

Step 1: Create Cumulative Case Counts

First, we'll aggregate our malaria data to calculate cumulative case counts.

```
mal_notif_count_cumul <-
  mal_notif_count %>%
  mutate(cumul_cases = cumsum(cases))

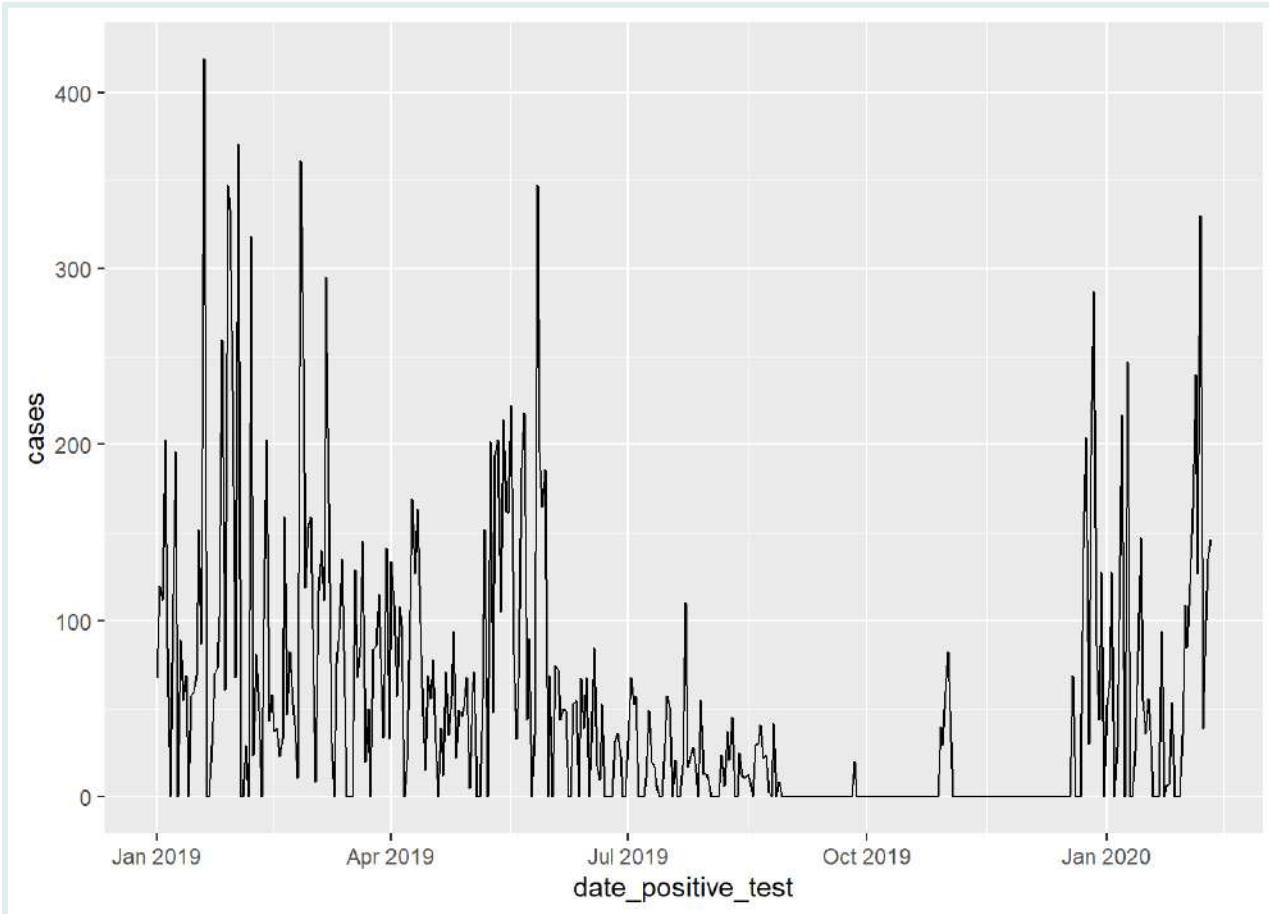
mal_notif_count_cumul
```

```
## # A tibble: 406 x 4
##   date_positive_test cases roll_cases cumul_cases
##   <date>           <int>     <dbl>      <int>
## 1 2019-01-01          67       NA        67
## 2 2019-01-02         120       NA       187
## 3 2019-01-03         112       NA       299
## 4 2019-01-04         203       NA       502
## 5 2019-01-05          85       NA       587
## 6 2019-01-06          0       NA       587
## 7 2019-01-07         115     83.4       702
## 8 2019-01-08         196     82.9       898
## 9 2019-01-09          0      79.3       898
## 10 2019-01-10         89     82.1       987
## # ... i 396 more rows
```

Step 2: Identifying the Need for a Secondary Y-Axis

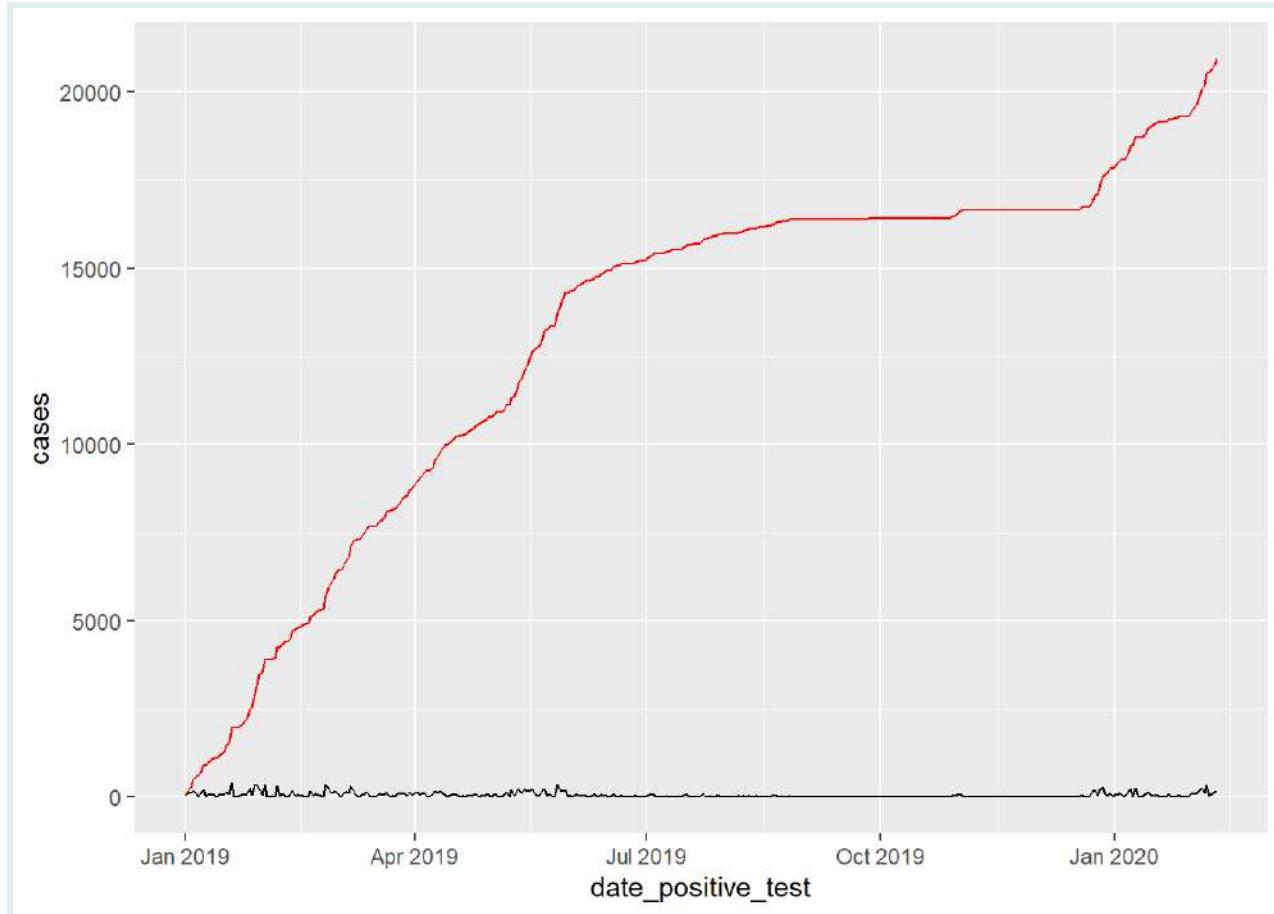
Now, we can start plotting. First, we plot just the daily cases:

```
# Plotting total malaria cases
ggplot(mal_notif_count_cumul, aes(x = date_positive_test)) +
  geom_line(aes(y = cases))
```



If we try to add cumulative cases on the same y-axis, the daily cases will be dwarfed and their magnitude will hard to read due to the much larger scale of cumulative data:

```
# Adding cumulative malaria cases to the plot
ggplot(mal_notif_count_cumul, aes(x = date_positive_test)) +
  geom_line(aes(y = cases)) +
  geom_line(aes(y = cumul_cases), color = "red")
```



To effectively display both sets of data, we must introduce a secondary y-axis.

Step 3: Calculating and Applying the Scale Factor

Before adding a secondary axis, we need to determine a *scale factor* by comparing the ranges of cases and cumulative cases.

The scale factor is typically the ratio of the maximum values of the two datasets. Let's see what the maximum values are for each variable:

```
max(mal_notif_count_cumul$cases)
```

```
## [1] 419
```

```
max(mal_notif_count_cumul$cumul_cases)
```

```
## [1] 20939
```

With a maximum of around 20000 for the cumulative cases, and about 400 for the daily cases, we can see that the cumulative cases are about 50 times larger than the

daily cases, so our scale factor will be about 50.

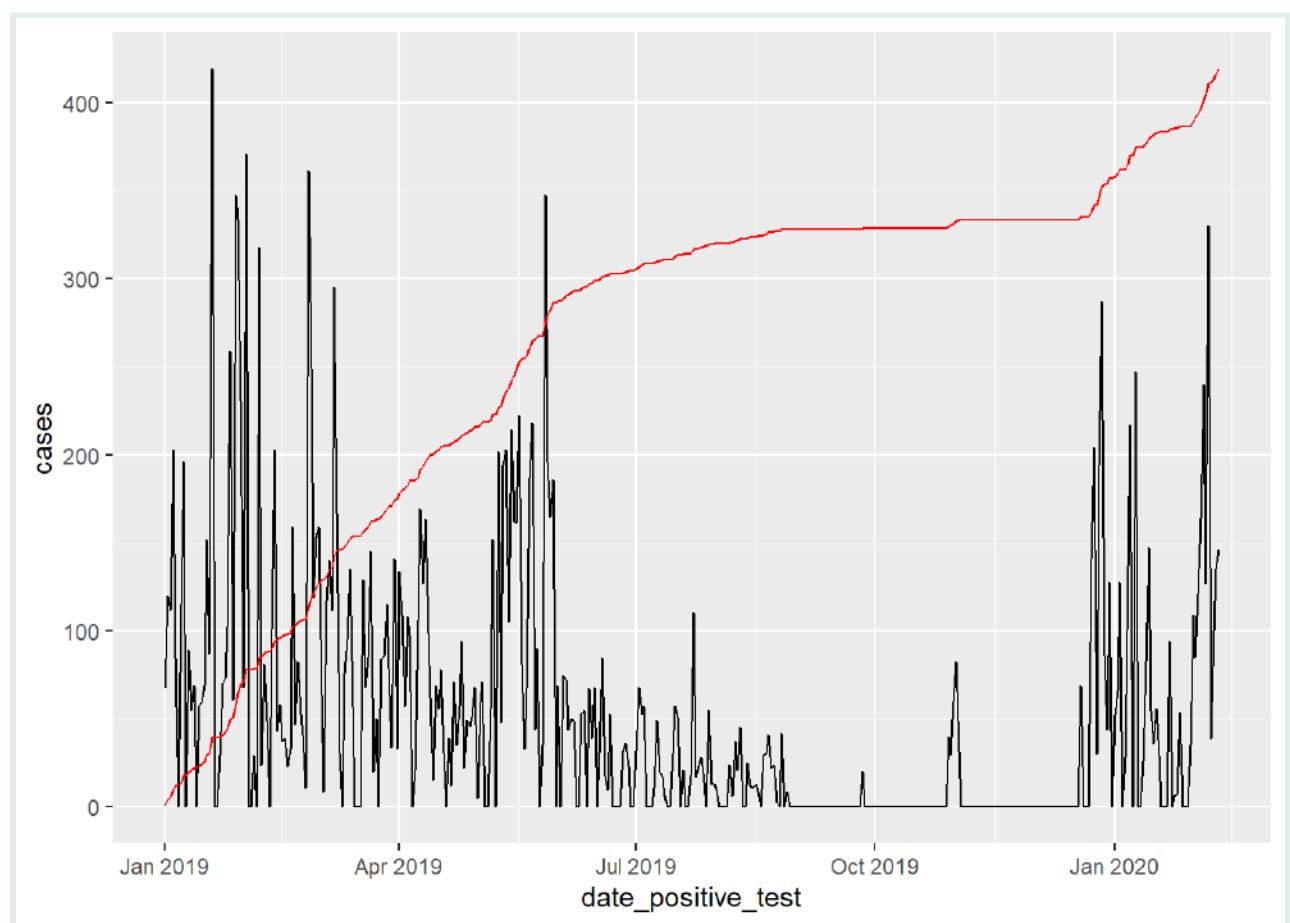
More precisely, the scale factor will be:

```
max(mal_notif_count_cumul$cumul_cases) / max(mal_notif_count_cumul$cases)
```

```
## [1] 49.97375
```

We'll need to divide the cumulative cases by this ratio to force the two variables to be on a similar scale:

```
ggplot(mal_notif_count_cumul, aes(x = date_positive_test)) +  
  geom_line(aes(y = cases)) +  
  geom_line(aes(y = cumul_cases / 49.97), color = "red") # divide by scale  
  factor
```



Great! Now we can see both sets of data clearly on the same plot, and their maximum points are aligned. However, the y-axis is no longer relevant for the red cumulative cases line. We need to add a secondary y-axis for this.

SIDE NOTE



SIDE NOTE

Normally, you would assign the scale factor to a variable, and then use that variable in the `geom_line()` function. In this case, we're typing it out directly in the function for easier understanding.

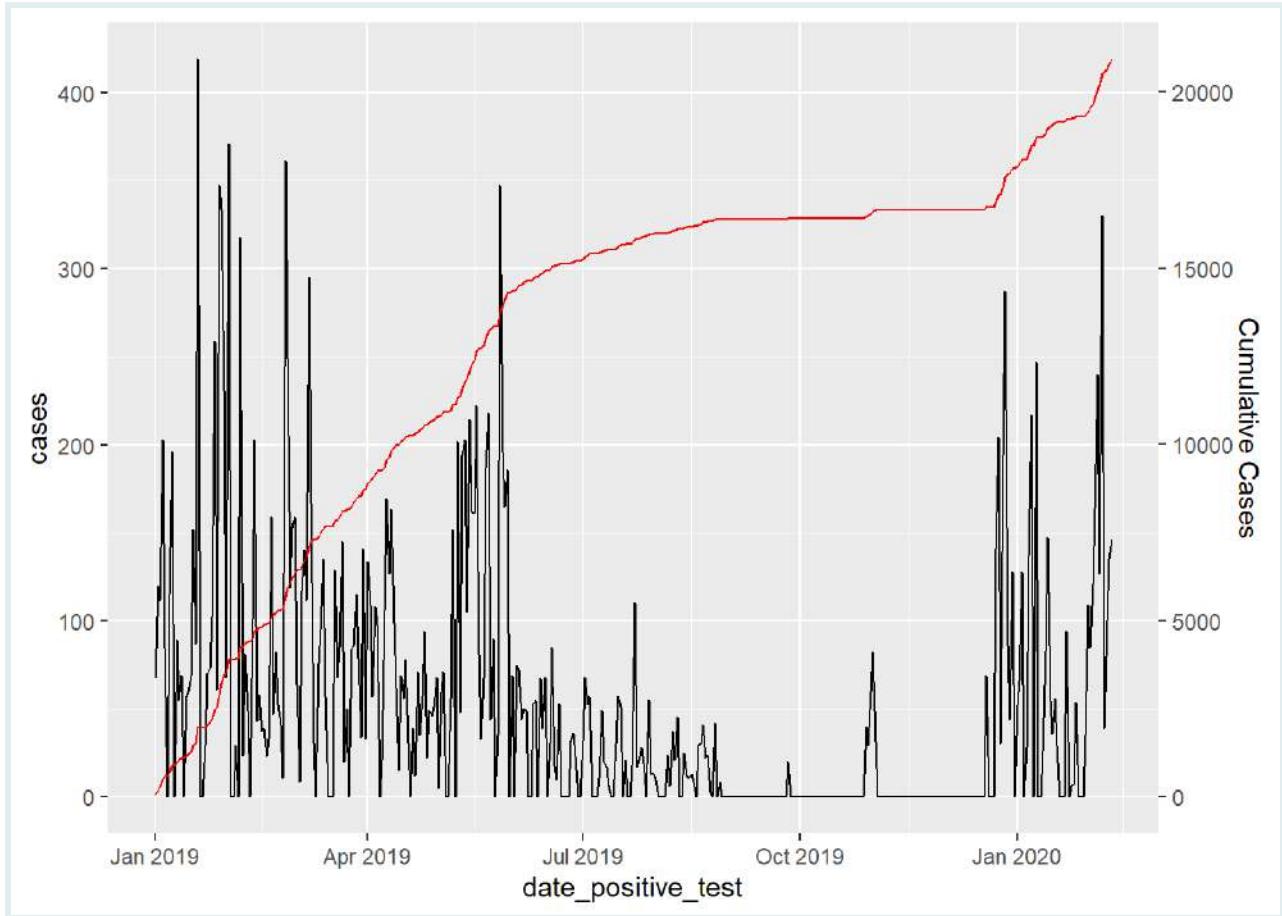
Step 4: Adding the Secondary Y-Axis

We'll use the `sec_axis()` function from `{ggplot2}`. The key arguments are `trans`, which indicates how much to multiply or divide the original y axis, and `name`, which specifies the name of the secondary axis.

In our case, we want the secondary axis to be about 49.97 times larger than the original axis, so we'll use `trans = ~ .x * 49.97`. (The `~` symbol is a special operator that tells R to treat the expression that follows it as a function, whose input is indicated by the `.x` symbol.)

Let's implement this:

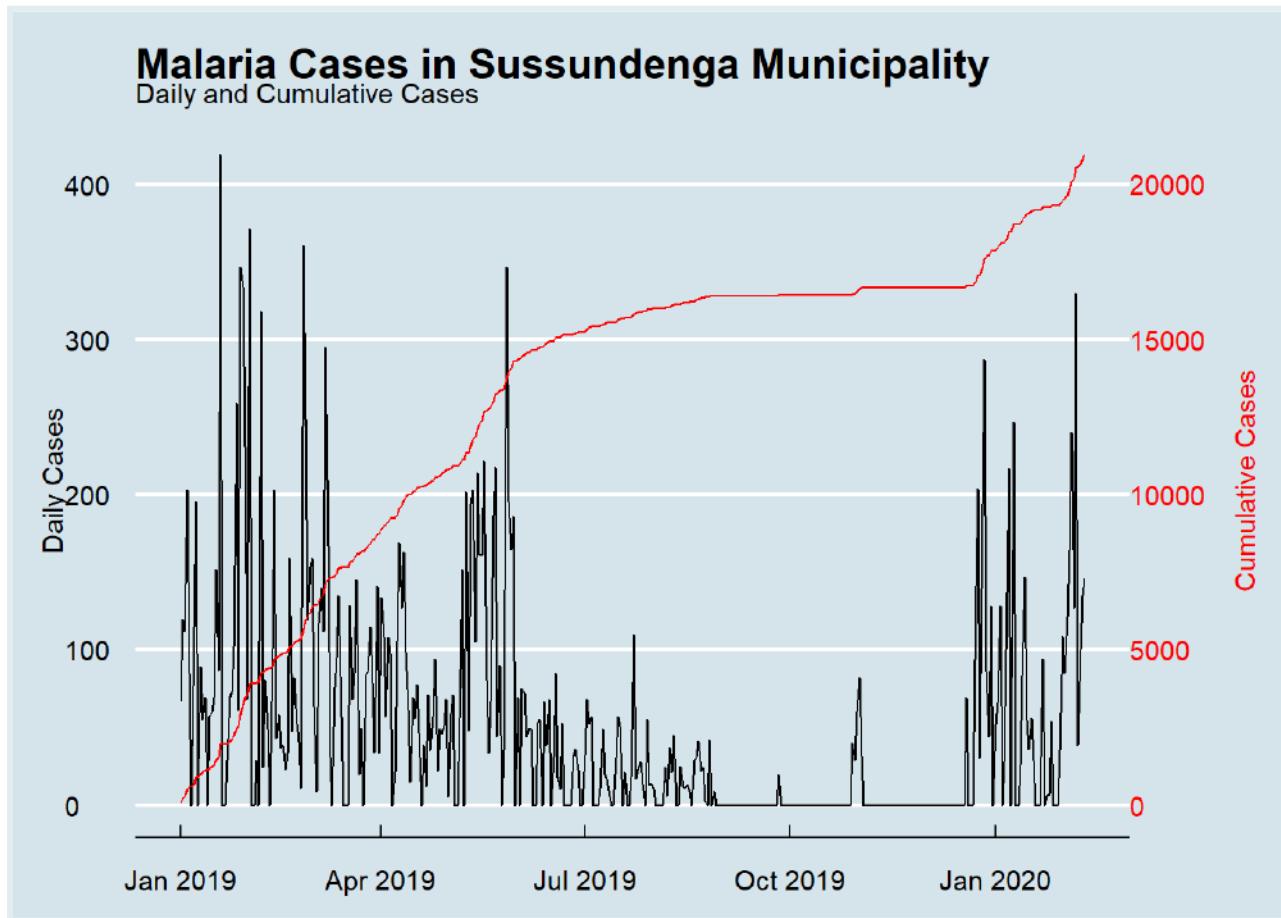
```
# Add a secondary y-axis
ggplot(mal_notif_count_cumul, aes(x = date_positive_test)) +
  geom_line(aes(y = cases)) +
  geom_line(aes(y = cumul_cases / 49.97), color = "red") +
  scale_y_continuous(sec.axis = sec_axis(trans = ~ .x * 49.97,
                                         name = "Cumulative Cases"))
```



Step 5: Enhancing Plot Readability

To improve readability, we'll make the secondary axis labels red, matching the color of the cumulative cases line, and we'll add some additional formatting to the plot:

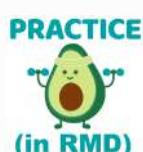
```
# Finalizing the plot with color-coordinated axes
ggplot(mal_notif_count_cumul, aes(x = date_positive_test)) +
  geom_line(aes(y = cases)) +
  geom_line(aes(y = cumul_cases / 49.97), color = "red") +
  scale_y_continuous(
    name = "Daily Cases",
    sec.axis = sec_axis(~ . * 49.97, name = "Cumulative Cases")
  ) +
  labs(title = "Malaria Cases in Sussundenga Municipality",
       subtitle = "Daily and Cumulative Cases",
       x = NULL) +
  theme_economist() +
  theme(axis.text.y.right = element_text(color = "red"),
        axis.title.y.right = element_text(color = "red"))
```



All done! We've successfully added a secondary y-axis to a plot, enabling the comparison of two datasets with different scales in a single visualization.

Q: Secondary axes

Revisit the dataset `colom_hiv_deaths_per_day`.



```
colom_hiv_deaths_per_day
```

```
## # A tibble: 2,543 x 2
##   date_death deaths
##   <date>      <int>
## 1 2010-01-05     1
## 2 2010-01-06     0
## 3 2010-01-07     0
## 4 2010-01-08     0
## 5 2010-01-09     1
## 6 2010-01-10     0
## 7 2010-01-11     1
## 8 2010-01-12     0
```

```
## 9 2010-01-13      0  
## 10 2010-01-14     0  
## # i 2,533 more rows
```



Your task is to create a plot with two y-axes: one for the daily deaths and another for the cumulative deaths in Colombia.

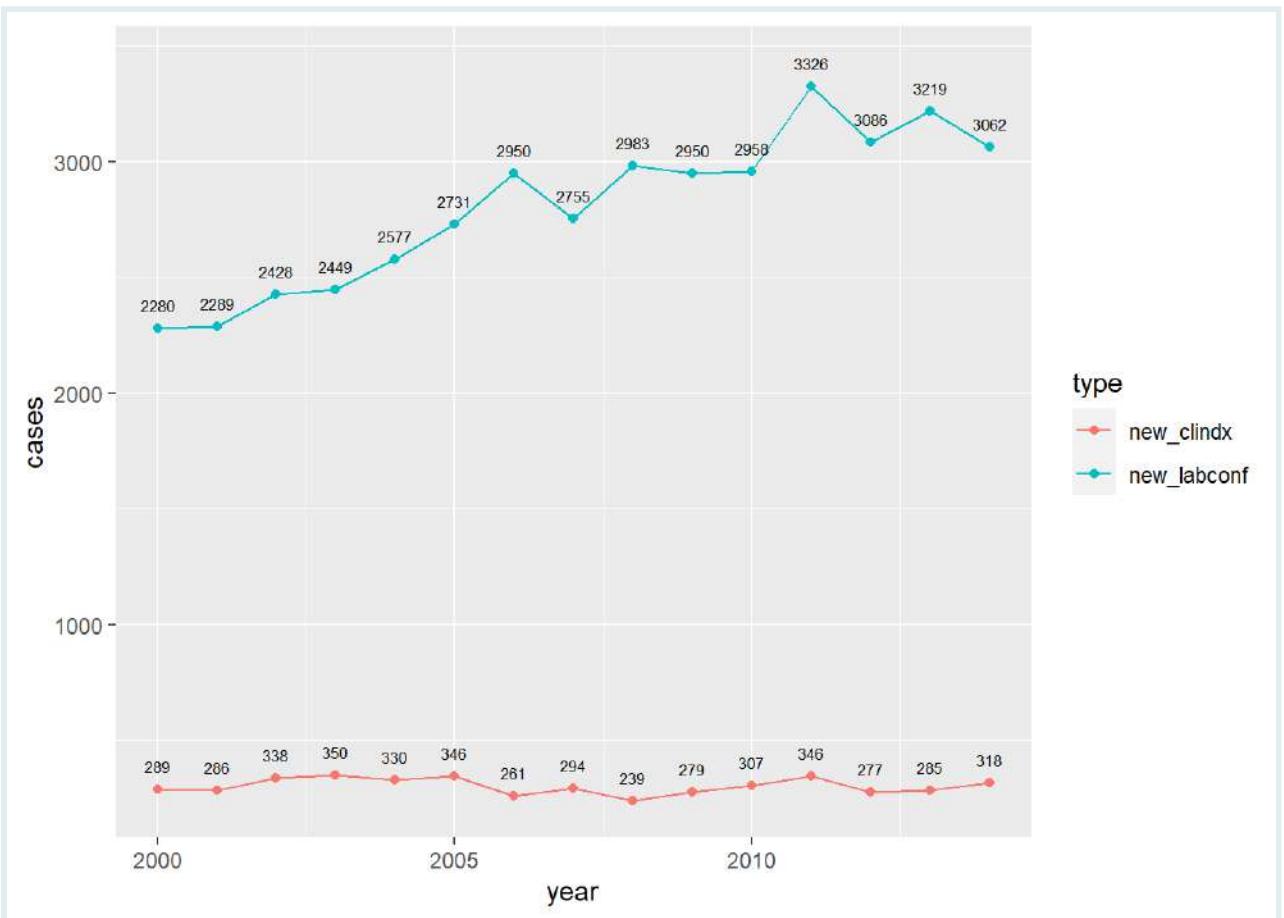
Wrap up

In this lesson, you developed key skills for wrangling, visualizing, and enhancing time series data to uncover and communicate meaningful trends over time. These skills will come in handy as you continue to explore and analyze time series data in R.

Answer Key

Q: Reshaping and Plotting TB Data

```
tb_benin_long <- tb_data_benin %>%  
  pivot_longer(cols = c("new_clindx", "new_labconf")) %>%  
  rename(type = name, cases = value)  
  
ggplot(tb_benin_long, aes(x = year, y = cases, colour = type, group = type)) +  
  geom_line() +  
  geom_point() +  
  geom_text(aes(label = cases), size = 2.2, nudge_y = 100, color = "black")
```



Q: Aesthetic improvements {.unlisted .unnumbered}

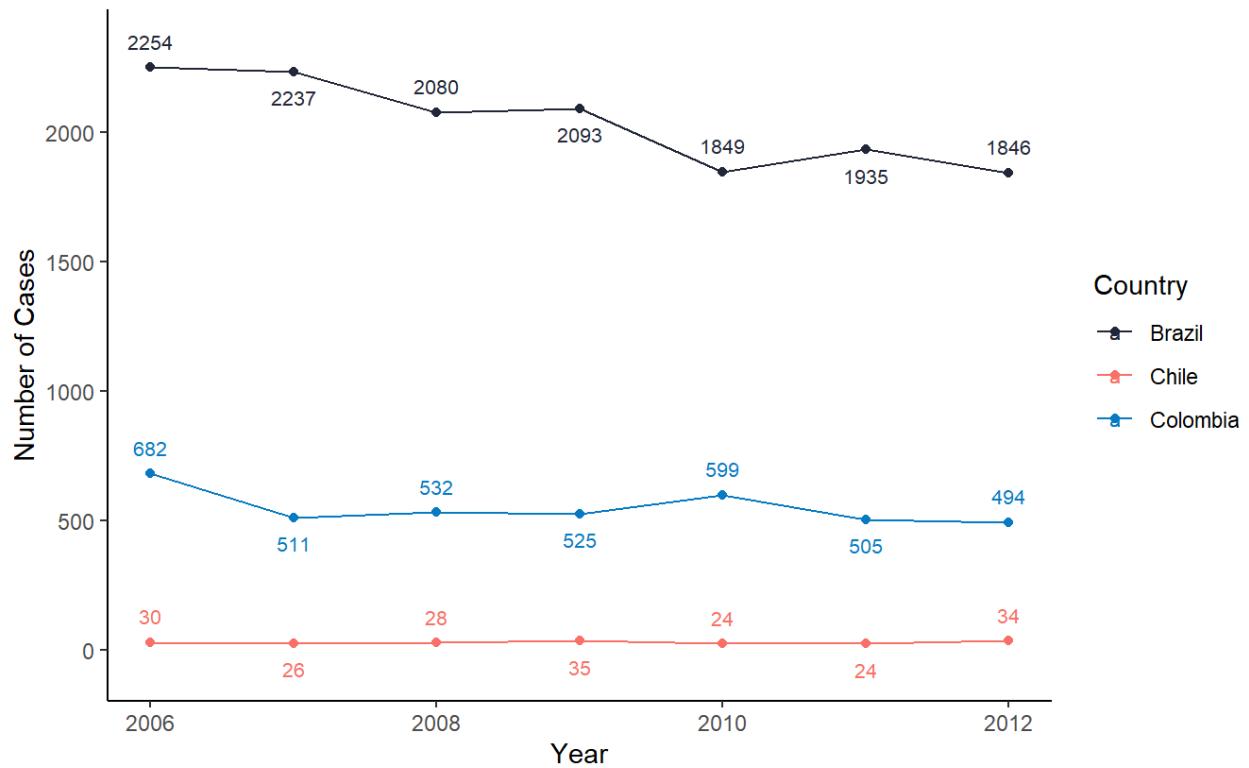
```
even_years_southam <- tb_child_cases_southam %>%
  filter(year %% 2 == 0) # Keep only years that are multiples of 2

odd_years_southam <- tb_child_cases_southam %>%
  filter(year %% 2 == 1) # Keep only years that are not multiples of 2

tb_child_cases_southam %>%
  ggplot(aes(x = year, y = tb_cases_children, color = country)) +
  geom_line() +
  geom_point() +
  geom_text(data = even_years_southam, aes(label = tb_cases_children),
            nudge_y = 100, size = 2.8) +
  geom_text(data = odd_years_southam, aes(label = tb_cases_children),
            nudge_y = -100, size = 2.8) +
  scale_color_manual(values = c("#212738", "#F97068", "#067BC2")) +
  labs(title = "Tuberculosis Notifications in Three South American Countries",
       subtitle = "Child Cases, 1993-2022",
       caption = "Source: World Health Organization",
       x = "Year",
       y = "Number of Cases",
       color = "Country") +
  theme_classic()
```

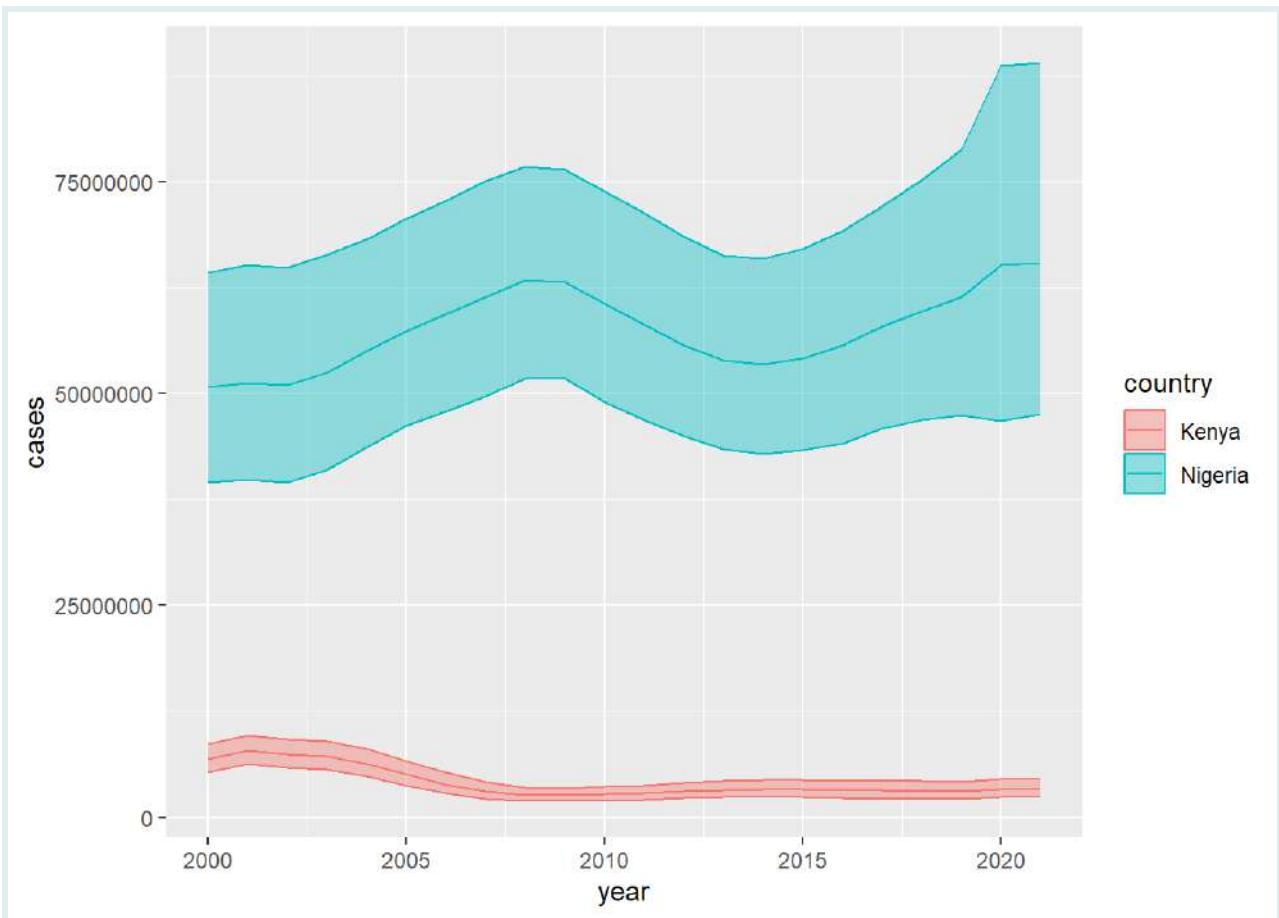
Tuberculosis Notifications in Three South American Countries

Child Cases, 1993-2022



Q: Plotting confidence intervals

```
nig_ken_mal %>%
  separate(malaria_cases,
           into = c("cases", "cases_lower", "cases_upper"),
           sep = "\\\\(|to") %>%
  mutate(across(c("cases", "cases_lower", "cases_upper"),
               ~ str_replace_all(.x, "[^0-9]", "") %>%
                 as.numeric())
        )) %>%
  ggplot(aes(x = year, y = cases, color = country, fill = country)) +
  geom_line() +
  geom_ribbon(aes(ymin = cases_lower, ymax = cases_upper), alpha = 0.4)
```



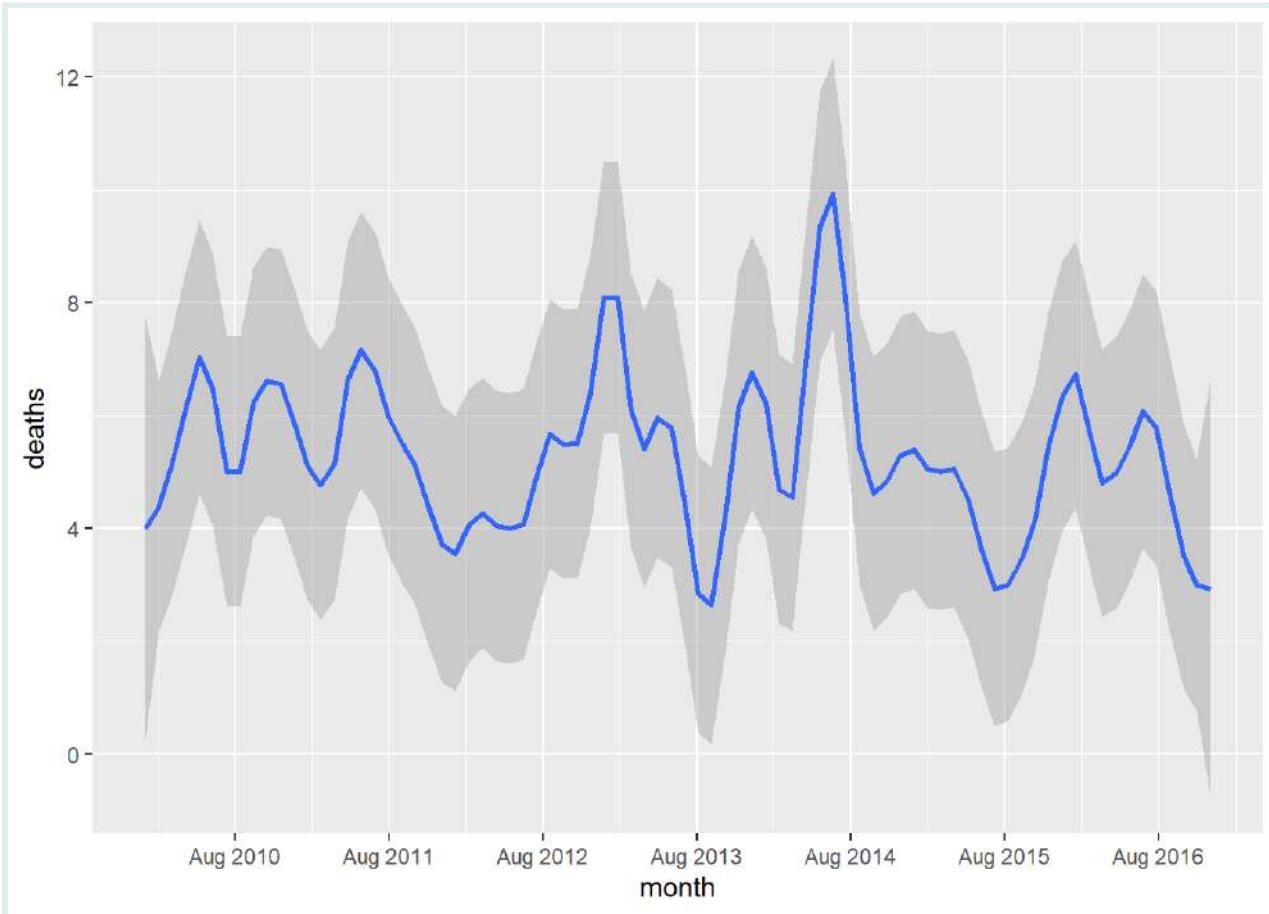
Q: Smoothing HIV Death Data in Colombia

```

hiv_monthly_deaths_table <-
  colom_hiv_deaths %>%
  # Aggregate data to count deaths per month
  mutate(month = floor_date(date_death, unit = "month")) %>%
  group_by(month) %>%
  summarize(deaths = n())

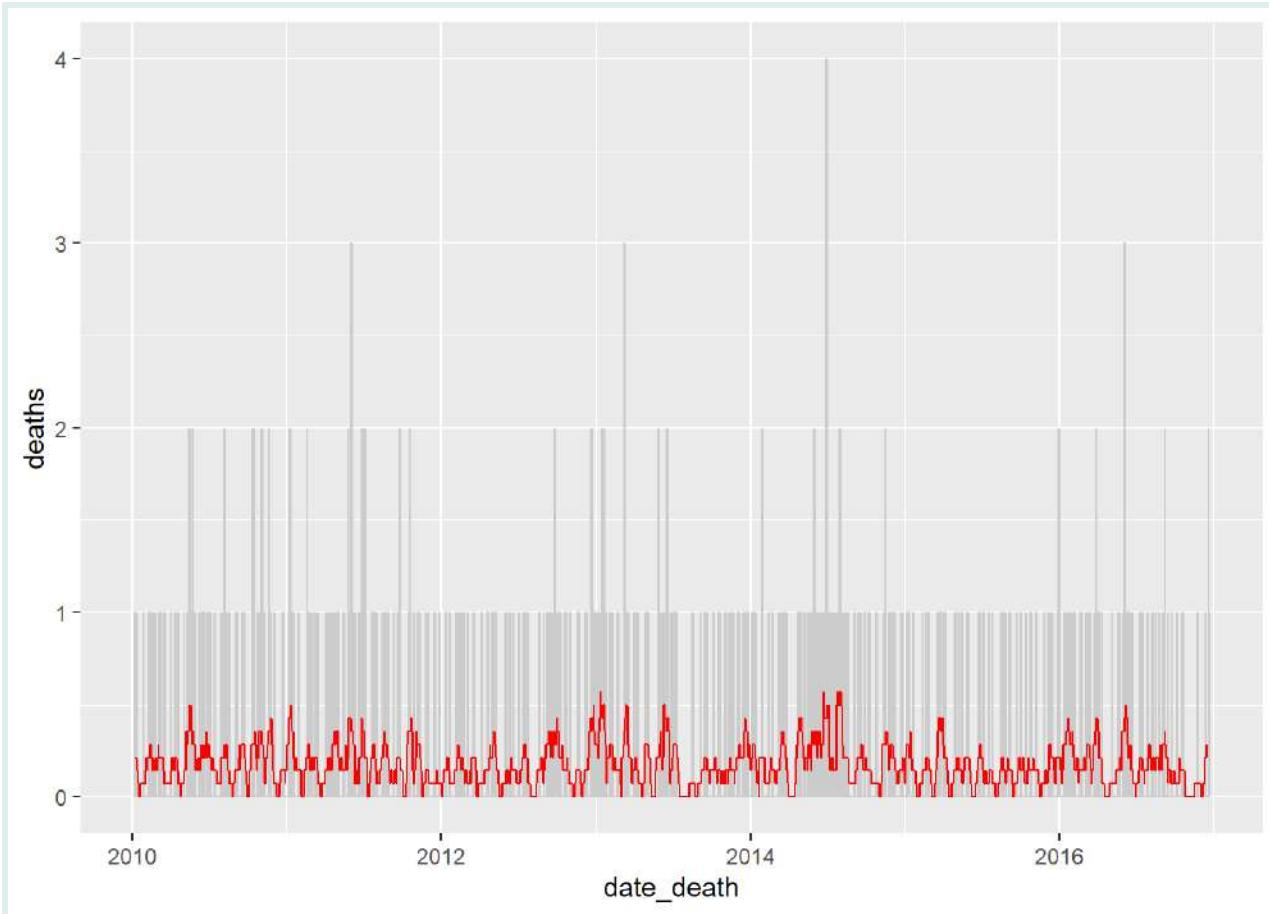
# Create the epicurve
ggplot(hiv_monthly_deaths_table, aes(x = month, y = deaths)) +
  # Apply smoothing to the curve
  geom_smooth(method = "loess", span = 0.1) +
  scale_x_date(date_breaks = "12 months", date_labels = "%b %Y")

```



Q: Smoothing with rolling averages

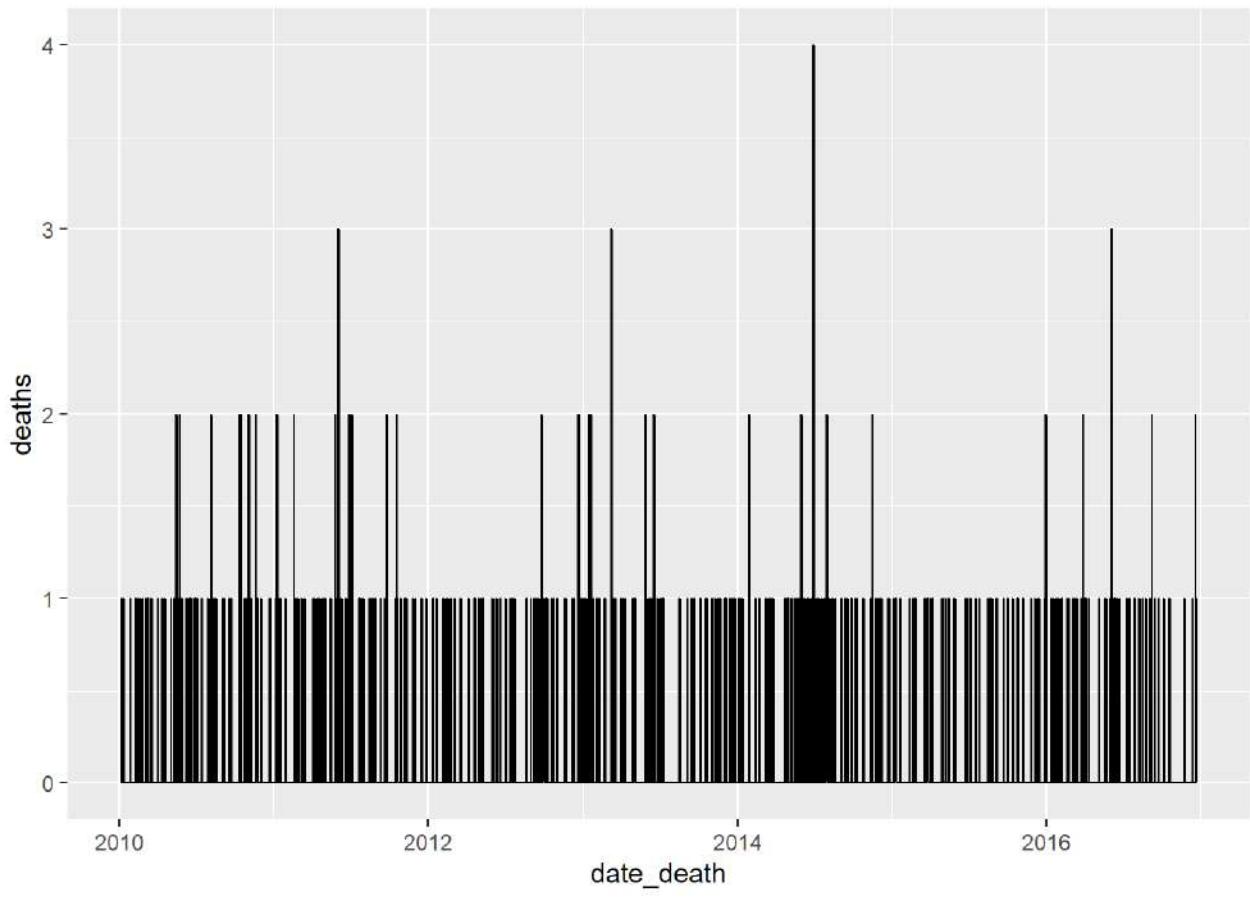
```
colom_hiv_deaths_per_day %>%
  mutate(roll_deaths = rollmean(deaths, k = 14, fill = NA)) %>%
  ggplot(aes(x = date_death, y = deaths)) +
  geom_line(color = "gray80") +
  geom_line(aes(y = roll_deaths), color = "red")
```



Q: Secondary axes

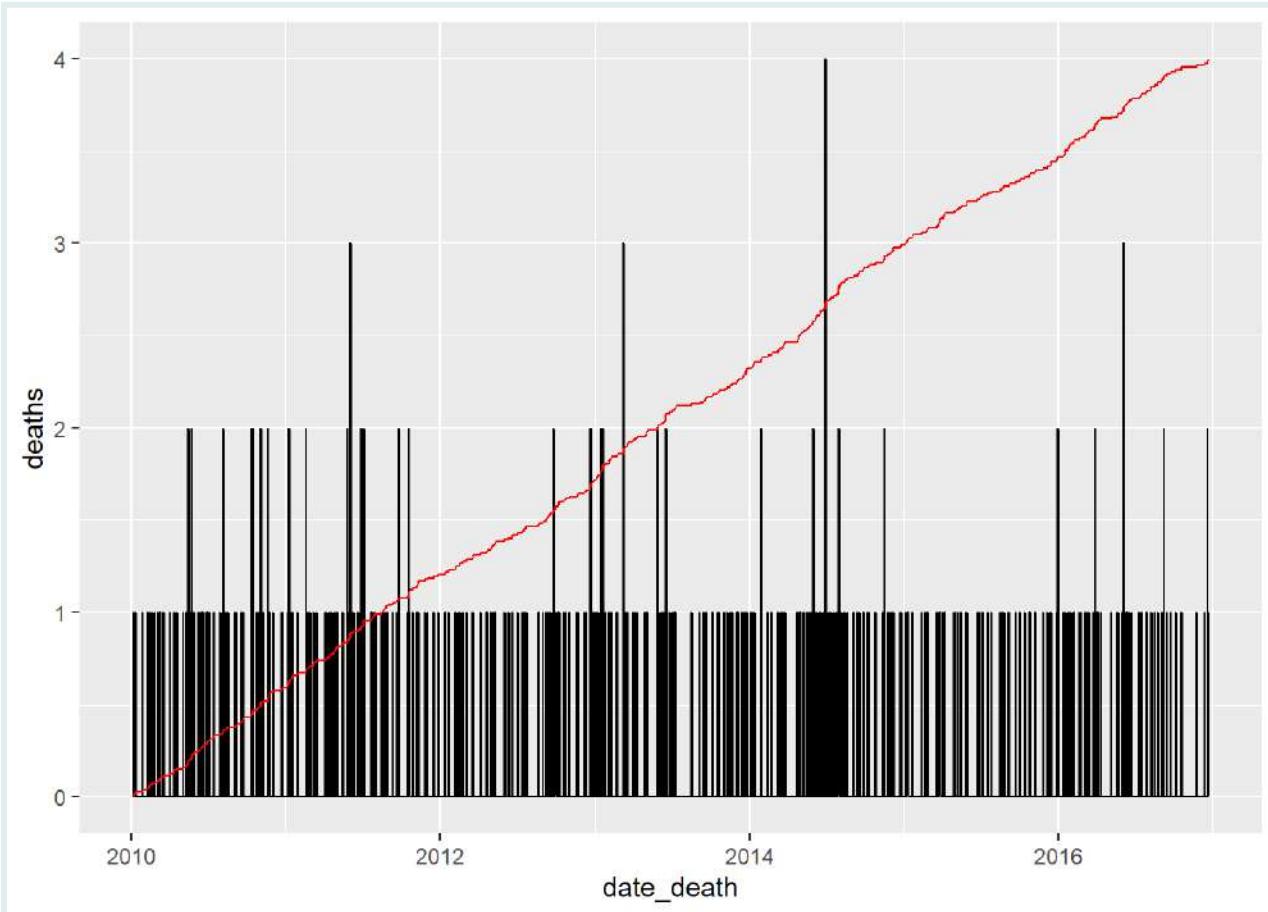
```
# Step 1: Calculate cumulative deaths
colom_hiv_deaths_cumul <- colom_hiv_deaths_per_day %>%
  mutate(cum_deaths = cumsum(deaths))

# Step 2: Plot daily deaths
ggplot(colom_hiv_deaths_cumul, aes(x = date_death)) +
  geom_line(aes(y = deaths))
```

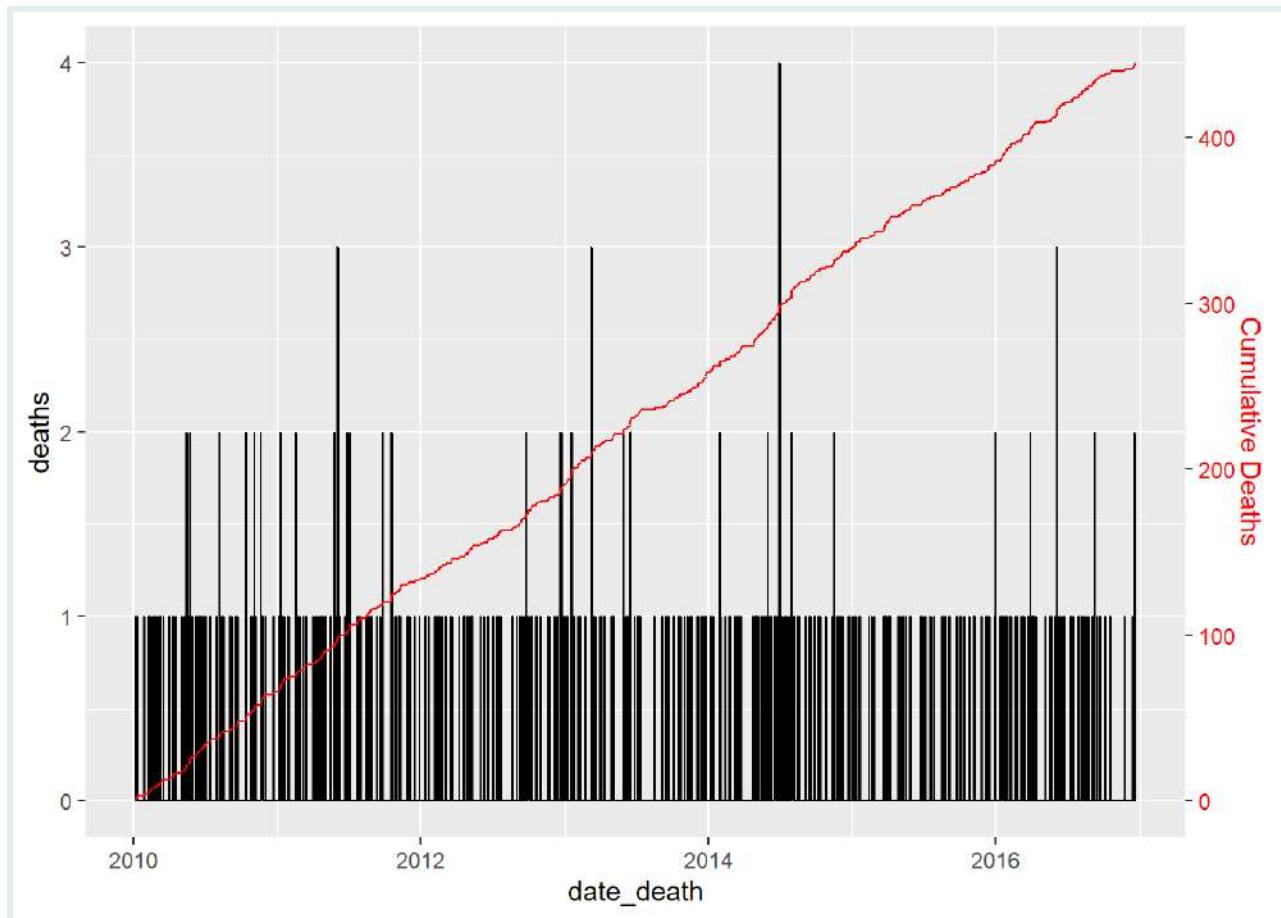


```
# Step 3: Calculate scale factor
scale_factor <- max(colom_hiv_deaths_cumul$cum_deaths) /
  max(colom_hiv_deaths_cumul$deaths)

# Step 4: Add cumulative deaths to the plot
ggplot(colom_hiv_deaths_cumul, aes(x = date_death)) +
  geom_line(aes(y = deaths)) +
  geom_line(aes(y = cum_deaths / scale_factor), color = "red")
```



```
# Step 5: Add secondary y-axis
ggplot(colom_hiv_deaths_cumul, aes(x = date_death)) +
  geom_line(aes(y = deaths)) +
  geom_line(aes(y = cum_deaths / scale_factor), color = "red") +
  scale_y_continuous(sec.axis = sec_axis(trans = ~ .x * scale_factor, name =
    "Cumulative Deaths")) +
  theme(axis.text.y.right = element_text(color = "red"),
    axis.title.y.right = element_text(color = "red"))
```



Contributors

The following team members contributed to this lesson:



IMAD EL BADISY

Data Science Education Officer
Deeply interested in health data



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science



KENE DAVID NWOSU

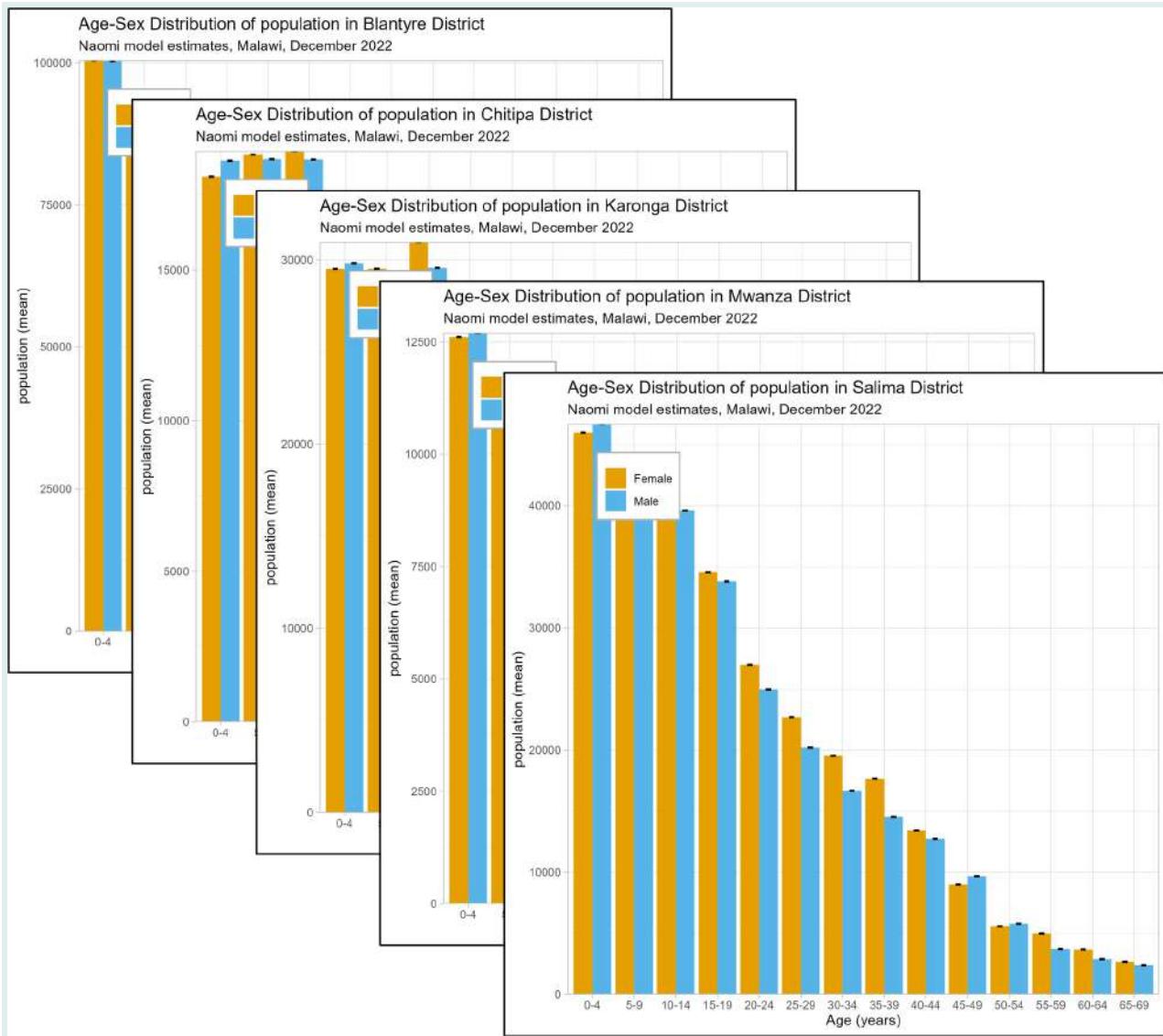
Data analyst, the GRAPH Network
Passionate about world improvement

Automating Data Visualization with `{ggplot2}` and `{purrr}`

Introduction
Packages
Introduction to the data: HIV in Malawi
The challenge of repetitive plotting
Create custom plotting functions
Looping through a vector of variables
Finalize and save
WRAP UP!
Answer Key

Introduction

There are often situations when you need to perform repetitive plotting tasks. For example, you'd like to plot the same kind of data (e.g. the same epidemic indicator) for several states, provinces, or cities. In this lesson, we will explore how to automate data visualizations using the powerful combination of `{ggplot2}` and `{purrr}` in R. First, we'll delve into the programmatic side of `{ggplot2}`, creating custom plotting functions to streamline your graphing tasks. Then we'll use `{purrr}` to iterate through different variables, allowing us to generate and save a multitude of plots in one step! Learning how to automate plotting will greatly enhance your data analysis workflows.



Learning Objectives

1. **Recognize the challenges of repetitive filtering and plotting:** Identify when repetitive plotting is needed and create a workflow involving data subsetting, plotting, and saving.
2. **Create custom plotting functions:** Develop custom functions for repetitive plotting tasks with, including variable and row subsetting. Learn to add multiple inputs for dynamic plot customization.
3. **Iterate plotting tasks:** Apply plotting functions over a vector of variables, with the help of `purrr::map()`.
4. **Use nested loops for automation:** Use `map()` within a for loop to iterate over a combination of subsets and response variables.

By the end of this lesson, you'll have the skills to automate `{ggplot2}` graphs, saving time and enhancing the reproducibility of your data-driven narratives.

Packages

In this lesson we will use the following packages:

- `{tidyverse}` metapackage
 - `{ggplot2}` for creating plots
 - `{purrr}` for iterating functions through a vector
- `{here}` for project-relative file paths
- `{glue}` for concatenating strings and automating plot annotation

```
# Load packages
pacman::p_load(tidyverse, here, glue)
```

Introduction to the data: HIV in Malawi

Today, we will be looking at a dataset of subnational HIV epidemic indicators from the Malawi Ministry of Health Department of HIV & AIDS and Viral Hepatitis, for December 2022. These estimates were derived from a small-area estimation model, called Naomi, to estimate key measures stratified by subnational administrative units, sex, and five-year age groups. The original dataset can be accessed [here](#).

We have prepared a subset of that data to analyse in this lesson:

```

# Import data from CSV
hiv_mwi_agesex <- read_csv(here("data/clean/hiv_mwi_agesex.csv"))

# View data frame
hiv_mwi_agesex

```

- **Geographic area:**
 - area_level - administrative unit (country, region, or district)
 - area_name - name of the geographic area
- **Demographic information**
 - age_group and sex
 - **HIV indicators:** total population, people living with HIV (PLHIV), HIV prevalence, incidence, ART coverage, and PLHIV who are aware of their status.
 - indicator - short code
 - indicator_label - full name
- **Statistical measures:** model estimates with probabilistic uncertainty
 - mean, lower, upper

KEY POINT



The Naomi model synthesizes data from multiple data sources to give small-area estimates of key HIV indicators for sub-Saharan Africa. These estimates are instrumental for HIV program planning, resource allocation, and target setting. You can learn more about the Naomi model [here](#).

Visualizing age-sex distribution

Age and sex disparities in HIV prevalence are observed globally and are affected by a multitude of overlapping factors, including gender discrimination, sexual behavior, and access to healthcare and education. These factors can influence both the likelihood of initial infection and the outcomes for people living with HIV.

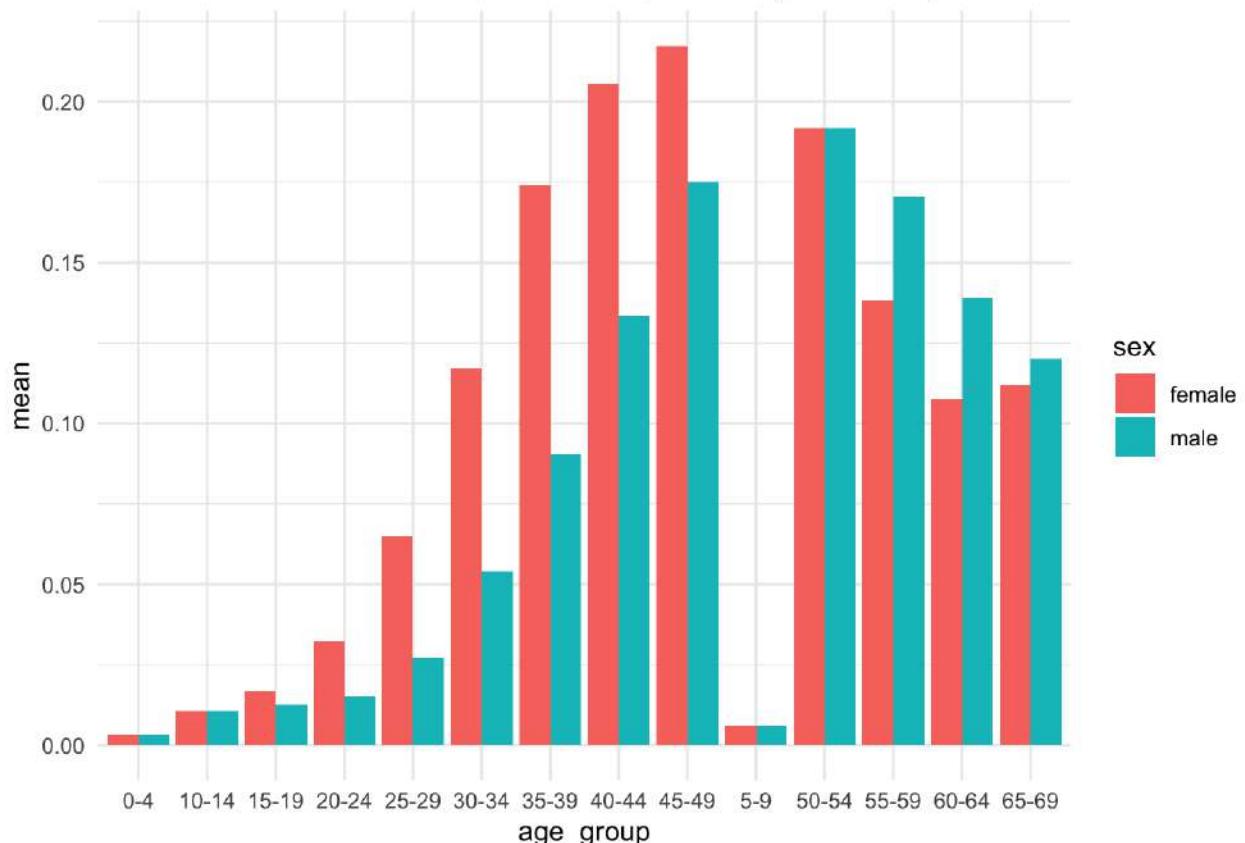
In this lesson we will focus on visualizing the age-sex distribution of various indicators at the national, regional, and district level. First let's use `ggplot()` to create a national-level bar chart of HIV prevalence, grouped by age and split by sex.

```

# Bar chart of age-sex distribution
hiv_mwi_agesex %>%
  filter(area_level == "Country",
        indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  labs(title = 'National estimates of HIV prevalence, Malawi (Dec
2022)')

```

National estimates of HIV prevalence, Malawi (Dec 2022)



Oops! Something looks off with the order of age groups. This is important to fix because we will be creating plots grouped by age and sex for the rest of this lesson.

Reveling the x-axis variable

The variable **age_group** is a **character** vector, which is not inherently ordered the way a **factor** is.



```
# View unique values the age_group variable  
hiv_mwi_agesex %>% pull(age_group) %>% unique()
```

```
## [1] "0-4"    "5-9"    "10-14"   "15-19"   "20-24"   "25-29"   "30-  
34"    "35-39"   "40-44"   "45-49"   "50-54"  
## [12] "55-59"  "60-64"  "65-69"
```

If we look at the unique values of the variable, we get a vector that is correctly ordered from youngest to oldest. However, {ggplot2}

arranges character variables “alphabetically”, which means our “5-9” age group is plotted in the wrong place.

To arrange our bar plot the correct age sequence, we can convert `age_group` to a factor and specify the order of levels using `forcats::fct_relevel()`:



```
# Create a vector of correctly ordered age group values
ordered_age_groups <- hiv_mwi_agesex %>%
  pull(age_group) %>% unique()

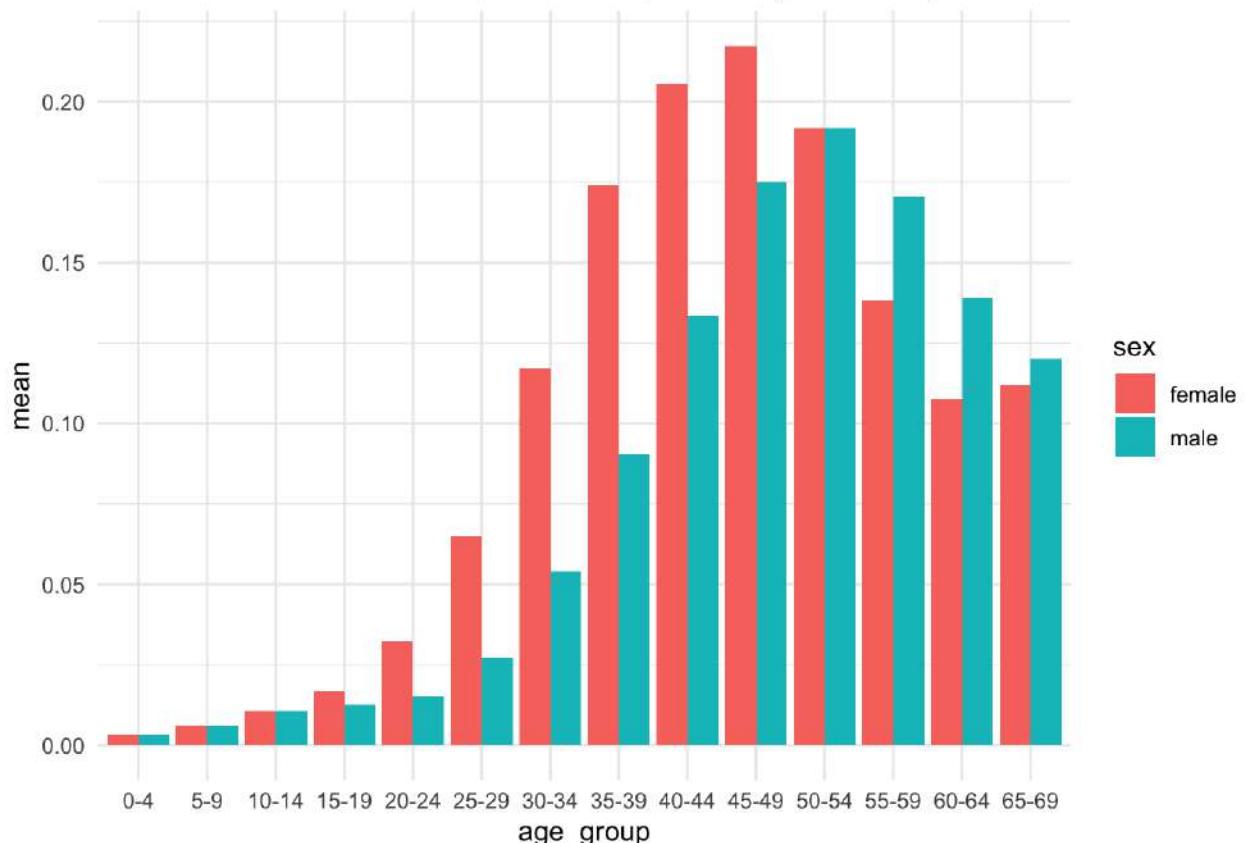
# Reorder age_group levels and save to a new data frame
hiv_malawi <- hiv_mwi_agesex %>%
  mutate(age_group = forcats::fct_relevel(age_group,
  ordered_age_groups))
```

Now we are ready to plot age distributions with our new `hiv_malawi` data frame.

Let's try the same `ggplot()` code again, with `hiv_malawi`:

```
# Plot with leveled age groups
hiv_malawi %>%
  filter(area_level == "Country",
        indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  labs(title = "National estimates of HIV prevalence, Malawi (Dec
2022)")
```

National estimates of HIV prevalence, Malawi (Dec 2022)



Much better!

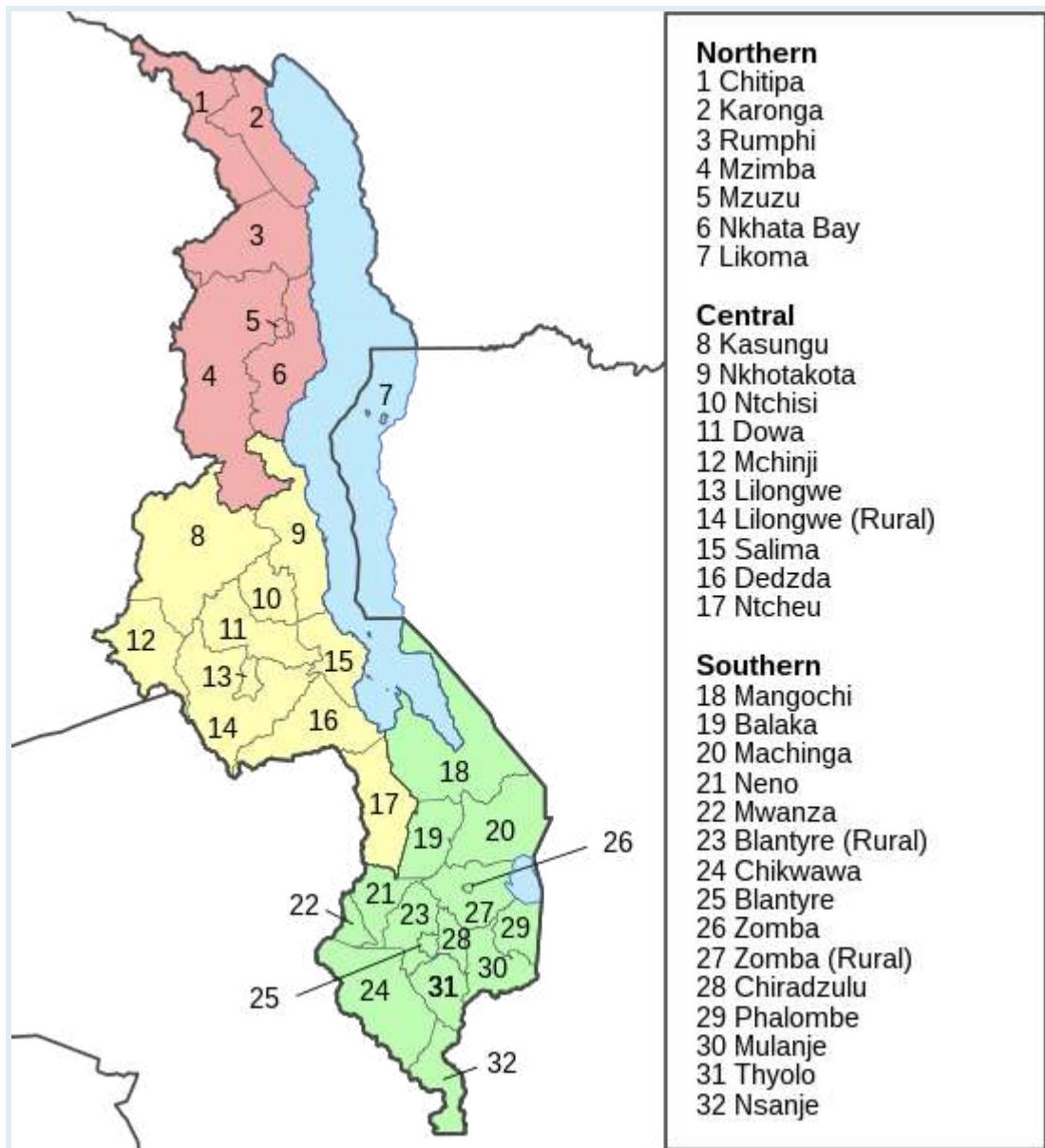
Insights on HIV prevalence

SIDE NOTE



The graph reveals a discernible gender disparity in HIV prevalence starting at age 15, likely tied to the onset of sexual activity. Women face a significantly higher prevalence than men through their 20s to 40s, which may reflect factors such as biological vulnerability and social dynamics. Interestingly, this trend reverses after the 50s, where men show higher rates. This shift could be influenced by men's sexual behavior, mortality rates, and access to or seeking of treatment.

Now, let's dig deeper and investigate if the same trends are observed when we zoom into more localized areas. We can filter the data to plot the age-sex distribution at different geographic areas. Our dataset includes estimates aggregated for the 3 main regions and 28 districts of Malawi. We'll start by focusing on Malawi's first administrative level – the **three regions**.



Malawi map of regions and districts

The challenge of repetitive plotting

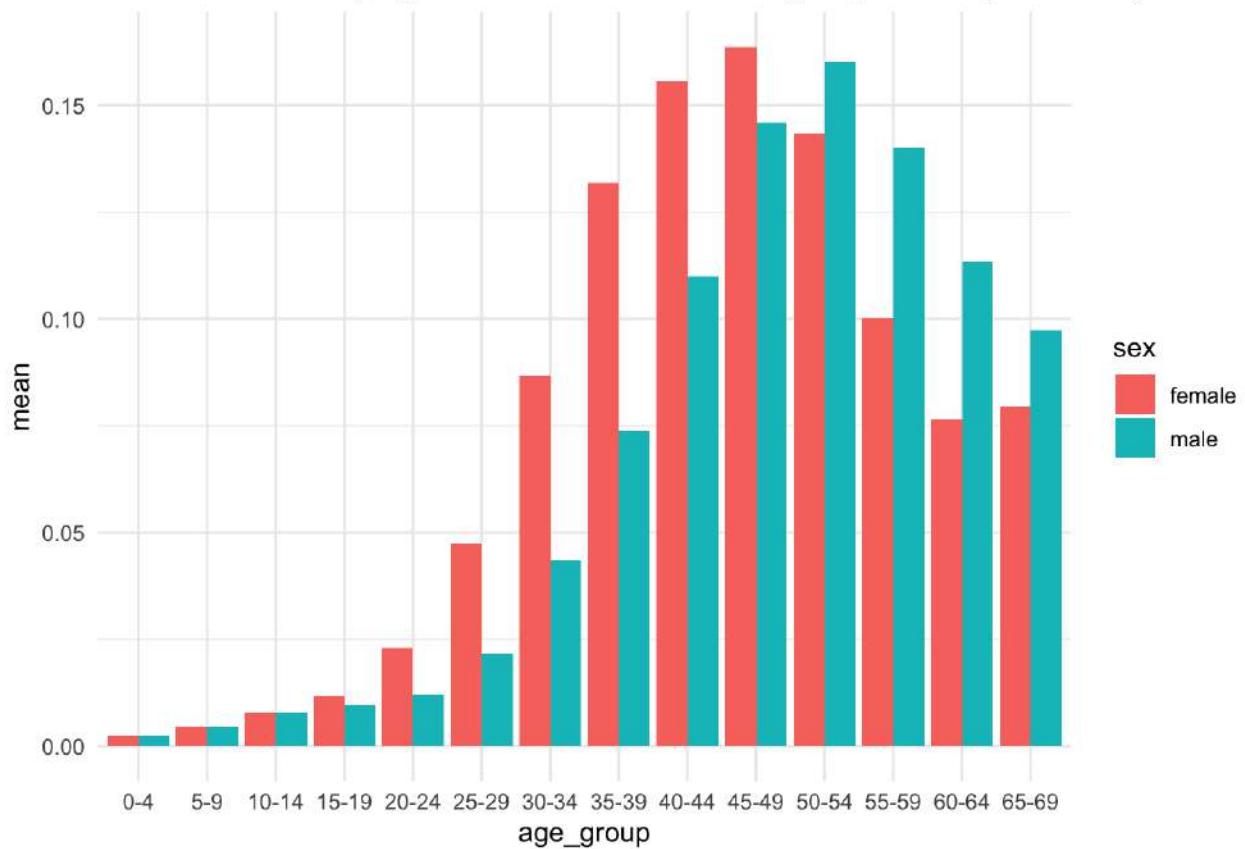
In this section, we will highlight a common challenge in data visualization: repetitive filtering of data subsets.

Let's first create a plot of HIV prevalence for the "Northern" region of Malawi. We can copy the code for the national plot we made earlier, and replace "Country" with "Region", and "Malawi" with the name of the Region we want to plot.

```
# Example of repetitive filtering and plotting - Region 1

hiv_malawi %>%
  # Filter to Northern Region
  filter(area_level == "Region",
         area_name == 'Northern',
         indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  # Change the title
  labs(title = "HIV Prevelance by age and sex in Northern Region,
Malawi (Dec 2022)")
```

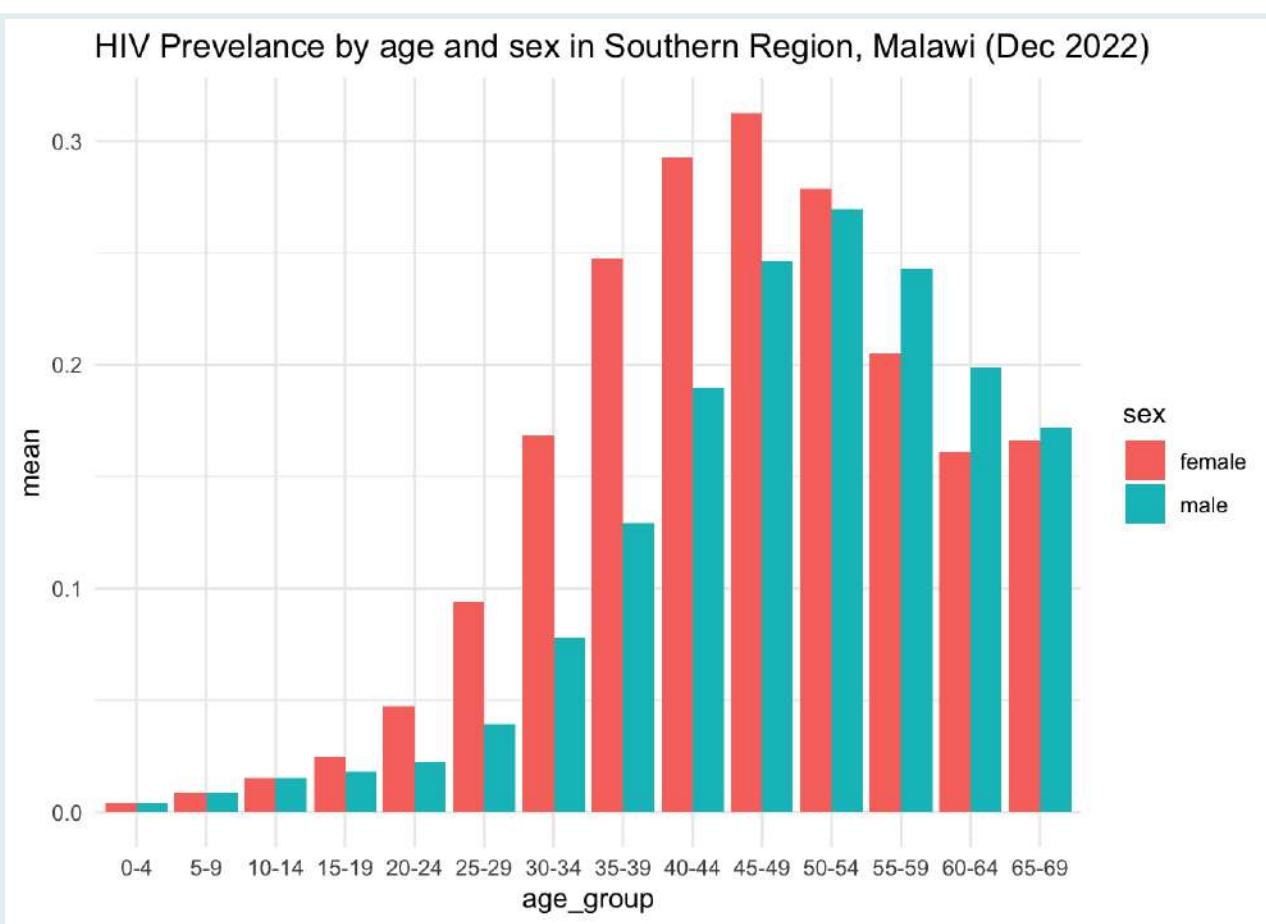
HIV Prevelance by age and sex in Northern Region, Malawi (Dec 2022)



Now let's repeat this for the other two regions:

```
# Example of repetitive filtering and plotting - Region 2

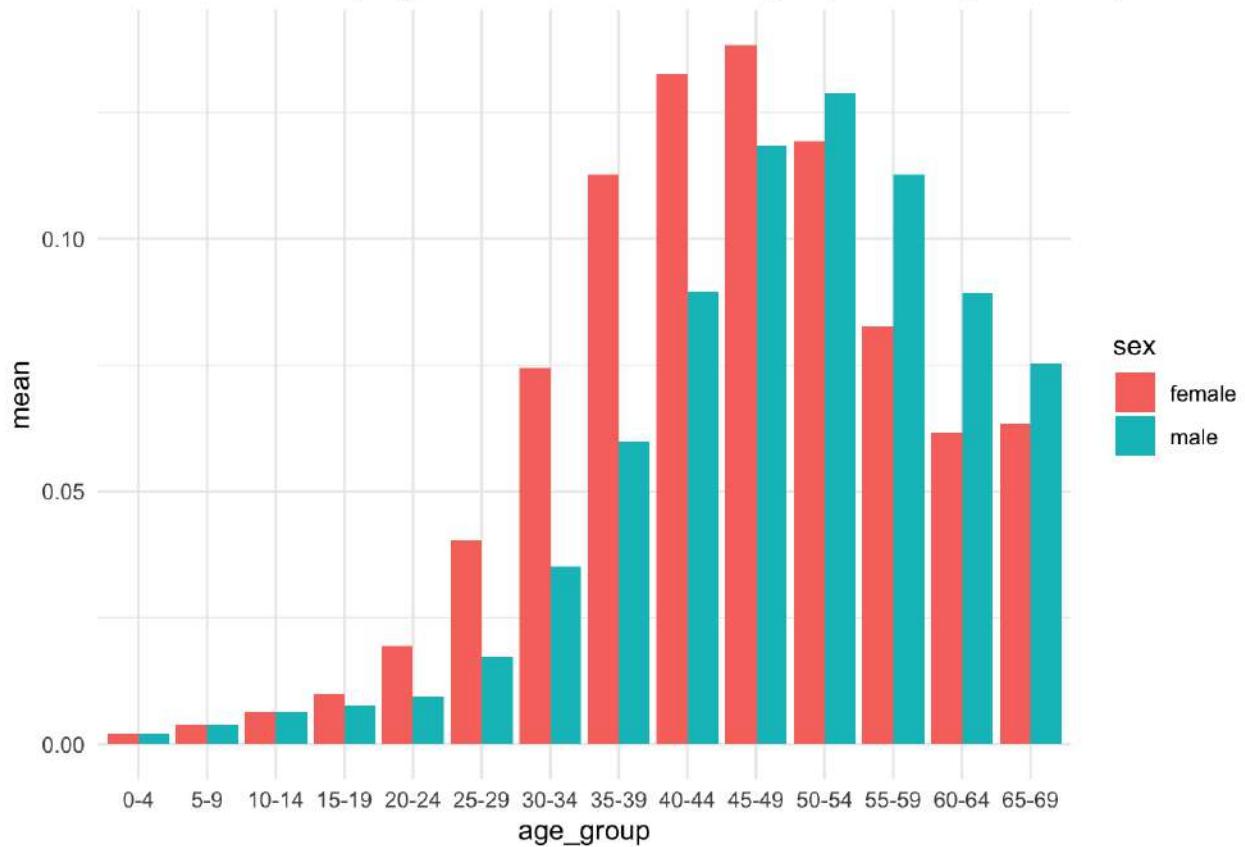
hiv_malawi %>%
  # Filter to Southern Region
  filter(area_level == "Region",
         area_name == 'Southern',
         indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  # Change the title
  labs(title = "HIV Prevelance by age and sex in Southern Region,
Malawi (Dec 2022)")
```



```
# Example of repetitive filtering and plotting - Region 3

hiv_malawi %>%
  # Filter to Central Region
  filter(area_level == "Region",
         area_name == 'Central',
         indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  # Change the title
  labs(title = "HIV Prevelance by age and sex in Central Region,
Malawi (Dec 2022)")
```

HIV Prevelance by age and sex in Central Region, Malawi (Dec 2022)



While the above method of copying and replacing names works for a small number of subgroups, the limitations of manual filtering become evident as the number of subgroups increases. Imagine doing this for each of the 28 districts in the data - it would be highly inefficient and an error-prone process!

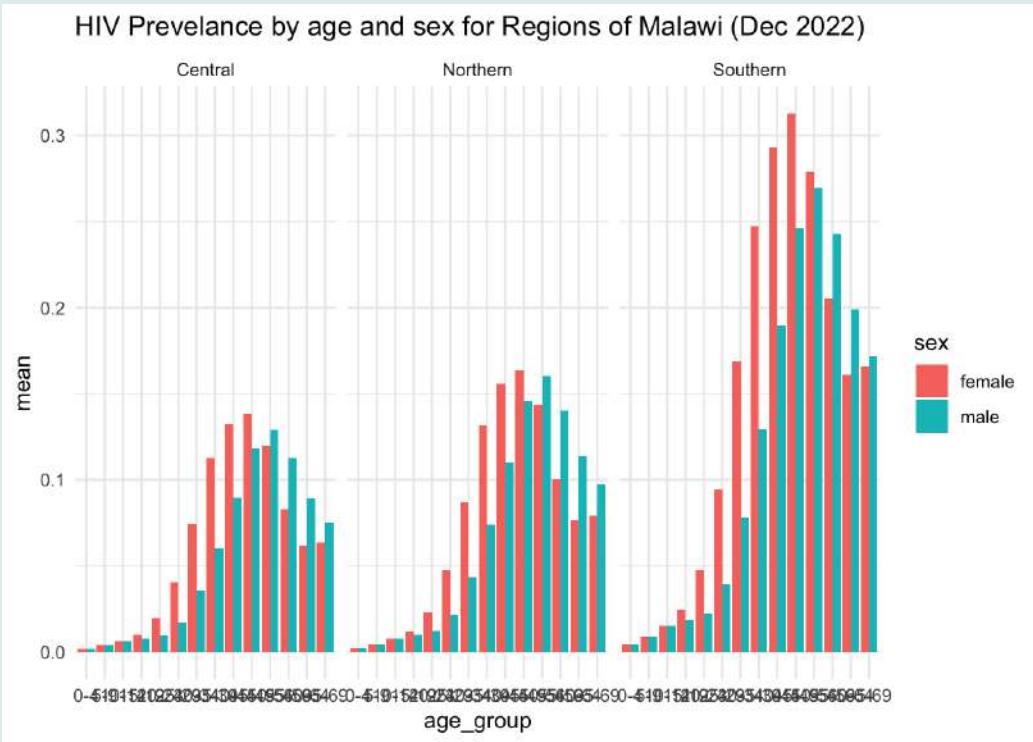
Instead of copying and pasting the same code and adjusting the variables, we'll demonstrate how to handle this challenge in the next sections, using a combination of `{ggplot2}` and functional programming techniques.

Faceting for small multiples

Another option is to create a faceted plot, broken down by region or district. Again, the limitation is that there can be too many levels in your grouping variable, or too much data to fit in each subset. A plot made up of 28 facets would be crowded and virtually unreadable.

```
# Example of faceted subplots by region
hiv_malawi %>%
  filter(area_level == "Region",
        indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  labs(title = "HIV Prevelance by age and sex for
Regions of Malawi (Dec 2022)") +
  facet_wrap(~area_name)
```

SIDE NOTE



Other times we may generate individual plots for separate documents or slides. Instead of sticking solely to faceting, we can create functions that allow us to create a series of plots systematically.

Question 1: Filtering and plotting



Make an age-sex bar plot of **ART coverage** in **Lilongwe** district. You can start with the code we used to create the regional plots. This time, you will need to filter the data to **District**, and adjust the plot title accordingly.

Now adapt your code to create the same plot for **Mzimba** district.

Create custom plotting functions

Single-argument function

The first step to automating our plotting is to create a small function that filters the data for us, and plots the subsetted data.

For example, to visualize the mean prevalence for a region, we can **define a function** that takes a subset condition as an input argument and creates the regional plot with the filtered data.

```
# Simple function for filtering to region and plotting age sex grouped  
bar chart  
  
plot_region<- function(region_name){  
  # copy the code from above and replace area name with a placeholder  
  hiv_malawi %>%  
    filter(area_level == "Region",  
          area_name == {{region_name}},  
          indicator == "prevalence") %>%  
    ggplot(aes(x = age_group,  
               y = mean,  
               fill = sex)) +  
    geom_col(position = "dodge") +  
    theme_minimal() +  
    labs(title = {{region_name}})  
}
```

The code inside the user-defined function above is basically the same as the one to create the previous chart. The only difference is that we do not specify a specific region name, but create a **placeholder**, here called **{{region_name}}**, to control the filtering condition and title. In the **filter()** function, we subset our data based on **region_name**; in the **labs()** function we use this string as the plot title.

WATCH OUT

Curly curlyies

WATCH OUT

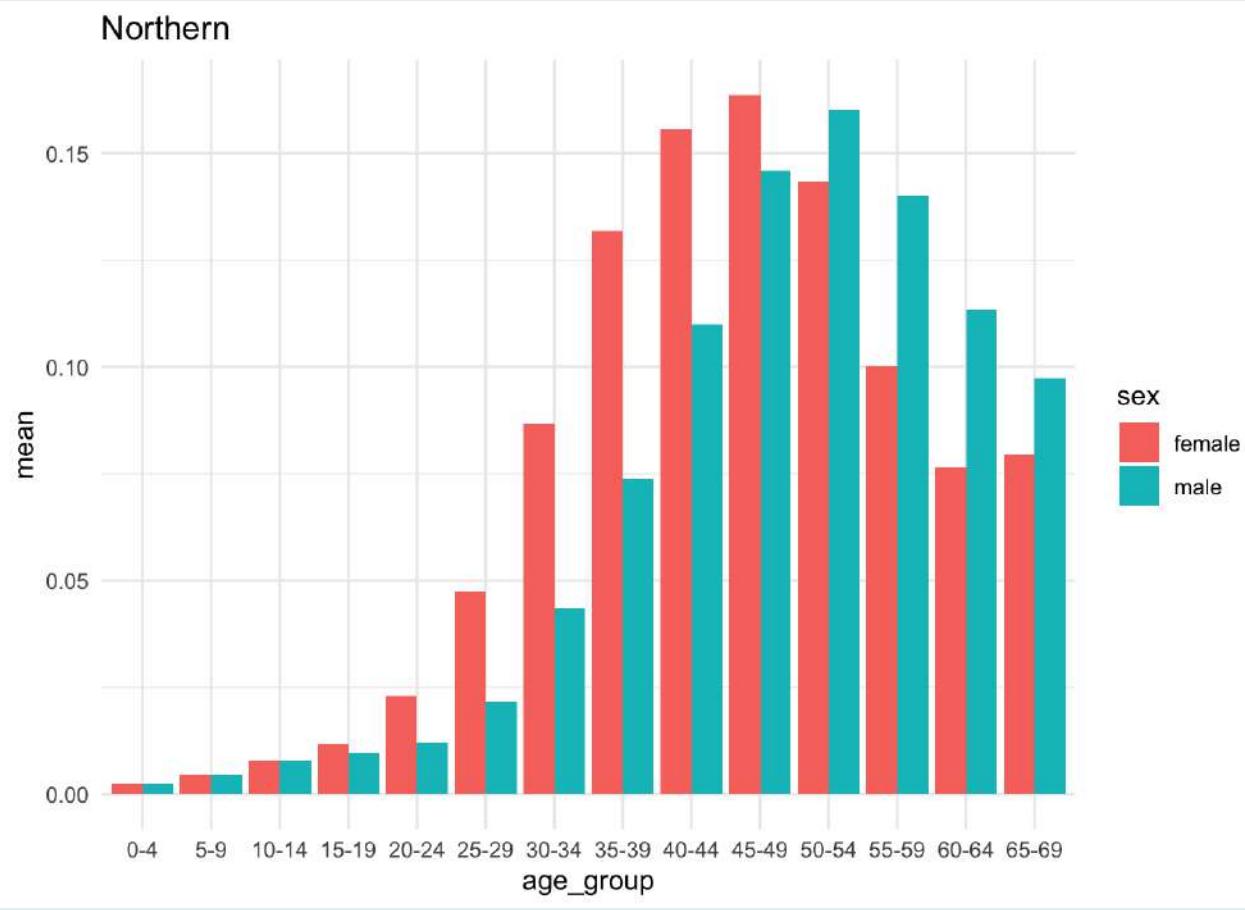


Notice the use of curly braces `{ }{ }` inside the `plot_region()` function. This practice is recommended when using `{tidyverse}` functions inside another custom function, to avoid errors. See [here](#) for more examples.

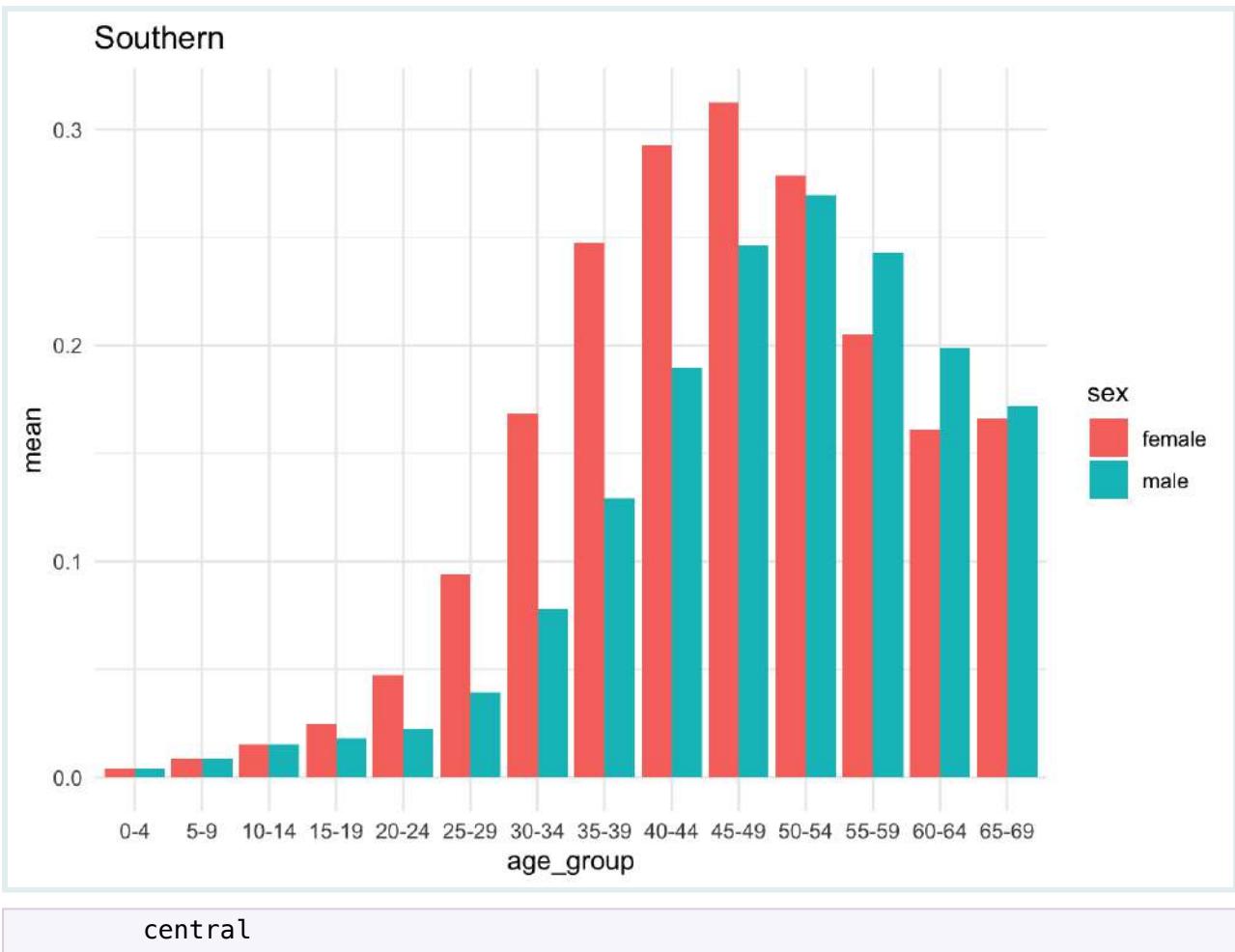
Now let's run the function for each region featured in the data set, and see what we get!

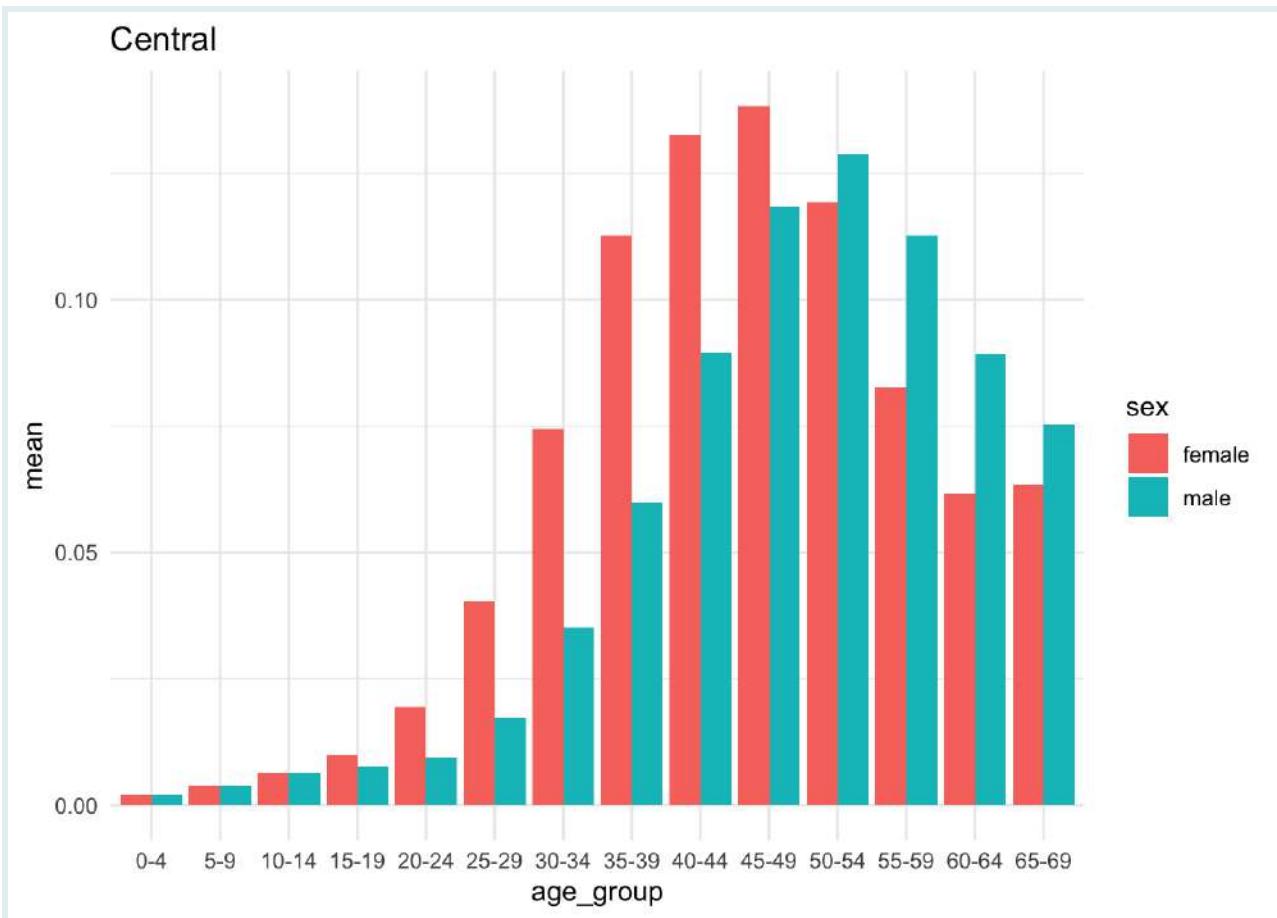
```
# Create individual plots for the three regions
northern <- plot_region('Northern')
southern <- plot_region('Southern')
central <- plot_region('Central')

# Print plots
northern
```



southern





You can see that using a custom function is much more efficient than repeating and editing the same code chunk. If changes are necessary, we don't need to alter the code for each individual plot. Instead, we can make a small adjustment to our `plot_region()` function.

These plots show that the age-sex patterns of HIV prevalence holds the same at national and regional levels. But you can see that overall prevalence is much higher in the Southern region than others.

Customizing titles with `glue()`

PRO TIP



The plots generated by our custom function look *almost* exactly the same as the ones before - can you spot one difference? That's right, the title! Instead of just "Central", we want it to say 'Central region estimates of HIV prevalence'.

We can fix that with the `glue()` function inside our custom function:

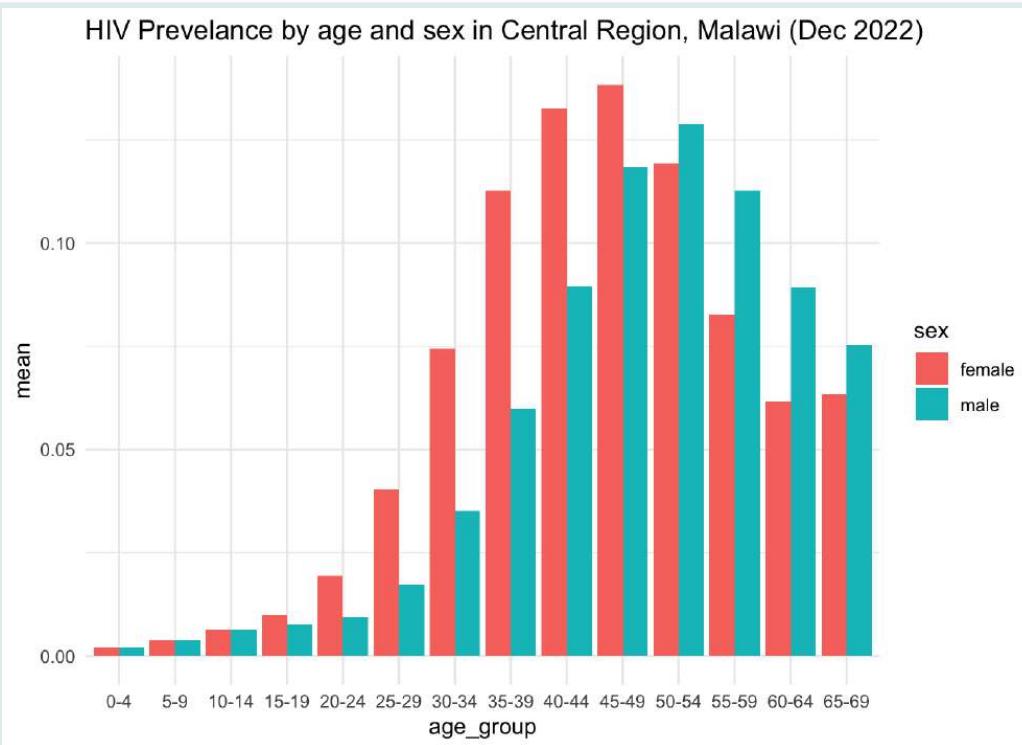
```

# Adapt function to include custom title
plot_region2 <- function(region_name){
  hiv_malawi %>%
    filter(area_level == "Region",
          area_name == {{region_name}},
          indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
    geom_col(position = "dodge") +
    theme_minimal() +
    labs(title = glue("HIV Prevelance by age and sex in
{region_name} Region, Malawi (Dec 2022)"))
}

# Test function
plot_region2("Central")

```

PRO TIP



PRACTICE (in RMD)

Question 2: Custom plotting function for districts

- Create custom function called **plot_district()** which takes **district_name** as an input, and creates an age-sex plot of **persons living with HIV** (the “plhiv” indicator), at the district level. Use **glue()** to create a custom title.



Use your function to create a plot for the **Chitipa** and **Neno** districts.

Multiple input function

In the previous section, our `plot_region()` function accepted just one input: `region_name`, and filtered data to the “Region” level only.

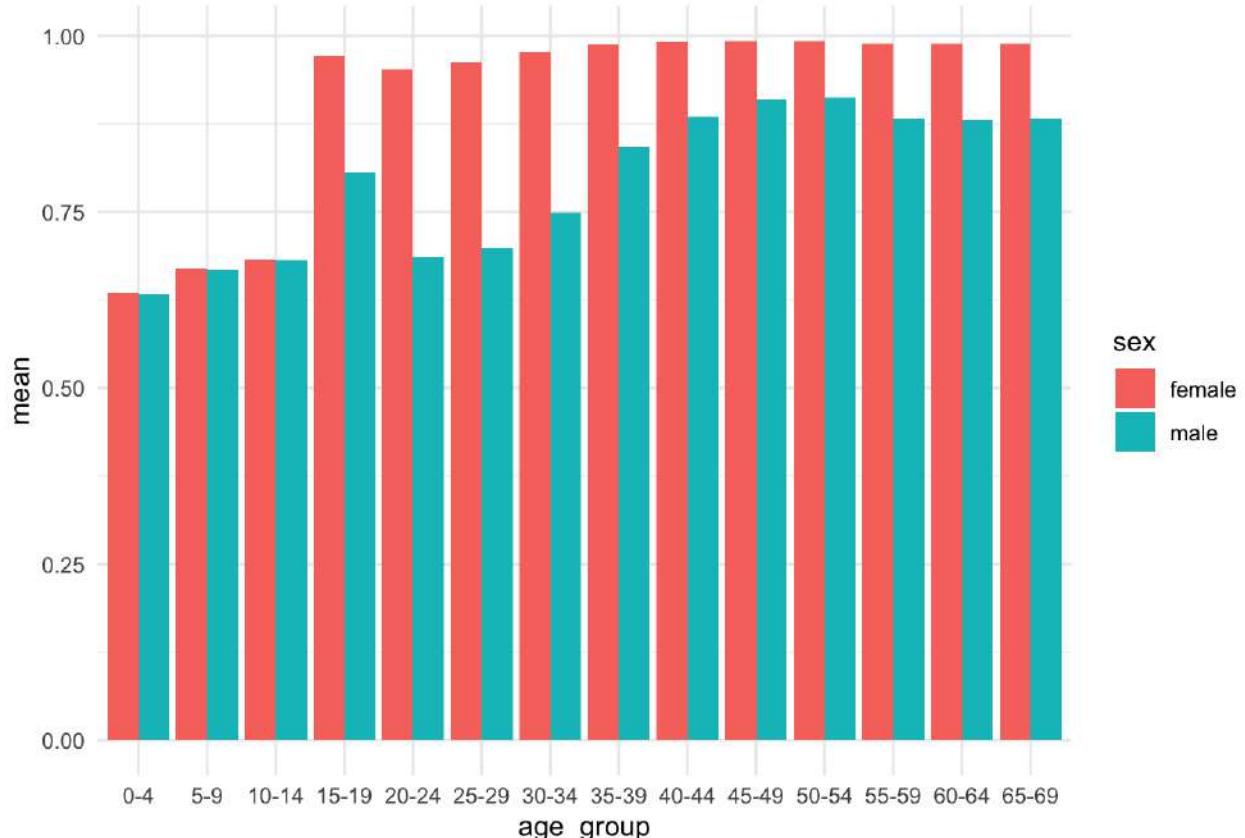
We can make our function even more versatile by allowing us to customize the HIV indicator to plot on the y-axis, and whether to filter by “Region” or “District”.

```
# Create custom function with multiple inputs
plot_malawi <- function(area_name, area_level, hiv_indicator){
  hiv_malawi %>%
    # filter by 3 conditions
    filter(
      area_level == {{area_level}},
      area_name == {{area_name}},
      indicator == {{hiv_indicator}}) %>%
    ggplot(aes(x = age_group,
               y = mean,
               fill = sex)) +
    geom_col(position = "dodge") +
    theme_minimal() +
    # custom title
    labs(title = glue("Mean {hiv_indicator} by age group in {area_name}
{area_level}, Malawi (Dec 2022)"))
}
```

Now we can apply the new custom function `plot_malawi()` to any indicator, at any geographic level in our dataset when we specify the 3 required inputs.

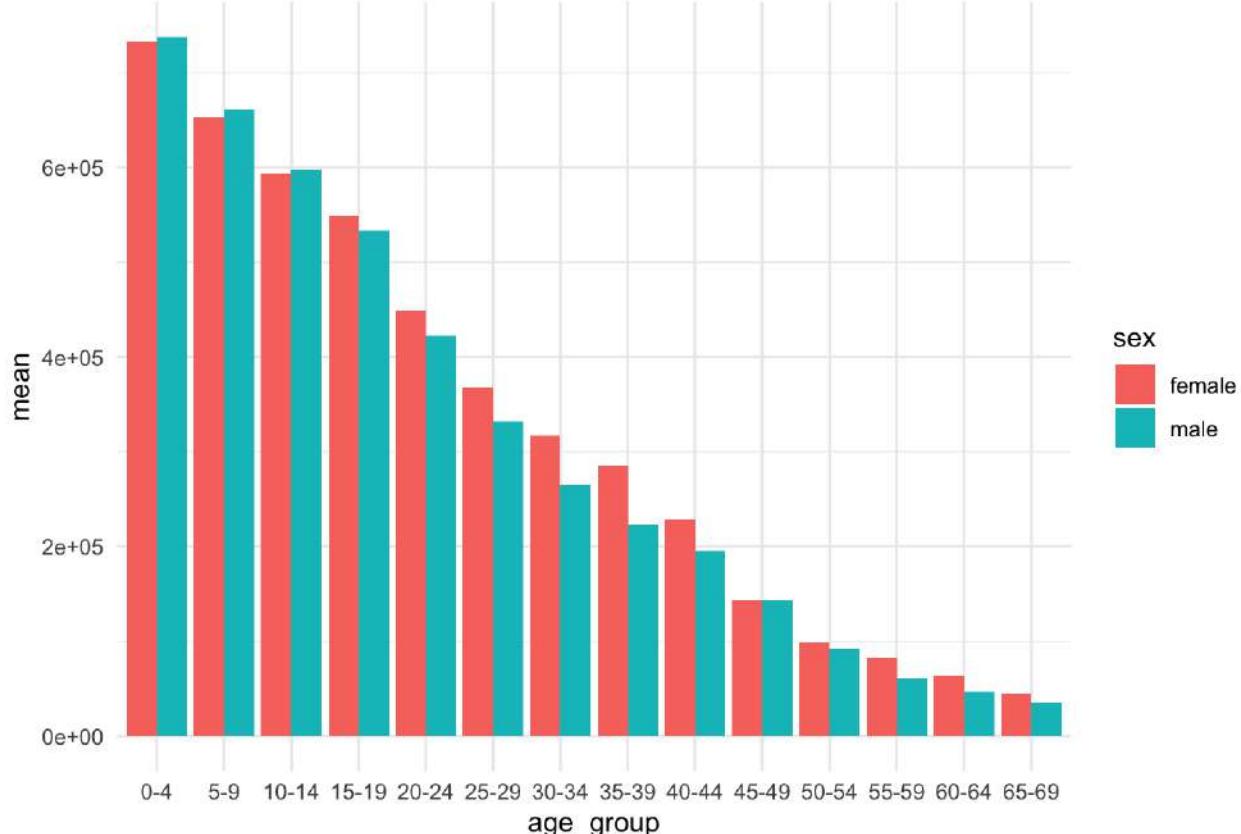
```
# ART coverage for a district
plot_malawi("Chitipa", "District", "art_coverage")
```

Mean art_coverage by age group in Chitipa District, Malawi (Dec 2022)



```
# Population for a region  
plot_malawi("Southern", "Region", "population")
```

Mean population by age group in Southern Region, Malawi (Dec 2022)



Filtering to area level

SIDE NOTE



The reason we added `area_level` is to avoid a situation where a district and region share the same name. We don't have such cases in this dataset, however it is not uncommon for states/provinces to have the same name as a prominent district/city within its borders (e.g., New York city is in New York state). As an added bonus, it allows us to customize the title of our plot to mention area level.

Using custom functions, we can create plots for different regions, districts, and indicators without the need to copy-paste the `{ggplot2}` code and make multiple adjustments manually.

But this is still repetitive! It still requires some copying and pasting and replace the names. Even though it's just one line, it's still not automated!

For example, if we wanted to use our custom function to create a plot of PLHIV for each of the 28 districts, we'd have to do this:

```

# Apply custom function to each district
chitipa_plhiv <- plot_malawi("Chitipa", "District", "plhiv")
karonga_plhiv <- plot_malawi("Karonga", "District", "plhiv")
nkhatabay_plhiv <- plot_malawi("Nkhatabay", "District", "plhiv")
rumphi_plhiv <- plot_malawi("Rumphi", "District", "plhiv")
mzimba_plhiv <- plot_malawi("Mzimba", "District", "plhiv")
likoma_plhiv <- plot_malawi("Likoma", "District", "plhiv")
kasungu_plhiv <- plot_malawi("Kasungu", "District", "plhiv")
nkhotakota_plhiv <- plot_malawi("Nkhotakota", "District", "plhiv")
ntchisi_plhiv <- plot_malawi("Ntchisi", "District", "plhiv")
dowa_plhiv <- plot_malawi("Dowa", "District", "plhiv")
salima_plhiv <- plot_malawi("Salima", "District", "plhiv")
lilongwe_plhiv <- plot_malawi("Lilongwe", "District", "plhiv")
mchinji_plhiv <- plot_malawi("Mchinji", "District", "plhiv")
dedza_plhiv <- plot_malawi("Dedza", "District", "plhiv")
ntcheu_plhiv <- plot_malawi("Ntcheu", "District", "plhiv")
mangochi_plhiv <- plot_malawi("Mangochi", "District", "plhiv")
machinga_plhiv <- plot_malawi("Machinga", "District", "plhiv")
zomba_plhiv <- plot_malawi("Zomba", "District", "plhiv")
mulanje_plhiv <- plot_malawi("Mulanje", "District", "plhiv")
phalombe_plhiv <- plot_malawi("Phalombe", "District", "plhiv")
balaka_plhiv <- plot_malawi("Balaka", "District", "plhiv")
chiradzulu_plhiv <- plot_malawi("Chiradzulu", "District", "plhiv")
blantyre_plhiv <- plot_malawi("Blantyre", "District", "plhiv")
mwanza_plhiv <- plot_malawi("Mwanza", "District", "plhiv")
thyolo_plhiv <- plot_malawi("Thyolo", "District", "plhiv")
chikwawa_plhiv <- plot_malawi("Chikwawa", "District", "plhiv")
nsanje_plhiv <- plot_malawi("Nsanje", "District", "plhiv")
neno_plhiv <- plot_malawi("Neno", "District", "plhiv")

```

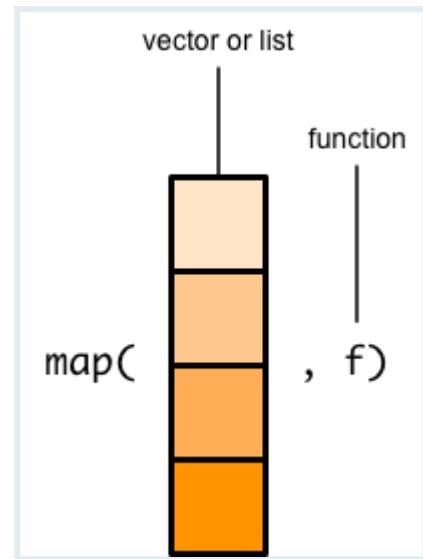
What a pain! Fortunately, R provides with a way to **iterate** our custom function through all the regions or districts, without any copying and pasting.

Looping through a vector of variables

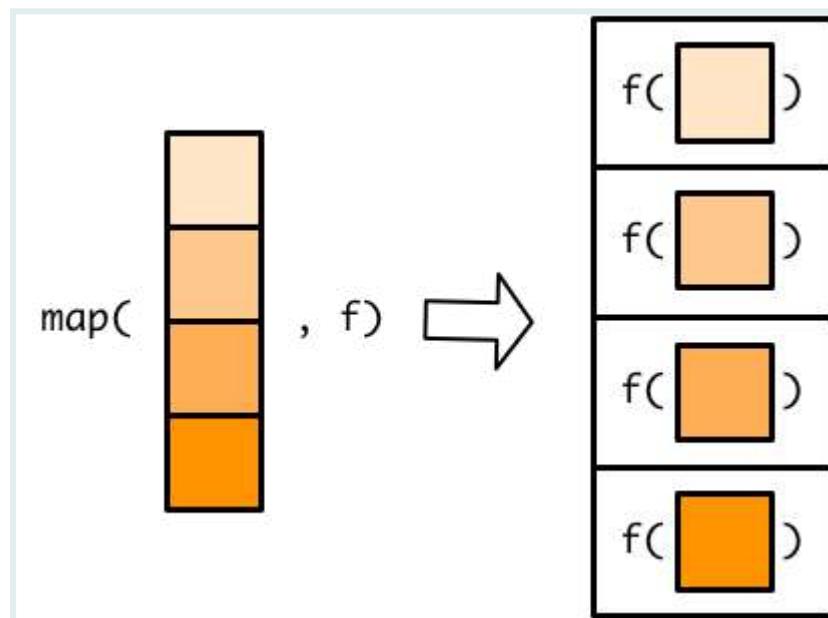
Introducing `purrr::map()`

We can make a vector of names and run the function to go through all the names in that vector with the `map()` function form the `{purrr}` package.

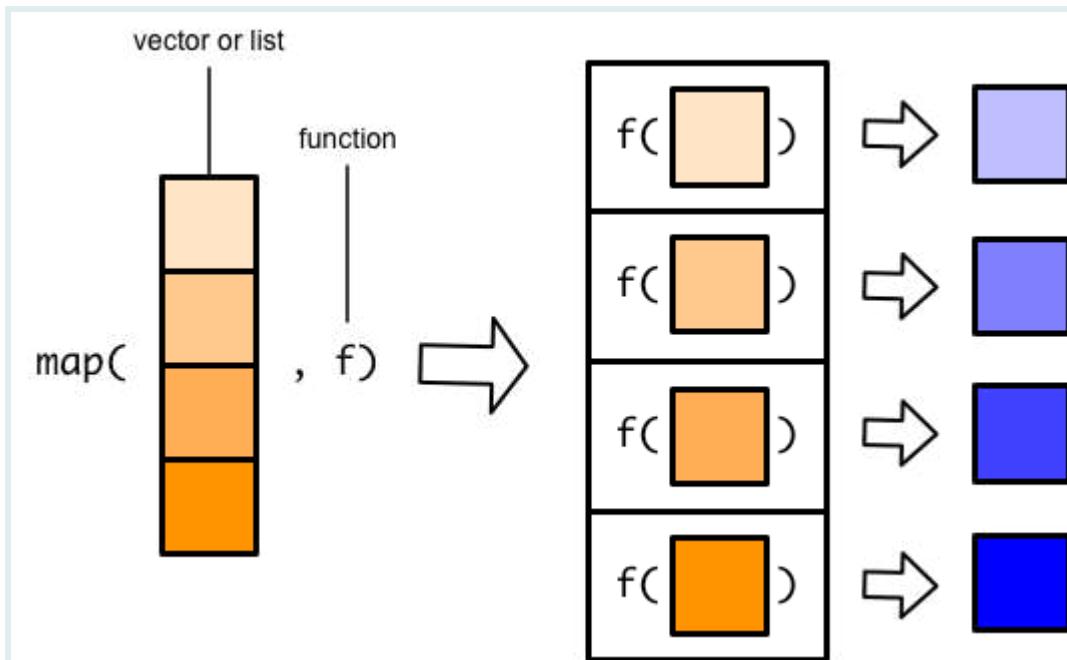
We will give `map()` two arguments: a **vector** and a **function**.



`map()` will then apply the function to each element of the input vector.



Applying the function to each element of the input vector results in one output element per each input element.



`map()` then combines all these output elements into a list.

```
list(  ,  ,  ,  )
```

For example, here we have a custom function that takes an input and appends a “Dr.” prefix to the beginning of the string.

```
# Example single argument function
add_dr <- function(full_name) {
  return(paste("Dr.", full_name))
}
# Apply the function to a single name
add_dr("Mohamed Hsin Bennour")
```

```
## [1] "Dr. Mohamed Hsin Bennour"
```

Now let's say we have a vector of names for which we want to add the prefix “Dr.”

```
# List of people
phd_students <- c("Mohamed Hsin Bennour", "Imad El Badisy",
"Kenechukwu David Nwosu")
```

We pass the vector of names to `purrr::map()`, and insert our custom `add_dr()` function as an argument. This will allow us to apply the custom function to all elements of the vector, iterating the process.

```
# Loop function over vector of variables
purrr::map(phd_students, add_dr)
```

```
## [[1]]
## [1] "Dr. Mohamed Hsin Bennour"
##
## [[2]]
## [1] "Dr. Imad El Badisy"
##
## [[3]]
## [1] "Dr. Kenechukwu David Nwosu"
```

Above you will notice that the output of `purrr::map()` is a list. To retrieve elements from list, we can first assign it to an object and then use the `[[` operator, like so:

```
# Pipe vector to map() and save output as a list
phd_grads <- phd_students %>% purrr::map(add_dr)

# Print list
phd_grads
```

```
## [[1]]
## [1] "Dr. Mohamed Hsin Bennour"
##
## [[2]]
## [1] "Dr. Imad El Badisy"
##
## [[3]]
## [1] "Dr. Kenechukwu David Nwosu"
```

```
# Call a specific element from the list
phd_grads[[2]]
```

```
## [1] "Dr. Imad El Badisy"
```

In essence, `map()` does the same work a `for` loop would do, but in a functional way.

Automating ggplots

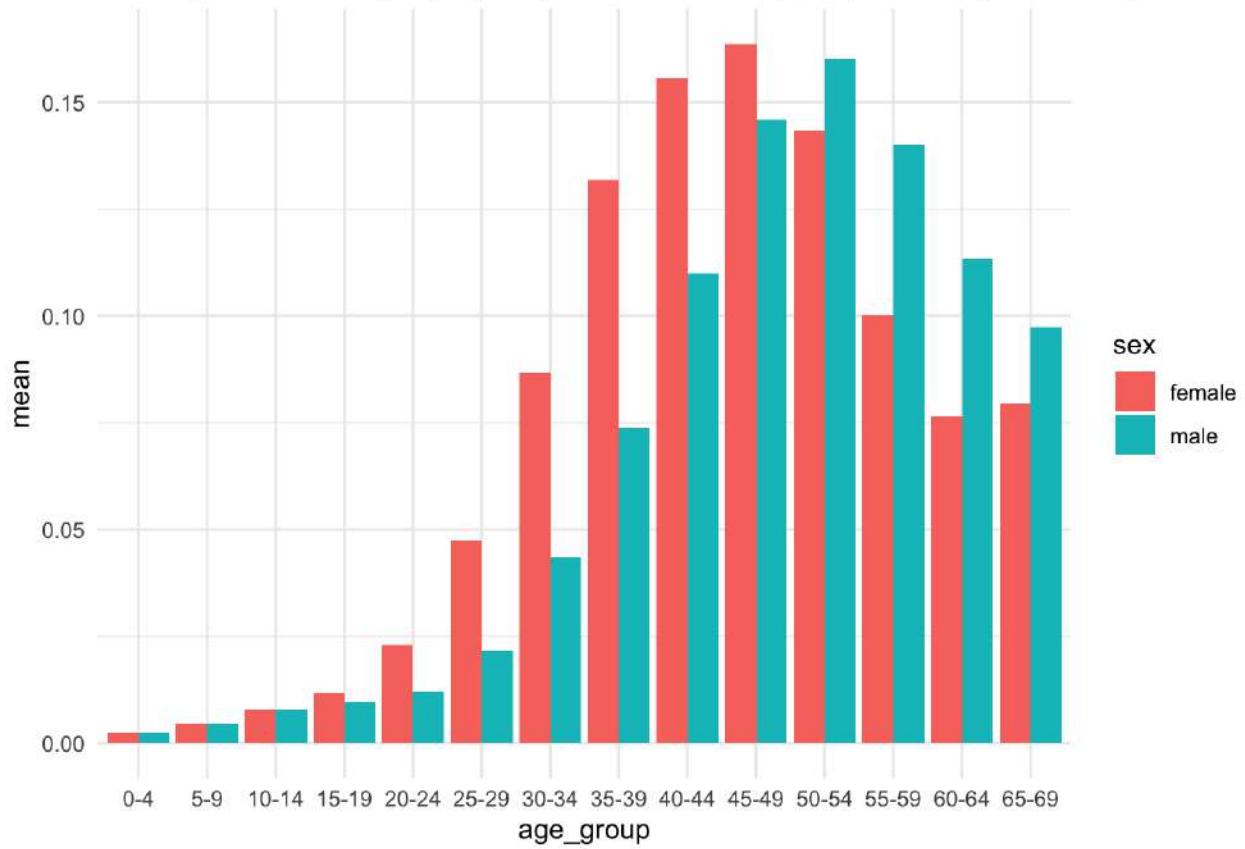
We can use the same workflow to create a list of plots, by applying our custom `plot_malawi()` function to a vector of region names.

```
# Create vector of the 3 Malawi regions
region_names <- c("Northern", "Central", "Southern")

# Apply plot_region() to region_names
region_names %>% map(plot_malawi, "Region", "prevalence")
```

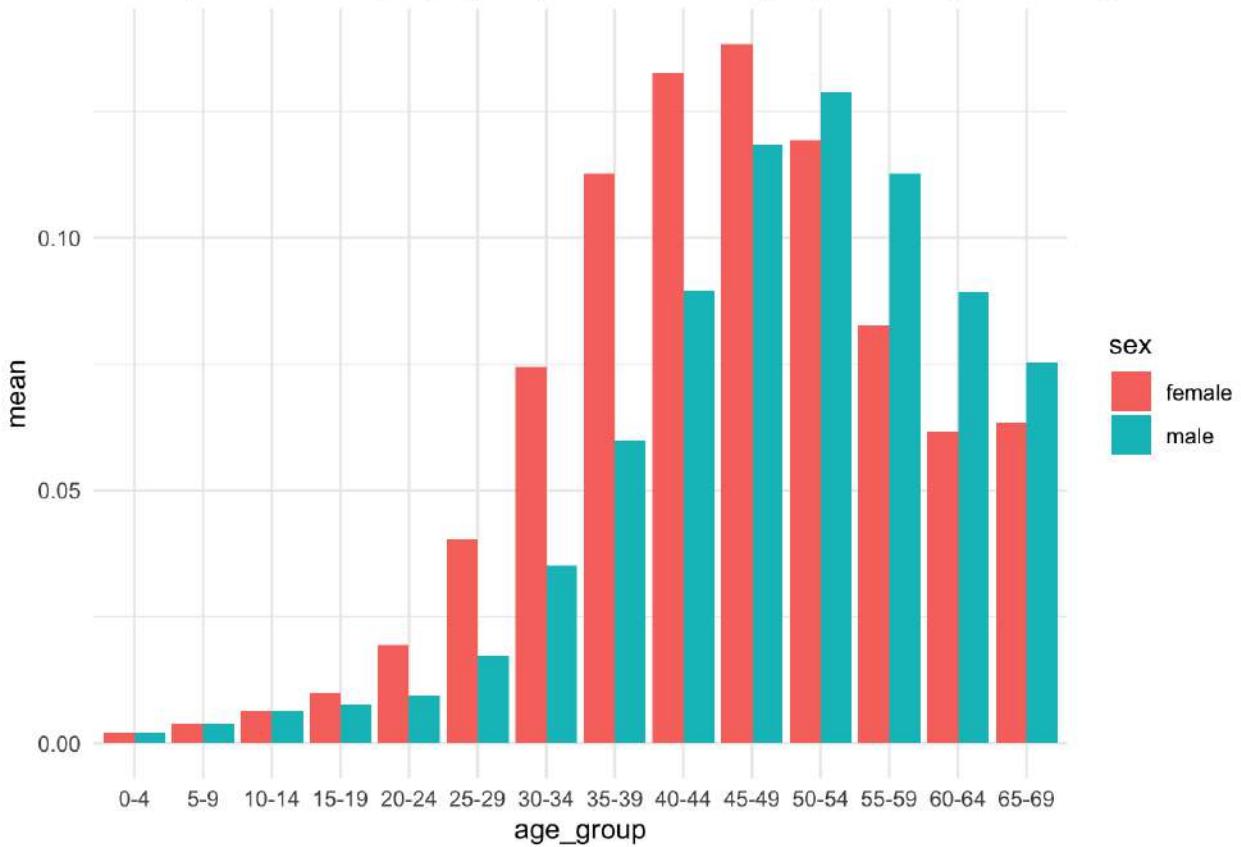
```
## [[1]]
```

Mean prevalence by age group in Northern Region, Malawi (Dec 2022)

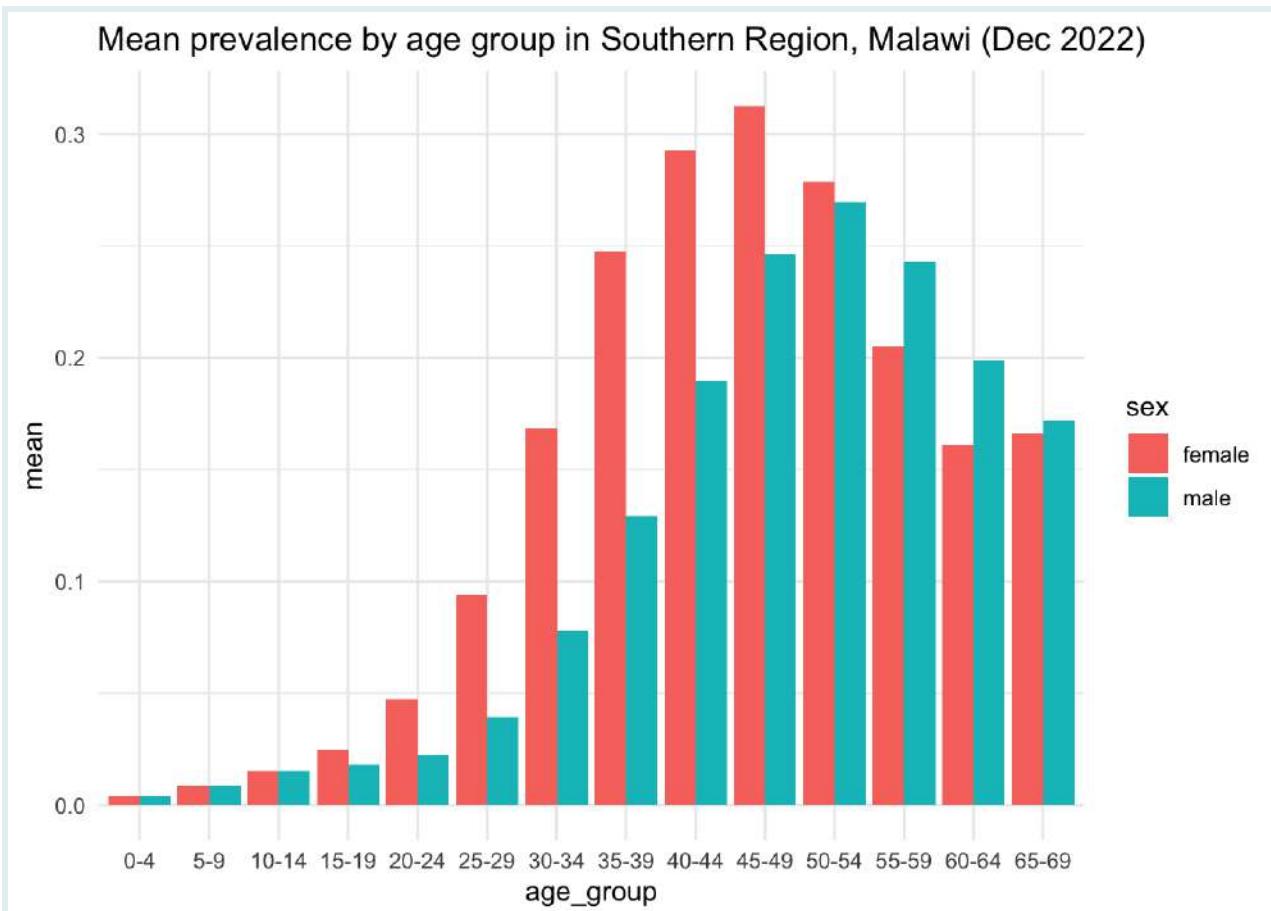


```
##  
## [[2]]
```

Mean prevalence by age group in Central Region, Malawi (Dec 2022)

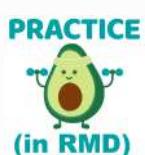


```
##  
## [[3]]
```



Now we created 3 plots with 2 lines of code.

Question 3: Iterating through a vector of districts



Create a vector of 5 district names from `hiv_malawi`.

Apply the `plot_malawi()` function to the vector of district names, to create five plots for **PLHIV** in one step.

Area level helper function

Looking at disease patterns at different geographic scales (country, state, county, city, etc.) is crucial in epidemiological analysis. We may want to make a plot for every state, or every county.

We can create a vector of all district names from `hiv_malawi` using this code pattern:

```
# Creating a vector of unique district names
district_names <- hiv_malawi %>%
  filter(area_level == "District") %>%
  pull(area_name) %>%
  unique()
# Print
district_names
```

```
## [1] "Chitipa"    "Karonga"     "Nkhatabay"   "Rumphi"      "Mzimba"
"Likoma"       "Kasungu"
## [8] "Nkhotakota" "Ntchisi"     "Dowa"        "Salima"      "Lilongwe"
"Mchinji"      "Dedza"
## [15] "Ntcheu"      "Mangochi"    "Machinga"    "Zomba"       "Mulanje"
"Phalombe"     "Balaka"
## [22] "Chiradzulu" "Blantyre"   "Mwanza"      "Thyolo"      "Chikwawa"
"Nsanje"       "Neno"
```

This code identifies unique area names at the “District” level. However, manually repeating this for different levels is inefficient. To optimize, we introduce a helper function called `area_lvl()`:

```
# Write helper function to get unique area names for a given level
area_lvl <- function(level){
  hiv_malawi %>%
    filter(area_level == {{level}}) %>%
    pull(area_name) %>%
    unique() %>%
    return()
}

# Test helper function
area_lvl("Region")
```

```
## [1] "Northern" "Central"   "Southern"
```

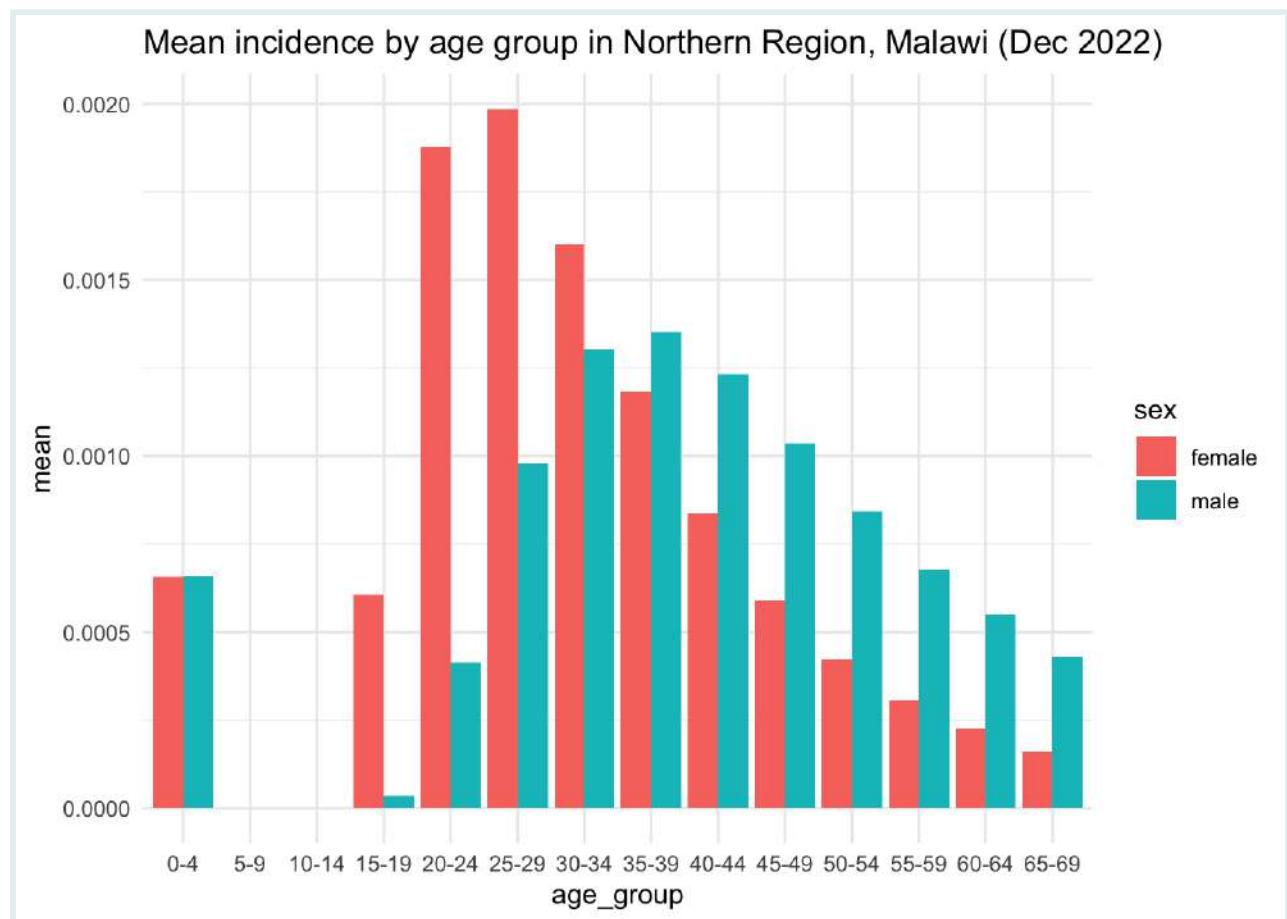
```
area_lvl("District")
```

```
## [1] "Chitipa"    "Karonga"     "Nkhatabay"   "Rumphi"      "Mzimba"
"Likoma"       "Kasungu"
## [8] "Nkhotakota" "Ntchisi"     "Dowa"        "Salima"      "Lilongwe"
"Mchinji"      "Dedza"
## [15] "Ntcheu"      "Mangochi"    "Machinga"    "Zomba"       "Mulanje"
"Phalombe"     "Balaka"
## [22] "Chiradzulu" "Blantyre"   "Mwanza"      "Thyolo"      "Chikwawa"
"Nsanje"       "Neno"
```

without having to create a custom vector first.

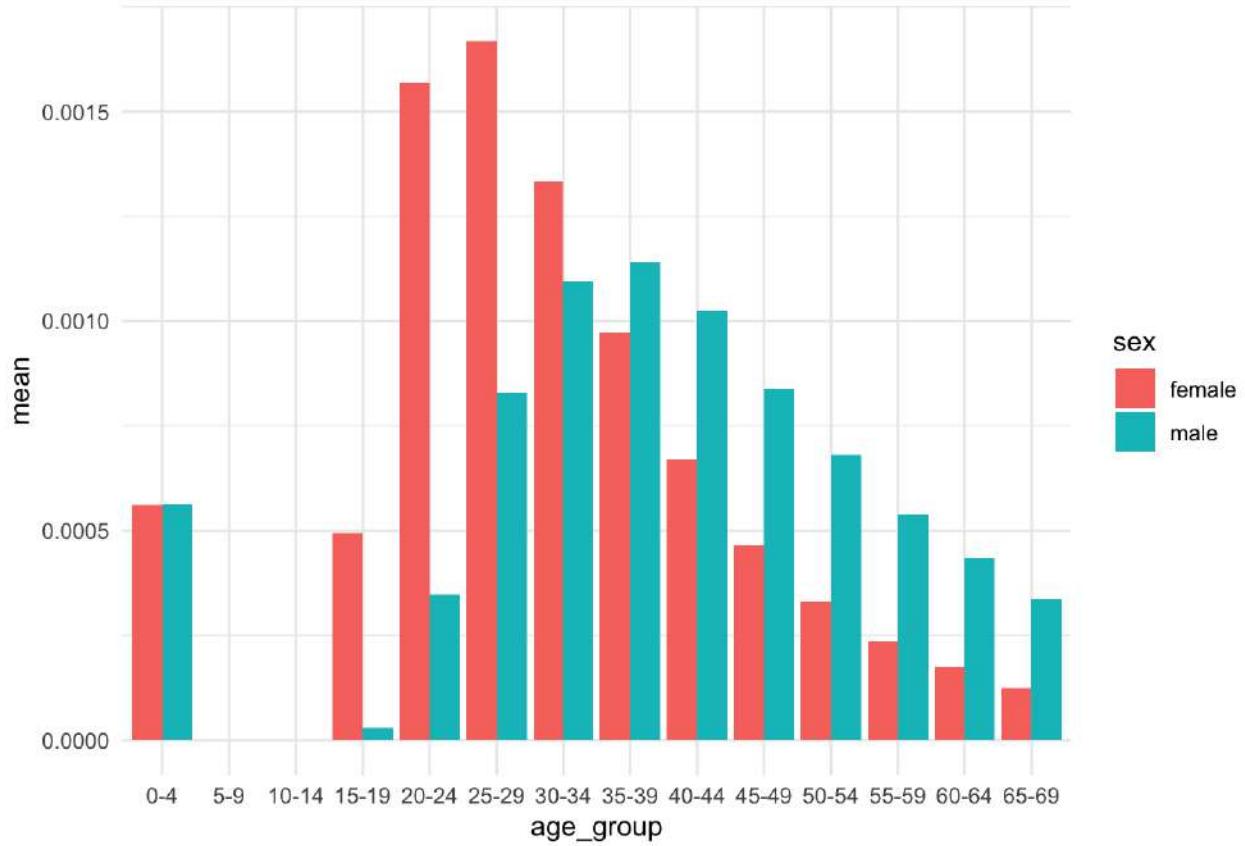
```
# Plot incidence for all regions  
area_lvl("Region") %>% map(plot_malawi, "Region", "incidence")
```

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



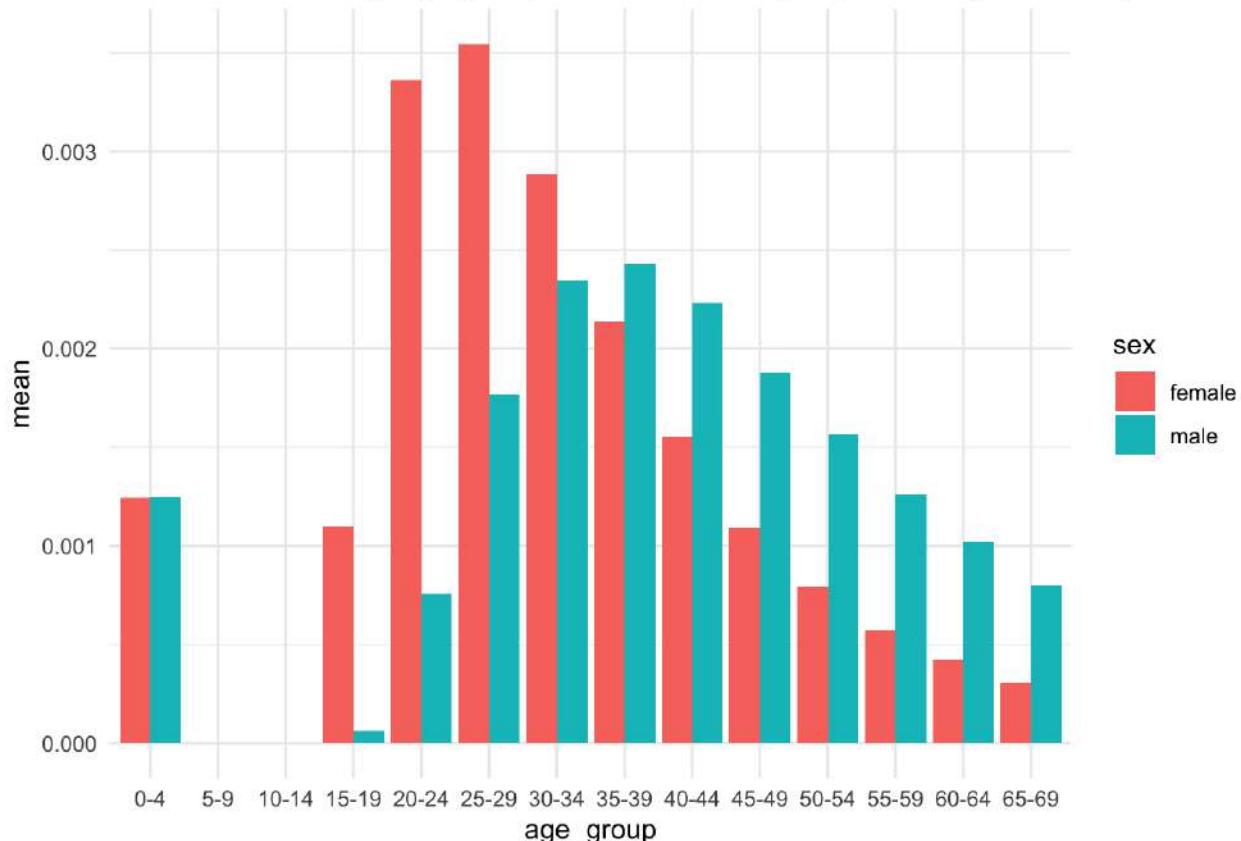
```
##  
## [[2]]
```

Mean incidence by age group in Central Region, Malawi (Dec 2022)



```
##  
## [3]
```

Mean incidence by age group in Southern Region, Malawi (Dec 2022)



```
# Plot HIV awareness for all districts
area_lvl("District") %>% map(plot_malawi, "District",
"aware_plhiv_prop")
```

Looping through two vectors

For only a few response variables we could easily copy and paste the code above, changing the hard-coded y-axis variable (`indicator`) each time. This process can get burdensome if we want to do this for many indicators, though.

Though we could use a nested loop, with a second `map()` function nested inside our previous one, this method is not as easy to interpret as using a `for` loop.

Here we will create a vector of two indicators, and feed them to a `for` loop. The `for` loop will feed each indicator to `map()`, and we will end up with 6 plots.

```

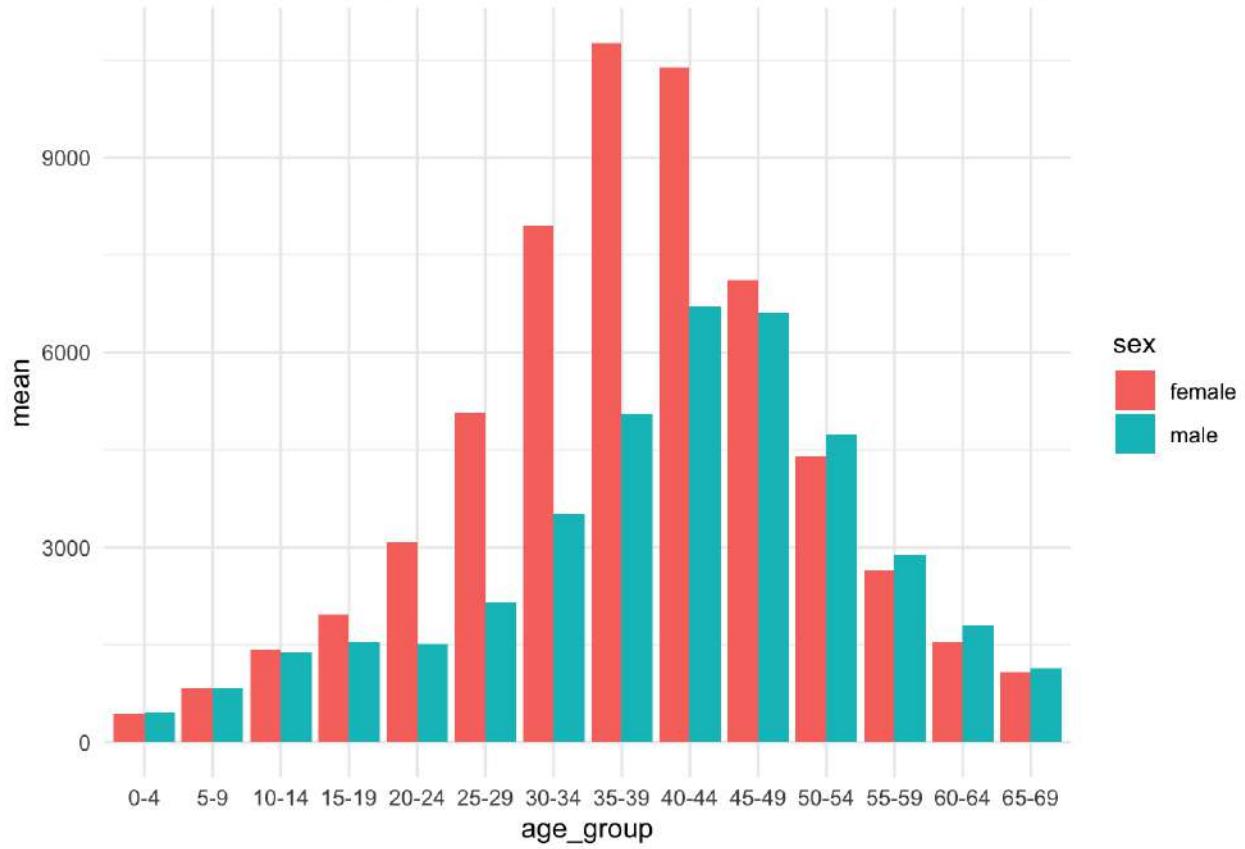
# Choose indicators: PLHIV and Prevalence
indicators <- c("plhiv", "prevalence")

# Nested loop to plot 3 regions x 2 indicators
for (i in 1:length(indicators)) {
  area_lvl("Region") %>%
    map(plot_malawi, "Region", indicators[i]) %>%
    print()
}

```

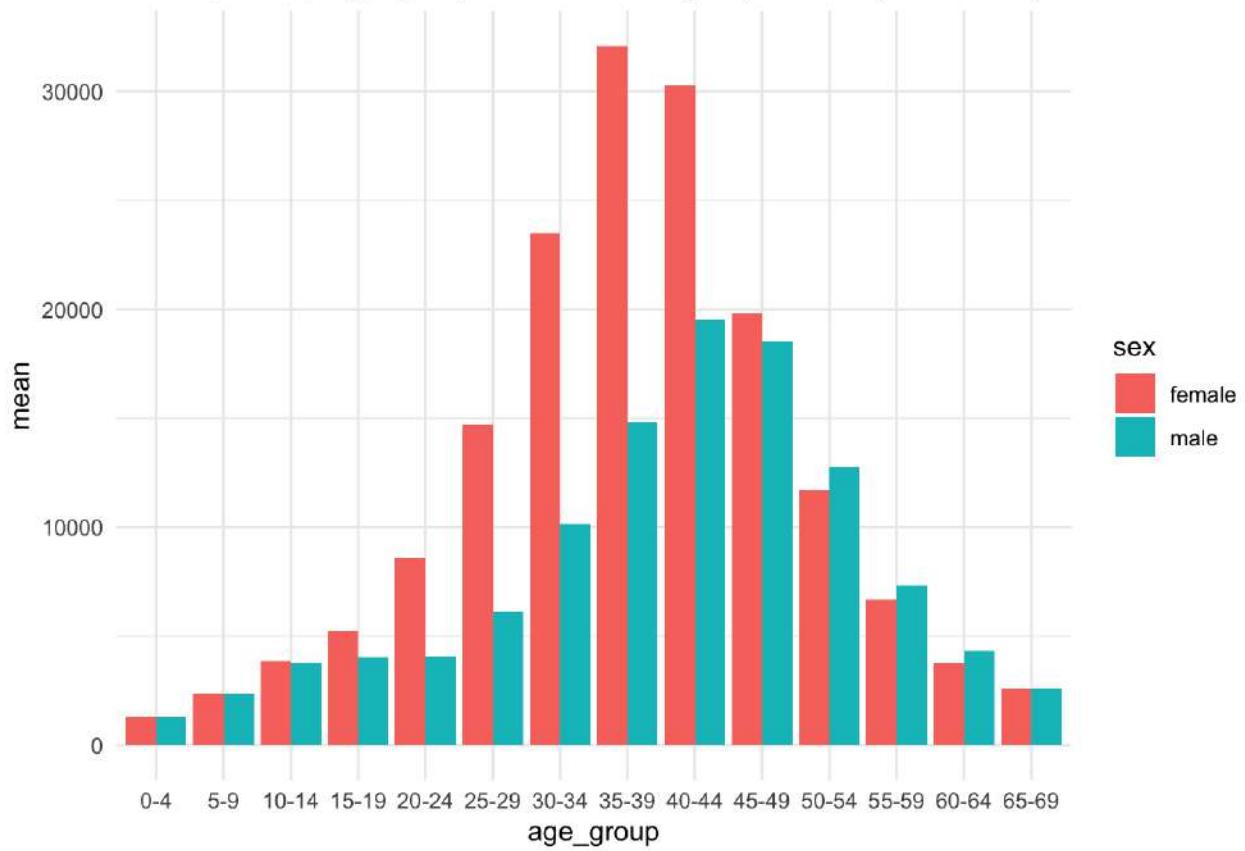
```
## [[1]]
```

Mean plhiv by age group in Northern Region, Malawi (Dec 2022)



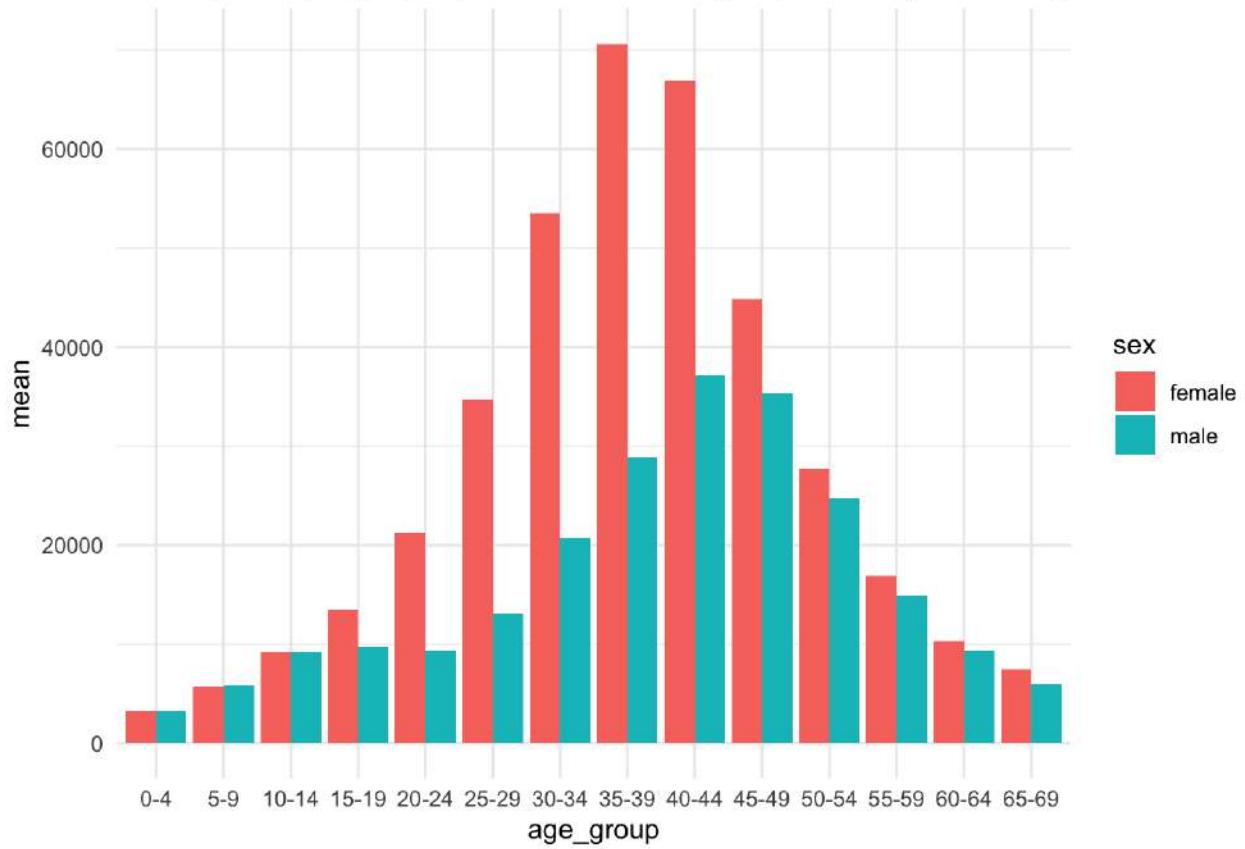
```
##  
## [[2]]
```

Mean plhiv by age group in Central Region, Malawi (Dec 2022)



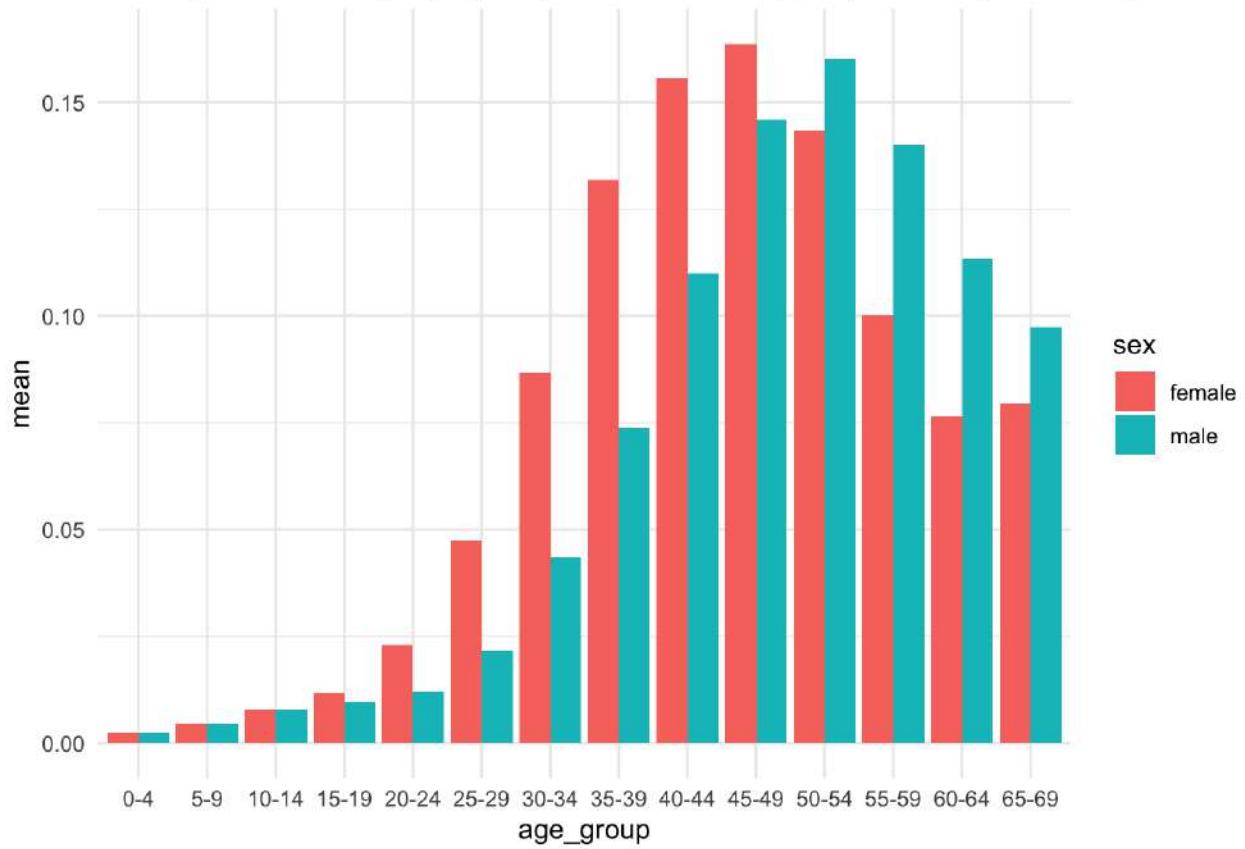
```
##  
## [[3]]
```

Mean plhiv by age group in Southern Region, Malawi (Dec 2022)



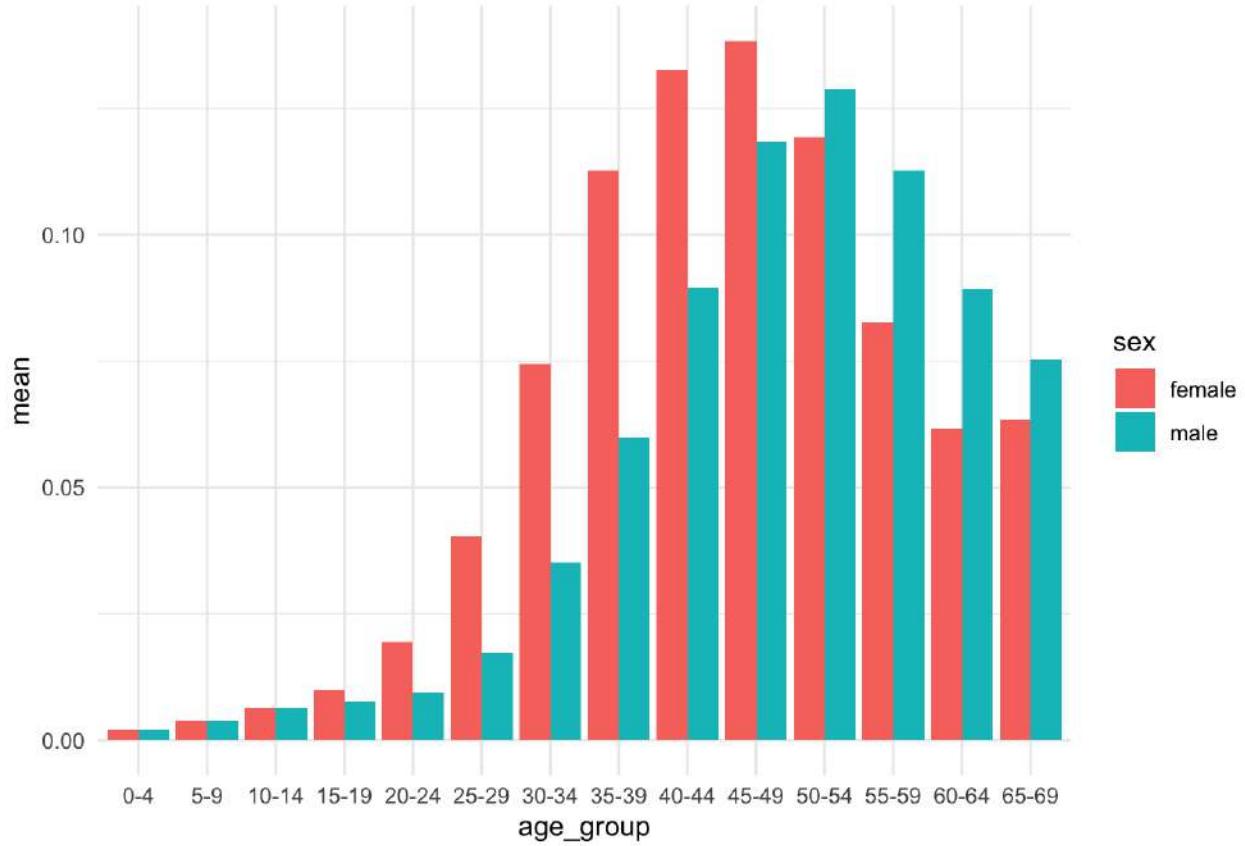
```
##  
## [[1]]
```

Mean prevalence by age group in Northern Region, Malawi (Dec 2022)

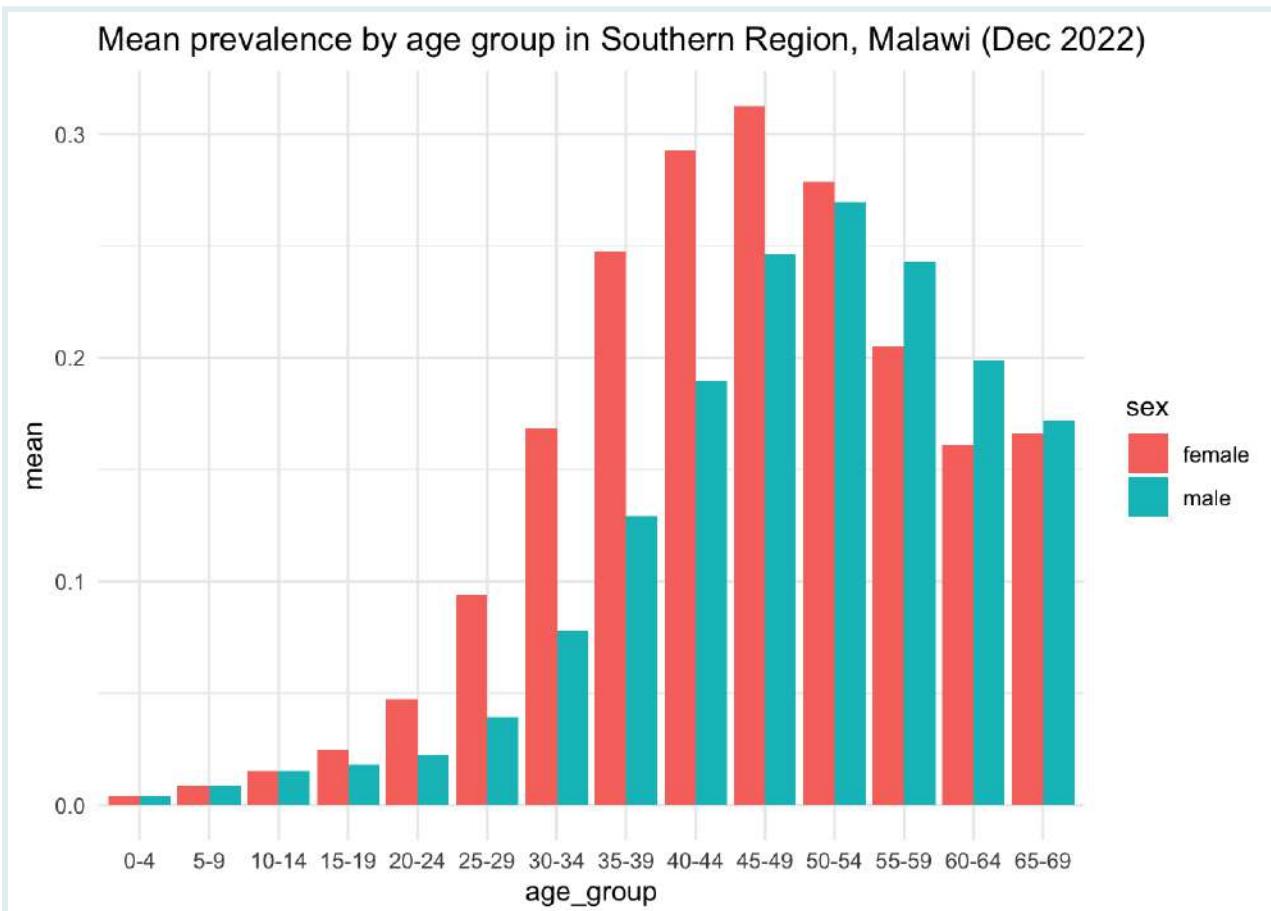


```
##  
## [2]
```

Mean prevalence by age group in Central Region, Malawi (Dec 2022)



```
##  
## [3]
```



We could change “Region” to “District”, and the above code would give us 56 plots, 2 indicators for each of the 28 districts.

Finalize and save

Now that our plotting process is fully automated, we can get ready to finalize them and save the images for further use.

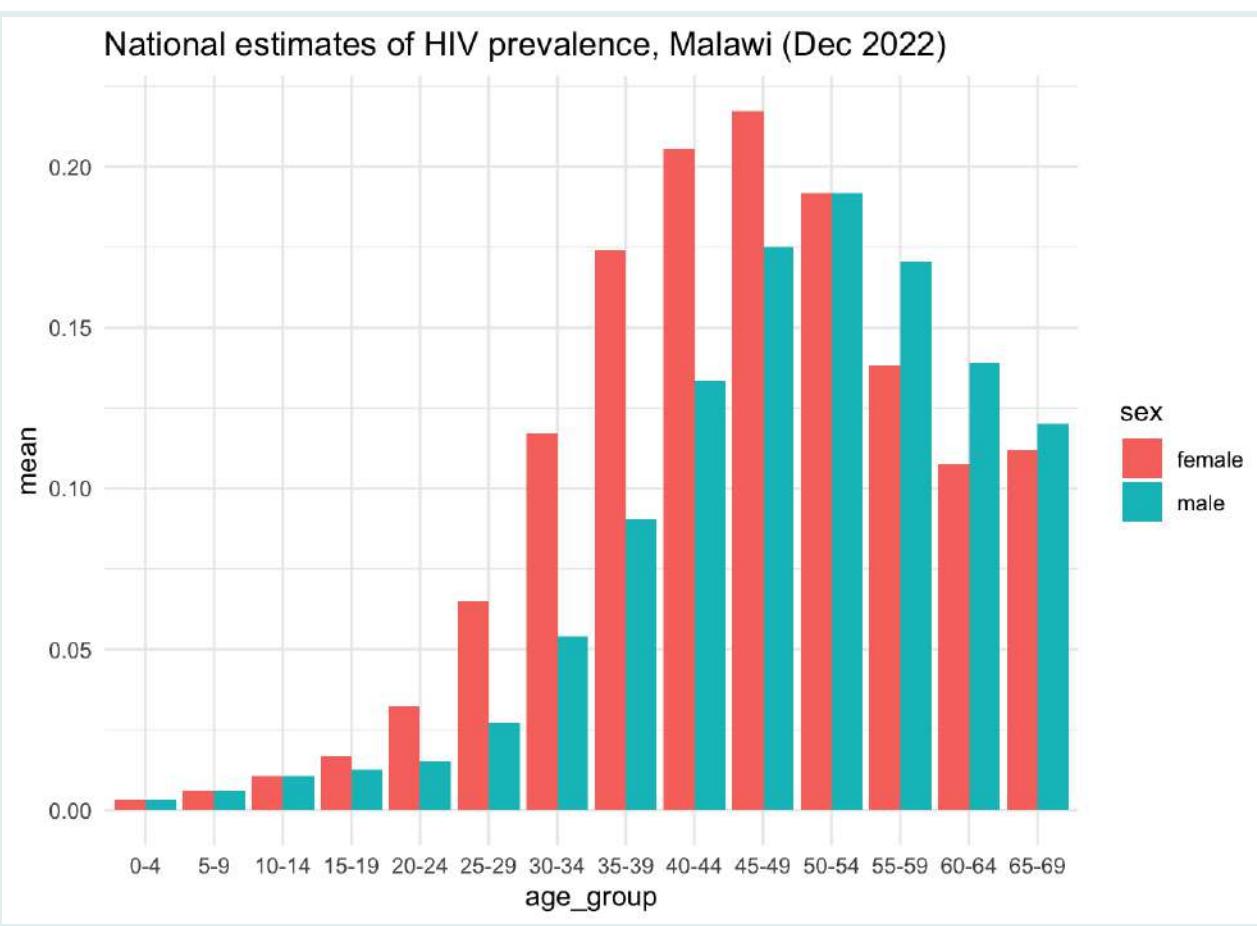
First let’s examine the plots and decide if any adjustments are needed. If changes are necessary, we don’t need to alter the code for each individual plot. Instead, we can make a small adjustment to our `plot_malawi()` function and then rerun it with the `map()` function. This is a powerful way to manage multiple plots efficiently.

Let’s revisit our original `ggplot()` code, which we’ve been filtering and reusing for all the plots:

```

# National prevalence age-sex bar chart
hiv_malawi %>%
  filter(area_level == "Country",
        indicator == "prevalence") %>%
  ggplot(aes(x = age_group,
             y = mean,
             fill = sex)) +
  geom_col(position = "dodge") +
  theme_minimal() +
  labs(title = "National estimates of HIV prevalence, Malawi (Dec
2022)")

```



This plot is not bad by any means, but there are still lots of features we can adjust and polish to make a final, publication-ready graph. Remember that `ggplot()` graphs are infinitely customizable!

The code below makes several modifications to create a final plot that looks more refined.

```

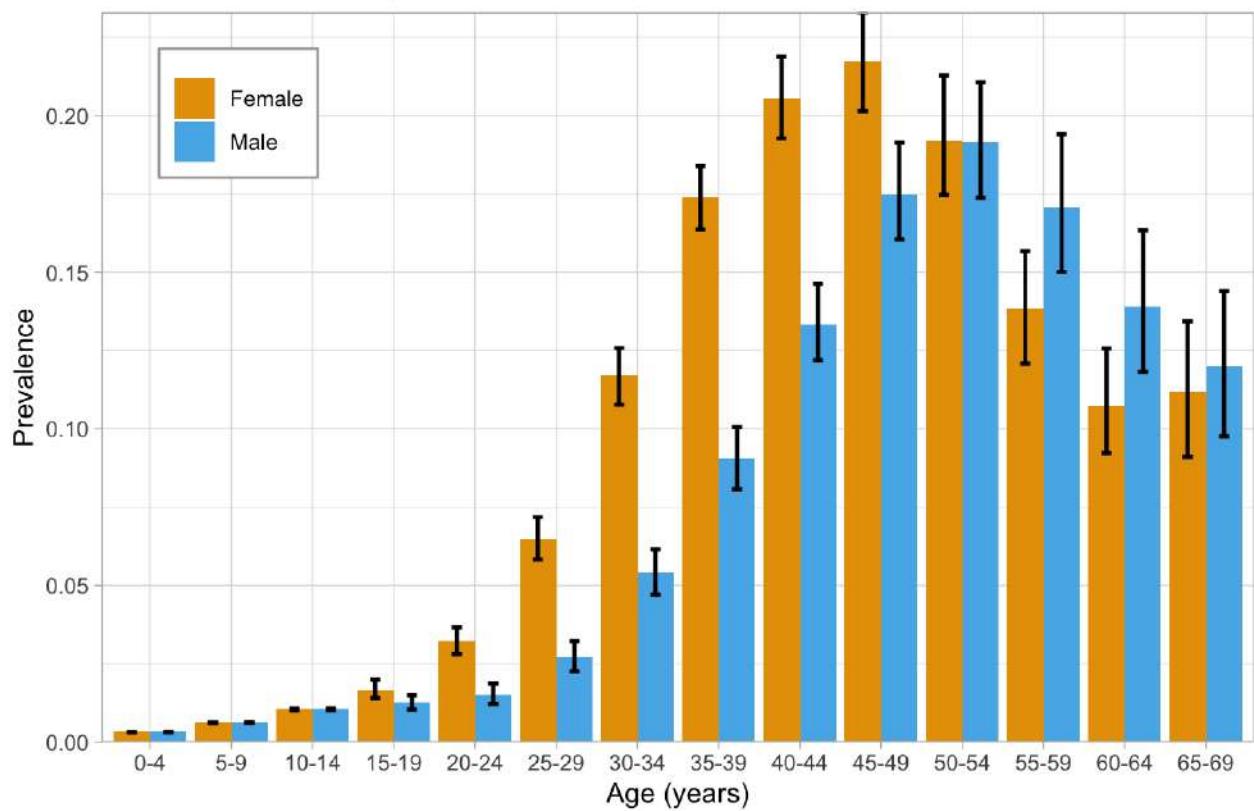
# National prevalence age-sex bar chart with additional modifications
hiv_malawi %>%
  filter(indicator == "prevalence", area_name == "Malawi") %>%
  ggplot(aes(x = age_group, y = mean, fill = sex,
             ymin = lower, ymax = upper)) +
  geom_col(position = position_dodge(width = 0.9)) +
  geom_errorbar(position = position_dodge(width = 0.9), # Add error
bars
              width = 0.25,
              linewidth = 0.8) +   # Adjust the size for thicker
lines

  scale_y_continuous(expand = c(0, 0)) + # Start y-axis at 0
  scale_fill_manual(values = c("female" = "#E69F00", "male" =
"#56B4E9"),
                     labels = c("Female", "Male")) + # Use colorblind-
friendly colors
  theme_light() + # Light theme for a cleaner look
#####
  theme(legend.position = c(0.05, 0.95), # Place legend in the top
left corner
        legend.justification = c(0, 1), # Anchor the legend at the
top left corner
        legend.background = element_rect(color = "darkgray", linewidth =
0.5),
        legend.title = element_blank()) +
  labs(x = "Age (years)",
       y = "Prevalence",
       title = "Age-Sex Distribution of HIV Prevalence in Malawi",
       subtitle = "Naomi model estimates, December 2022")

```

Age-Sex Distribution of HIV Prevalence in Malawi

Naomi model estimates, December 2022



Once we are satisfied with the look of the plot, we can use this new `ggplot()` code in a custom function:

```

# Create custom function with multiple inputs
plot_malawi_final <- function(area_name, area_level, hiv_indicator){
  hiv_malawi %>%
    # ADD PLACEHOLDERS FOR FILTERING BASED ON INPUTS
    filter(indicator == {{hiv_indicator}},
           area_level == {{area_level}},
           area_name == {{area_name}}) %>%
    ggplot(aes(x = age_group, y = mean, fill = sex,
               ymin = lower, ymax = upper)) +
    geom_col(position = position_dodge(width = 0.9)) +
    geom_errorbar(position = position_dodge(width = 0.9), ## add error
bars
               width = 0.25, linewidth = 0.8) +   # Adjust the size
for thicker lines
    ##### scale_y_continuous(expand = c(0, 0)) + # Start y-axis at 0
    scale_fill_manual(values = c("female" = "#E69F00", "male" =
"#56B4E9"),
                      labels = c("Female", "Male")) + # Use colorblind-
friendly colors
    theme_light() + # Light theme for a cleaner look
    ##### theme(legend.position = c(0.05, 0.95), # Place legend in the top
left corner
               legend.justification = c(0, 1), # Anchor the legend at the
top left corner
               legend.background = element_rect(color = "darkgray", linewidth
= 0.5),
               legend.title = element_blank()) ++
    # ADD PLACEHOLDERS FOR TITLE AND Y AXIS NAME
    labs(x = "Age (years)",
          y = hiv_indicator,
          title = glue("Age-Sex Distribution of {hiv_indicator} in
{area_name} {area_level}"),
          subtitle = "Naomi model estimates, Malawi, December 2022")
}

# Test function
plot_malawi_final("Chitipa", "District", "plhiv")

```

```

## Error in +labs(x = "Age (years)", y = hiv_indicator, title = glue("Age-Sex
Distribution of {hiv_indicator} in {area_name} {area_level}"), : invalid
argument to unary operator

```

We can use this new function to get plots for a set of regions or districts, just like we did with the previous `plot_malawi()` function.

```

# Iterate over regions
area_lvl("Region") %>% map(plot_malawi_final, "Region", "prevalence")

```

```
## Error in `map()`:  
## i In index: 1.  
## Caused by error in `+labs(x = "Age (years)", y = hiv_indicator, title =  
glue(  
##   "Age-Sex Distribution of {hiv_indicator} in {area_name}  
{area_level}"), subtitle = "Naomi model estimates, Malawi, December 2022")`:  
## ! invalid argument to unary operator
```

Now that we know the function works correctly and generates the plots we want, it's time to save them locally. This will enable us to access the plots without having to rerun the code and generate them every time.

We'll do this by making one final change to our plotting function. This time we will add the `ggsave()` function at the end, to save our plots as image files with unique and descriptive names.

```

# Create custom function to plot AND save to a specific file path
plot_save_final <- function(area_name, area_level, hiv_indicator){
  hiv_malawi %>%
    # ADD PLACEHOLDERS FOR FILTERING BASED ON INPUTS
    filter(indicator == {{hiv_indicator}},
           area_level == {{area_level}},
           area_name == {{area_name}}) %>%
    ggplot(aes(x = age_group, y = mean, fill = sex,
               ymin = lower, ymax = upper)) +
    geom_col(position = position_dodge(width = 0.9)) +
    geom_errorbar(position = position_dodge(width = 0.9), ## add error
                  bars
                  width = 0.25, linewidth = 0.8) +   # Adjust the size
    for thicker lines
    scale_y_continuous(expand = c(0, 0)) +  # Start y-axis at 0
    scale_fill_manual(values = c("female" = "#E69F00", "male" =
"##56B4E9"),
                      labels = c("Female", "Male")) + # Use colorblind-
friendly colors
    theme_light() + # Light theme for a cleaner look

    theme(legend.position = c(0.05, 0.95), # Place legend in the top
          left corner
          legend.justification = c(0, 1), # Anchor the legend at the
          top left corner
          legend.background = element_rect(color = "darkgray", linewidth =
0.5),
          legend.title = element_blank()) ++
    # ADD PLACEHOLDERS FOR TITLE AND Y AXIS NAME
    labs(x = "Age (years)",
          y = hiv_indicator,
          title = glue("Age-Sex Distribution of {hiv_indicator} in
{area_name} {area_level}"),
          subtitle = "Naomi model estimates, Malawi, December 2022") +
    # NEW CODE STARTS HERE: Save plot with custom file names
    ggsave(filename =
here(glue("outputs/{hiv_indicator}_{area_level}_{area_name}.jpg")))
}

```

Now that we have finalized our custom function called `plot_save_final()` let's try it out for the Chitipa district!

```
plot_save_final("Chitipa", "District", "plhiv")
```

You should now see a new file named “`plhiv_District_Chitipa.jpg`” in your outputs folder.

Just like before, let's now try to create a for loop for our two indicators `plhiv` and `prevalence`, but this time we will use our `plot_save_final()` function to create the plots **and** save the images in our outputs folder!

```
# Iterate over regions and SAVE
area_lvl("Region") %>% map(plot_save_final, "Region", "prevalence")
```

Next, let's loop through **two** vectors and save region-level plots for two more HIV indicators.

```
# Choose new indicators  
indicators2 <- c("plhiv", "art_coverage")  
  
# Loop through plot saving function  
for (i in 1:length(indicators2)) {  
  area_lvl("Region") %>%  
    map(plot_save_final, "Region", indicators2[i])  
}
```

If you access the outputs folder, you should now find **6** newly created and saved plots added to the folder. This is the magic of automation in R!



Question 4: Save a plot series

Using your `districts5` vector from the last question, write a for loop to create and save age-sex distribution graphs for: - Prevalence - ART coverage - PLHIV

WRAP UP!

In this lesson, we learned how to develop custom filtering and plotting functions with `{dplyr}` and `{ggplot2}`, and iterate them over vectors in two dimensions with `purrr::map()` and for loops.

In this way, we can efficiently generate customized plots and save them for future use without having to create them individually each time. This approach offers a powerful demonstration of how functional programming principles can be used to write cleaner, more modular, and easily maintainable code.

Answer Key

Contributors

The following team members contributed to this lesson:



BENNOUR HSIN

Data Science Education Officer
Data Visualization enthusiast



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Creating Parameterized Reports with {R Markdown}

Introduction
Learning Objectives
Packages
Dataset
Build a Single Report for One Country (e.g., “Angola”)
DRY Principle: Do Not Repeat Yourself!
Building a Parametrized Report
The Whole Game
WRAP UP!

Introduction

Parametrizing reports represents a significant task in epidemiological reporting, offering the flexibility to dynamically generate content based on specific parameters. This technique is invaluable, particularly in scenarios requiring report generation across varying factors, such as different countries, time periods, or disease incidences. The essence of parameterization lies in its ability to utilize a single, versatile template to produce a multitude of reports.

This not only ensures consistency and accuracy but also significantly enhances the efficiency of epidemiological data communication. With parameterized reports, complex data can be presented in a more accessible and interpretable format, aiding in the effective dissemination of critical health information.

Learning Objectives

- 1. Understand the Importance of Parameterization in R Markdown:** Grasp the fundamental concept of parameterization, its significance in epidemiological reporting, and how it transforms the way data is presented and analyzed.
- 2. Learn to Create Dynamic Reports Based on Specified Parameters:** Develop the skill to craft reports in R Markdown that automatically adjust their content based on given parameters, such as geographical regions or time frames.
- 3. Develop Skills in Creating Functions for Report Parameterization:** Acquire proficiency in writing functions in R that enable the parameterization of reports, thereby simplifying the process of report generation.
- 4. Explore Functional Programming with `map()` and `pwalk()`:** Delve into the functional programming aspects of R, specifically focusing on the usage of `map()` and `pwalk()` functions from the `{purrr}` package. Understand how these functions can be utilized to efficiently handle multiple sets of data inputs for report generation.

By achieving these objectives, you will be well-equipped to handle epidemiological datasets and present them in an informative, organized, and impactful manner. The ability to parameterize reports not only streamlines the process of data analysis but also greatly enhances the versatility and applicability of the findings presented.

Packages

This code snippet below shows how to load necessary packages using `p_load()` from the `{pacman}` package. If not installed, it installs the package before loading.

```
pacman::p_load(readr, ggplot2, dplyr, knitr, kableExtra, purrr)
```

```
## Error in download.file(url, destfile, method, mode = "wb", ...) :  
##   download from  
'https://cran.rstudio.com/bin/windows/contrib/4.3/kableExtra_1.3.4.zip'  
failed
```

Dataset

```
hiv_data <- read_csv(here::here("data/clean/hiv_incidence.csv"))
```

The dataset `hiv_incidence.csv` contains three columns:

1. `country`: The name of the country.
2. `year`: The year of the record.
3. `new_cases`: The number of new HIV cases reported in that country for the given year.

With this understanding, let's proceed to create the reports as outlined earlier. We will adapt the steps to fit this specific dataset. The tutorial will demonstrate how to build reports focusing on HIV prevalence data for different countries across various years.

Build a Single Report for One Country (e.g., “Angola”)

In this section, we focus on creating a visual report for a specific country, Angola, using R programming language. Our goal is to analyze and visualize the trend in new HIV cases in Angola over a set of years.

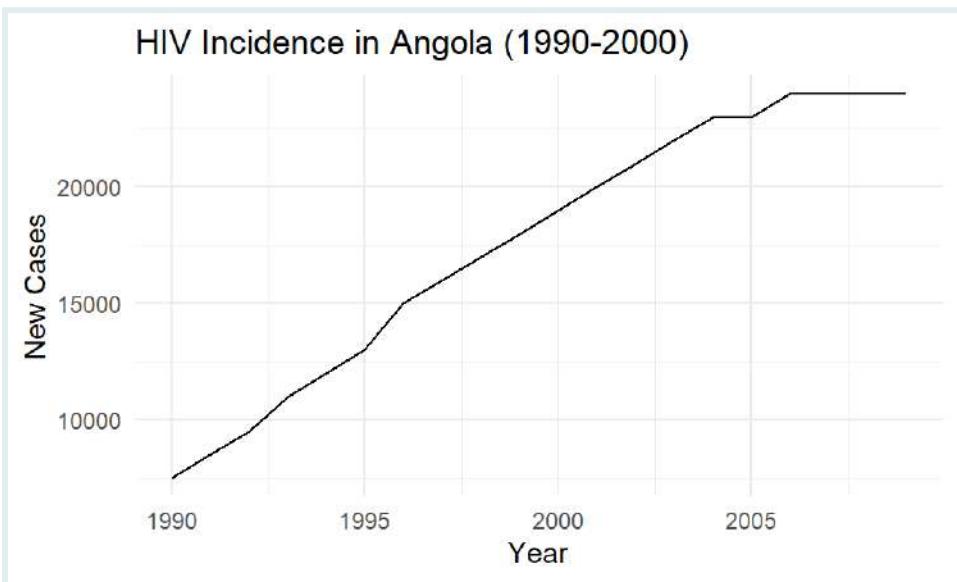
Output 1: Line Plot for “Angola”

The objective is to create a line plot that represents the trend in new HIV cases in Angola over the years. A line plot is an appropriate choice for this task as it clearly shows changes over time.

```

angola_data <- subset(hiv_data, country == "Angola")
ggplot(angola_data, aes(x = year, y = new_cases)) +
  geom_line() +
  theme_minimal() +
  labs(title = "HIV Incidence in Angola (1990-2000)",
       x = "Year",
       y = "New Cases")

```



Output 2: Statistical Table for “Angola”

This code snippet below is designed to generate a summary table for Angola, focusing on HIV cases by year. The table will highlight the most recent year's data for easier identification. Let's break down and explain each part of the code.

```

library(dplyr)
library(knitr)
library(kableExtra)

## Error: package or namespace load failed for 'kableExtra':
##   .onLoad failed in loadNamespace() for 'kableExtra', details:
##     call: !is.null(rmarkdown::metadata$output) && rmarkdown::metadata$output
##     error: 'length = 2' in coercion to 'logical(1)'

```

```

# Filter for Angola data
angola_data <- hiv_data %>% filter(country == "Angola")

# Summarize data by year
angola_summary <- angola_data %>%
  group_by(year) %>%
  summarise(Total_Cases = sum(new_cases))

# Identify the most recent year
most_recent_year <- max(angola_summary$year)

# Display the table using kable and highlight the most recent year
angola_summary %>%
  kable("html", caption = "Summary of HIV Cases in Angola by Year") %>%
  kable_styling(bootstrap_options = c("striped", "hover")) %>%
  row_spec(which(angola_summary$year == most_recent_year), background =
    "lightblue")

```

Error in row_spec(.., which(angola_summary\$year == most_recent_year), background = "lightblue"): could not find function "row_spec"



Build a simple report using R makrdown file

Now that you have two nicely prepared outputs – a line plot and a statistical table for Angola's HIV data – you can compile them into a simple, cohesive report using R Markdown. This report will not only showcase your analytical skills but also your ability to communicate findings effectively.

Steps to Follow:

1. Create a new RMarkdown file
2. Write the YAML header
3. Add a data preparation section
4. Add a line plot and statistical table
5. Knit the RMarkdown file



Build a Report for Multiple Countries (e.g., “Angola”, “Nigeria”, “Mali”)

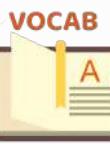
CHALLENGE

In addition to Angola, we want to create the same report for Nigeria and Mali. This challenge requires you to extend the analytical approach used for Angola to these additional countries.

Is there a more streamlined method to do this without duplicating our lines of code?

DRY Principle: Do Not Repeat Yourself!**Approach 1: Produce Output Through a Customized Function**

The function `generate_country_report()` bellow, is designed to automate the process of generating a report for a given country from a dataset. This approach follows the DRY (Don't Repeat Yourself) principle by consolidating repetitive tasks into a single function.



DRY (Don't Repeat Yourself): This is a principle of software development aimed at reducing repetition of software patterns, replacing them with abstractions or using data normalization to avoid redundancy. In essence, DRY promotes the use of functions or methods instead of repetitive blocks of code. This principle is fundamental in various programming languages and design practices.

```

generate_country_report <- function(data, country_name) {

  # Plotting the data
  country_data_plot <- subset(data, country == country_name)
  p <- ggplot(country_data_plot, aes(x = year, y = new_cases)) +
    geom_line() +
    theme_minimal() +
    labs(title = paste("HIV Incidence in", country_name, "(1990-2000)"),
          x = "Year",
          y = "New Cases")
  print(p)

  # Creating the summary table
  country_data_table <- data %>%
    filter(country == country_name) %>%
    group_by(year) %>%
    summarise(Total_Cases = sum(new_cases))

  # Identify the most recent year
  most_recent_year <- max(country_data_table$year)

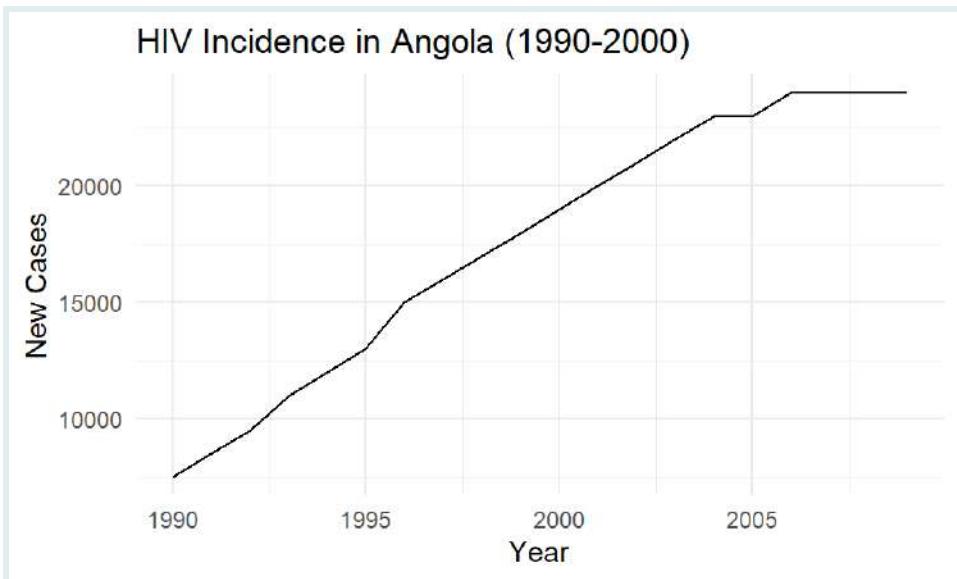
  # Display the table using kable and highlight the most recent year
  table_output <- country_data_table %>%
    kable("html", caption = paste("Summary of HIV Cases in", country_name, "by
      Year")) %>%
    kable_styling(bootstrap_options = c("striped", "hover")) %>%
    row_spec(which(country_data_table$year == most_recent_year), background =
      "lightblue")

  print(table_output)
}

# Example usage of the function
generate_country_report(hiv_data, "Angola")

```

Error in row_spec(., which(country_data_table\$year == most_recent_year), :
could not find function "row_spec"



Let's explain this code:

- 1. Function Declaration:** The function `generate_country_report()` takes two parameters: `data` (the dataset) and `country_name` (the name of the country for which the report is to be generated).
- 2. Data Plotting:** The function begins by subsetting the data for the specified country. It then creates a line plot of new HIV cases over the years using `{ggplot}`.
- 3. Summary Table Creation:** Next, it filters the data for the specified country, groups it by year, and calculates the total number of new cases for each year.
- 4. Highlighting Recent Data:** The function identifies the most recent year in the data and highlights it in the summary table.
- 5. Table Display:** The summary table is displayed using `kable()` and `kable_styling()` for a clean and interactive HTML output. The row corresponding to the most recent year is highlighted.
- 6. Printing Outputs:** Both the plot (`p`) and the table (`table_output`) are printed to the output.



Functional Programming Using `{purrr}`:

For scaling this approach to handle multiple countries efficiently, consider using functional programming paradigms, specifically with the `{purrr}` package in R. This allows for iterating over a list of



countries and applying the `generate_country_report()` function to each, streamlining the process and adhering to the DRY principle.

This function-based approach makes the code more organized and readable and also enhances its reusability and scalability, crucial aspects of efficient programming practices.

Using `map()`

Suppose you have a list of countries and you need to generate reports for each one. The `map()` function from the `{purrr}` package is an effective tool for this task, allowing you to apply the `generate_country_report()` function to each country in the list.

Here's an example of how you can use `map()` with `generate_country_report()`:

```
library(purrr)
# Vector of countries
countries <- c("Angola", "Nigeria", "Mali")

# Assuming 'hiv_data' is your dataset
# Apply 'generate_country_report' to each country
map(countries, ~generate_country_report(hiv_data, .))
```

Explanation

- `countries` is a vector containing the names of the countries for which you want to generate reports.
- `map()` takes this vector and applies the `generate_country_report()` function to each element (country name). The `~` symbol is used to define a formula where `.` represents the current element in the vector.
- For each country in `countries`, `generate_country_report(hiv_data, .)` is called, thus generating a report for that country.

This approach is advantageous when creating similar reports for multiple countries. It automates the process and efficiently applies the function to each country, ensuring consistency and saving time.

Building a Parametrized Report

YAML Header Configuration

- **Defining Parameters:** Parameters are defined in the YAML header. This is the section at the beginning of your R Markdown file where you specify various settings for your document, such as its title, output format, and parameters.

Example with an HIV Dataset

Consider an R Markdown file designed to generate a report on HIV data for different countries. You can set a parameter in the YAML header to dynamically change the country for which the report is generated.

YAML in the R Markdown file:

```
---
title: "Country HIV Report"
output: html_document
params:
  country: "Angola"
---
```

Explanation

1. Document Title and Output: In the YAML header, the document is titled “Country HIV Report”, and the output format is specified as an HTML document.

2. Parameter Definition:

- `params`: This field is used to define parameters.
- `country`: Under `params`, a parameter named `country` is declared. This parameter will control which country’s data is used in the report.
- “Angola”: The default value for the `country` parameter is set to “Angola”. This means that when the report is generated without specifying a different country, it will automatically use data for Angola.

Using the Parameter in the Document

Inside the R Markdown document, you can reference this parameter using `params$country`. For example, if you have a function like `generate_country_report()`, you can call it within a code chunk as `generate_country_report(data, params$country)`. This will use the value of the `country` parameter to determine which country’s data to report on.



By using parameters, your R Markdown document becomes more adaptable. If you need to generate a report for a different country, you simply change the parameter value when knitting the document, rather than altering the code itself. This makes your document more efficient and versatile, especially when dealing with reports that require frequent updates or modifications based on different data subsets.

Implementing Parameters in the Analysis

- **Filtering Data Based on Parameters:** Use the parameters to dynamically filter the TB dataset. For example, if the report is for a specific country and year, you can filter the data accordingly using `dplyr`.

Example code snippet

```
filtered_data <- tb_data %>%
  filter(country == params$country, year == params$year)
```

Explanation of Parameter Mapping

- **Accessing Parameters:** Parameters defined in the YAML header of your R Markdown document can be accessed in the code using `params$parameter_name`. This is similar to how you access columns in a dataframe using `dataframe$column_name`.
- **Dynamic Reporting:** When you create visualizations or summaries, you use `params$country` and `params$year` to ensure that the output reflects the specific country and year set as parameters. This means the content of the report will adapt based on the values provided for these parameters at the time of knitting the document.

SIDE NOTE



In the context of R Markdown with parameters, `params$parameter_name` accesses the value of a parameter named `parameter_name` that was defined in the YAML header. This enables dynamic and interactive report generation based on these parameter values.

Overall, the `$` operator is a key part of R syntax that facilitates manipulating elements in R objects.

Knitting with Different Parameters

- **Knitting from RStudio:** Explain how to knit the document with different parameter values directly from RStudio's knit button, where you can specify the parameter values in a dialog box.
- **Command Line Knitting:** For automated report generation, you can use R commands to knit the document with different parameters. This is particularly useful for batch processing or automated report generation.
 - Example command:

```
rmarkdown::render("TB_Report.Rmd", params = list(country = "Angola"))
```

Knitting with Different Parameters

Creating dynamic and flexible reports in R Markdown often involves knitting the document with varying parameters. There are two main ways to do this: directly through RStudio's interface and via the command line for automation.

Knitting from RStudio

- **Using RStudio's Knit Button:** When you knit an R Markdown document in RStudio that contains parameters, a dialog box will appear. This dialog allows you to specify values for each parameter before knitting.
- **Interactive Parameter Selection:** This method is user-friendly and interactive, ideal for ad-hoc report generation where you can manually enter or change parameter values each time you knit the document.
- **Customizing Parameters:** In the dialog box, you can customize the parameters according to your needs. For instance, if you have a parameter for country, you can change its value to generate a report for a different country.

Command Line Knitting

- **Automated Report Generation:** For scenarios where you need to automate the report generation process, such as in batch processing or scheduled reports, you can knit documents using R commands.
- **Example Command:**

```
rmarkdown::render("TB_Report.Rmd", params = list(country = "Angola"))
```

This command uses the `render` function from the `rmarkdown` package. It specifies the file to be rendered (`"TB_Report.Rmd"`) and sets the `params` argument to a list of desired parameter values. In this example, the `country` parameter is set to “Angola”.

- **Flexibility and Scripting:** Command line knitting allows for more flexibility and can be integrated into scripts for automated workflows. This is particularly useful when dealing with large datasets or needing to generate multiple reports periodically.

Both methods provide the means to leverage the power of parametrized reports in R Markdown, catering to different needs — from interactive, user-driven report generation to automated, script-driven processes. This flexibility is one of the key strengths of using R Markdown for data analysis and reporting.

The Whole Game

Creating a parametrized report in R involves several key steps, from setting up your R project to automating the report generation process. Let's go through these steps in detail:

Step 0: Create an R Project

- **.Rproj File:** Start by creating an R project with a `.Rproj` file. This file helps manage paths and settings, making it easier to work with files and data within the project.

Step 1: YAML Header

Here's an example of the YAML header for your R Markdown document:

```
---
```

```
title: "Report - `r stringr::str_to_title(params$state)`"
date: "`r format(Sys.time(), '%B %d, %Y')`"
output: html_document
editor_options:
  chunk_output_type: console
params:
  state: "Alabama"
---
```

This header sets up the document title, date, output format, and defines a parameter `state` with a default value of “Alabama”.

Step 2: Set-Up Your Document Environment

In the initial code chunk, set up your environment by loading required packages and your dataset:

```
# Set-up global options
knitr::opts_chunk$set(warning = FALSE, message = FALSE, echo = FALSE)

# Load packages
pacman::p_load(ggplot2, dplyr, knitr, kableExtra)

# Load the dataset
hiv_data <- read.csv("hiv_incidence.csv")
```

Step 3: The Core of Your Report

Now, add sections with outputs, dynamically generated based on the selected state.

HIV Incidence Trend

```
hiv_incidence_data <- subset(hiv_data, country == params$country)
ggplot(hiv_incidence_data, aes(x = year, y = new_cases)) +
  geom_line() +
  theme_minimal() +
  labs(title = paste("HIV Incidence in", params$country, "(1990-2000)"),
       x = "Year",
       y = "New Cases")
```

HIV Cases Summary Table

```
## HIV Cases Summary Table

# Filter for selected country data
country_data <- hiv_data %>% filter(country == params$country)

# Summarize data by year
country_summary <- country_data %>%
  group_by(year) %>%
  summarise(Total_Cases = sum(new_cases))

# Identify the most recent year
most_recent_year <- max(country_summary$year)

# Display the table using kable and highlight the most recent year
country_summary %>%
  kable("html", caption = paste("Summary of HIV Cases in", params$country,
    "by Year")) %>%
  kable_styling(bootstrap_options = c("striped", "hover")) %>%
  row_spec(which(country_summary$year == most_recent_year), background =
    "lightblue")
```

Step 4: Knitting the Report

The document can now be knitted to generate a report for the default or specified state.

CHALLENGE



Create a new R Markdown with these components and knit it to see the output.

Step 5: Create an R Script to Parameterize the Whole Process

For automation, use an R script to parameterize the process:

```

# Load necessary packages
pacman::p_load(rmarkdown, purrr, stringr)

# Define a vector of states
states <- c("Alabama", "Alaska", "Arizona")

# Generate a tibble for reports
reports <- tibble(
  filename = str_c("state_report_", states, ".html"),
  params = map(states, ~list(state = .))
)

# Use pwalk to render each report
reports %>%
  select(output_file = filename, params) %>%
  pwalk(rmarkdown::render, input = "state_report.Rmd", output_dir = "output/")

```

This script uses `map()` and `pwalk()` to iterate over each state, rendering an individualized report for each.

The `map()` function creates a list of parameters for each state, and `pwalk()` applies the `rmarkdown::render()` function to each set of parameters, generating the reports.

With these steps, you can efficiently generate customized reports for multiple states or criteria, streamlining your data analysis and reporting workflow in R.

WRAP UP!

To wrap up, we've explored a comprehensive workflow for creating parametrized reports using R Markdown. This process allows for dynamic and efficient report generation tailored to specific criteria, such as geographic regions or time frames.

Learning outcomes

At the conclusion of this lesson, you must be able to:

- **Efficiently Manage R Projects:** Organize and manage R projects for various data analysis tasks.
- **Produce Dynamic R Markdown Documents:** Generate R Markdown documents that adapt to different data inputs.
- **Conduct Data Analysis and Visualization:** Analyze data and create visualizations, focusing on real-world datasets like HIV incidence.
- **Develop Automation Scripts for Reports:** Create scripts to automate the generation of parameter-based reports, enhancing consistency and efficiency.
- **Adopt Best Practices in R Programming:** Implement best practices for more efficient, readable, and maintainable R programming.

- **Enhance Report Creation Skills:** Improve capabilities in crafting comprehensive, informative, and visually appealing reports for diverse audiences.
- **Utilize Functional Programming Techniques:** Apply functional programming methods in R for streamlined data processing and reporting.

Happy rendering!

Contributors

The following team members contributed to this lesson:



IMAD EL BADISY

Data Science Education Officer
Deeply interested in health data



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

References

- Johnson, Paul. "R Markdown: The Definitive Guide: Yihui Xie, JJ Allaire, and Garrett Grolemund. Boca Raton, FL: Chapman & Hall/CRC Press, 2018, xxxiv+ 303 pp., \$38.95 (P), ISBN: 978-1-13-835933-8." (2020): 209-210.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.

