Lab 03 - EDA for Single Variable

Learning outcomes

In this lab you will learn and practise

- Customise your UI in Shiny
- Exploratory data analysis for a single variable
- Understanding the stats in histograms, `boxplot`, and `barplot`.
- Data frame subsetting, functions and loops

Part A: UI design

In this part, we will explore some UI design in Shiny.

Step 1: Begin by creating a Shiny app file. Go to File → New File → Shiny Web App. You'll encounter two application types: single file or multiple files. While both are functional, for this example, we'll use the single file option. Enter a name for your application, such as "week4App", and then click on the "create" button. This action will generate an "app.R" file.

Step 2: Inside the "app.R", you'll find a sample code. To visualise the application, you can simply click on "Run App" and interact with the provided sample.

Step 3: Next, let's customise the "app.R" file according to your preferences. You can begin by clearing the existing sample code and load dataset by using:

```
df <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine
```

Note that, `df` is just a variable name.

```
library(ggplot2)
library(shiny)

# load dataset
df <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine
# Define UI for application e.g.draws a histogram
ui <- fluidPage(
    titlePanel("JUST A NAME"),
    sidebarLayout(
        sidebarPanel(

        ),
        mainPanel(
            plotOutput(outputId = "show")
        )
    )
)
# Define server logic required draw a histogram
server <- function(input, output) {
    output$show <- renderPlot({
    })
}
# Run the application
shinyApp(ui = ui, server = server)
```

Step 4: Let's explore different input options within the `sidebarPanel`.

1. **Slider Input**: Begin by incorporating the provided code snippet into the `sidebarPanel`. When you run or reload the app, you will observe a slider element displayed on the web interface.

```
sliderInput(inputId = "bins", label = "Number of bins:", min=1, max=50, value=
```

2. **Select Input**: Modify your `sidebarPanel` according to the given instructions below. Then, reload the app to witness the changes.

```
sidebarPanel(
    sliderInput(inputId = "bins", label = "Number of bins:", min=1, max=50, va
    selectInput(inputId = "x1", label = "Choose X:", choices = names(df)),
    selectInput(inputId = "x2", label = "Choose a quality:", choices = c(df$qu
    selectInput(inputId = "x3", label = "Choose from quality and pH:", choices
),
```

3. **Radio Button**: Integrate the following code snippet into your `sidebarPanel` and reload the app.

```
radioButtons(inputId = "animal", label = "What's your favourite animal?", choi
```

You can explore more input options in https://mastering-shiny.org/basic-ui.html#inputs.

Step 5 (optional): Step 4 generally covers the essentials of basic design fullfillment. However, for a more personalised touch and highly customised Shiny apps, you can incorporate HTML elements into your UI. For further information on various HTML elements, you can refer to https://www.w3schools.com/html/html_elements.asp.

```
library(shiny)
df <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine
# Define UI for application that draws a histogram
ui <- fluidPage(
    h1("This is a heading"),
    p("This is some text", class = "my-class1"),
    tags$ul(
        tags$li("First bullet"),
        tags$li("Second bullet")
    ),
    # Application title
    titlePanel("JUST A NAME"),
    h2("heading 2"),
    sidebarLayout(
        sidebarPanel(
            sliderInput(inputId = "bins", label = "Number of bins:", min=1, ma
            selectInput(inputId = "x1", label = "Choose X:", choices = names(d
            selectInput(inputId = "x2", label = "Choose a quality:", choices =
            selectInput(inputId = "x3", label = "Choose from quality and pH:",
            radioButtons(inputId = "animal", label = "What's your favourite ar
            # adding the new div tag to the sidebar
            tags$div(class="header", checked=NA,
                     tags$p("Ready to take the Shiny tutorial? If so"),
                     tags$a(href="https://shiny.posit.co/r/articles/build/html
            ),
            h3("This is also a heading"),
            p("This is also some text", class = "my-class2"),
            tags$ul(
                tags$li("First bullet 2"),
                tags$li("Second bullet 2")
            ),
        ),

        mainPanel(
        plotOutput("distPlot")
        )
    )
)
# Define server logic required to draw a histogram
server <- function(input, output) {
    output$distPlot <- renderPlot({
    })
}
# Run the application
shinyApp(ui = ui, server = server)
```

Step 6: Having acquired the skills to craft a customised UI, let's shift our focus to the `mainPanel`. The function `plotOutput` plays a pivotal role in Shiny. When employing `plotOutput`, it's important to pair it with `renderPlot` in your server section. More functions are provided below.

Outputs render*() and *Output() functions work together to add R output to the UI.

DT::renderDataTable(expr, options, searchDelay, callback, escape, env, quoted, outputArgs)	dataTableOutput(outputId)
renderImage(expr, env, quoted, deleteFile, outputArgs)	imageOutput(outputId, width, height, click, dblclick, hover, brush, inline)
renderPlot(expr, width, height, res, …, alt, env, quoted, execOnResize, outputArgs)	plotOutput(outputId, width, height, click, dblclick, hover, brush, inline)
renderPrint(expr, env, quoted, width, outputArgs)	verbatimTextOutput(outputId, placeholder)
renderTable(expr, striped, hover, bordered, spacing, width, align, rownames, colnames, digits, na, …, env, quoted, outputArgs)	tableOutput(outputId)
renderText(expr, env, quoted, outputArgs, sep)	textOutput(outputId, container, inline)
renderUI(expr, env, quoted, outputArgs)	uiOutput(outputId, inline, container, …) htmlOutput(outputId, inline, container, …)

These are the core output types. See **htmlwidgets.org** for many more options.

`"distPlot"` is the outputId. Feel free to replace it with any suitable name. Remember, if you modify the outputId, the corresponding adjustment should also be made in the renderPlot function within the server section. For instance, if I've defined the id as `plotOutput("hist")`, then the server's renderPlot should align accordingly:

```
output$hist <- renderPlot({
    })
```

Part B: Server design

Let's design the server logic. To illustrate, I'll use the `geom_histogram()` function. Assuming we're working with an R dataframe named `df` and focusing on the column `fixed.acidity`, here's the code for plotting a histogram:

```
ggplot(df, aes(x = fixed.acidity)) +geom_histogram(bins = 20, color = "white")
```

To adapt this function for use within the Shiny server, follow these steps:

```
output$hist <- renderPlot({
ggplot(df, aes_string(x = input$x1)) +geom_histogram(bins = 20, color = "white
})
```

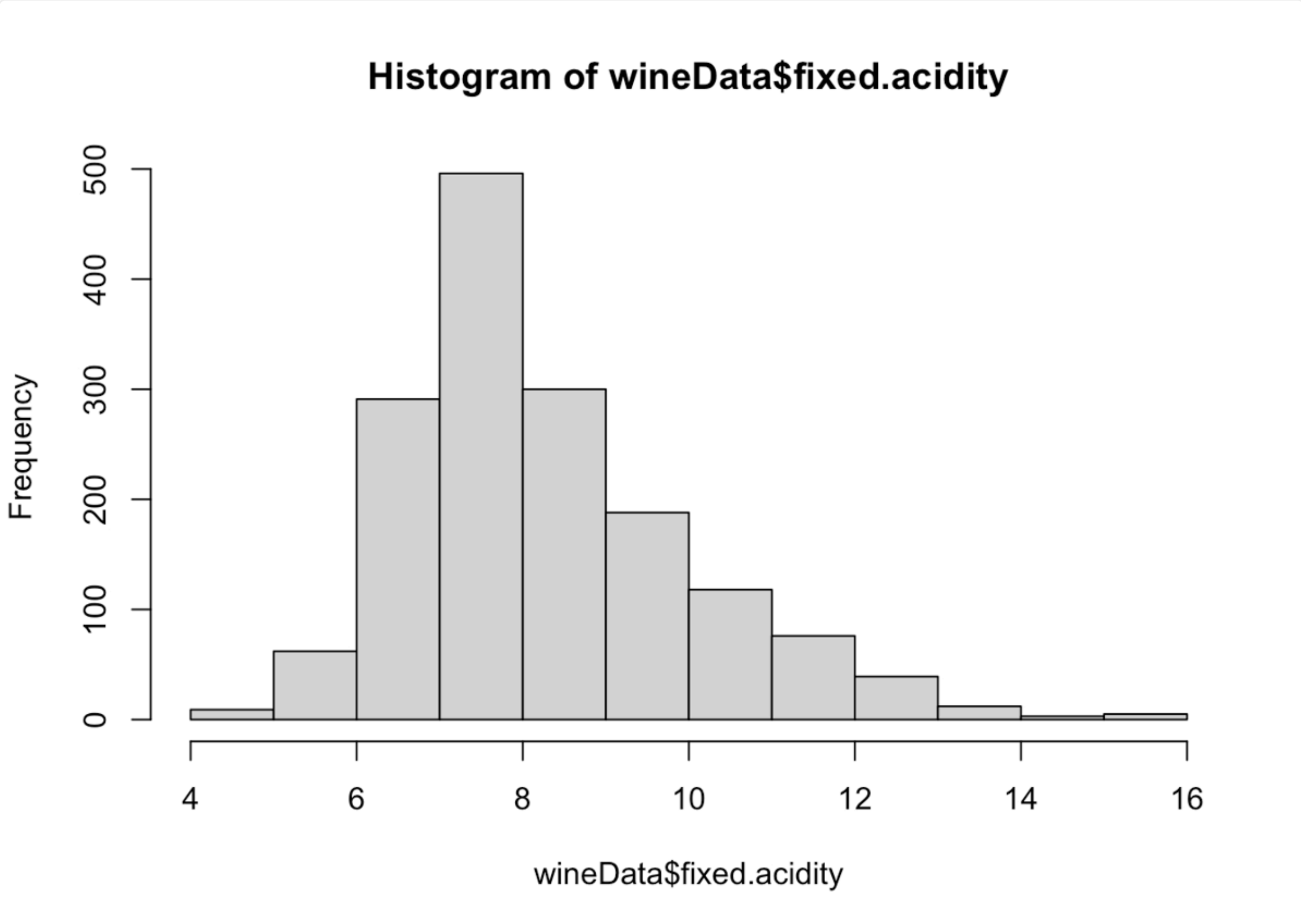
In this code:

- `input$x1` refers to the selected choice from the first `selectInput` where you've defined `inputId = "x1"`.
- We've replaced `aes` with `aes_string` to accommodate dynamic input variable names.

Part C: EDA for Single Variable

Note that you **do not need** to perform this part in Shiny. Let's continue analysing the Wine data from last week, we read the data into a data frame and did a simple histogram plot for the fixed.acidity. For example, using the following code:

```
wineData <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-database
fixedAcidity <- hist(wineData$fixed.acidity)
```



If it doesn't work, then download the `winequality-red.csv` file and read it from your local disk.

Inspect the variable `fixedAcidity` that stores the output from the `hist()` function above. In particular, look at `fixedAcidity$breaks`, `fixedAcidity$mids` and `fixedAcidity$counts`. What are the lengths of these three vectors?

Write some R code to understand how the stats are calculated in the R built-in Histogram

Excercise 1: Write an R statement that involves using `fixedAcidity$counts` and `fixedAcidity$mids` to find out how many observations falling into bin `(7,8]` in the above histogram. Here, `(` indicates *non-inclusion*, and `]` indicates *inclusion* (i.e., the number of observations whose *fixed acidity* values are greater than 7 but less than or equal to 8). Your R code should produce the following:

```
## [1] 496
```

Exercise 2: Extend Exercise 1 above as follows: Write a `for` loop which would count the numbers of observations whose fixed acidity values fall into the 12 bins `(4,5], (5,6], ..., (15,16]`. Save these numbers of observations to a vector named `counts`. Your `counts` vector should have the following values:

```
##  [1]   9  62 291 496 300 188 118  76  39  12   3   5
```

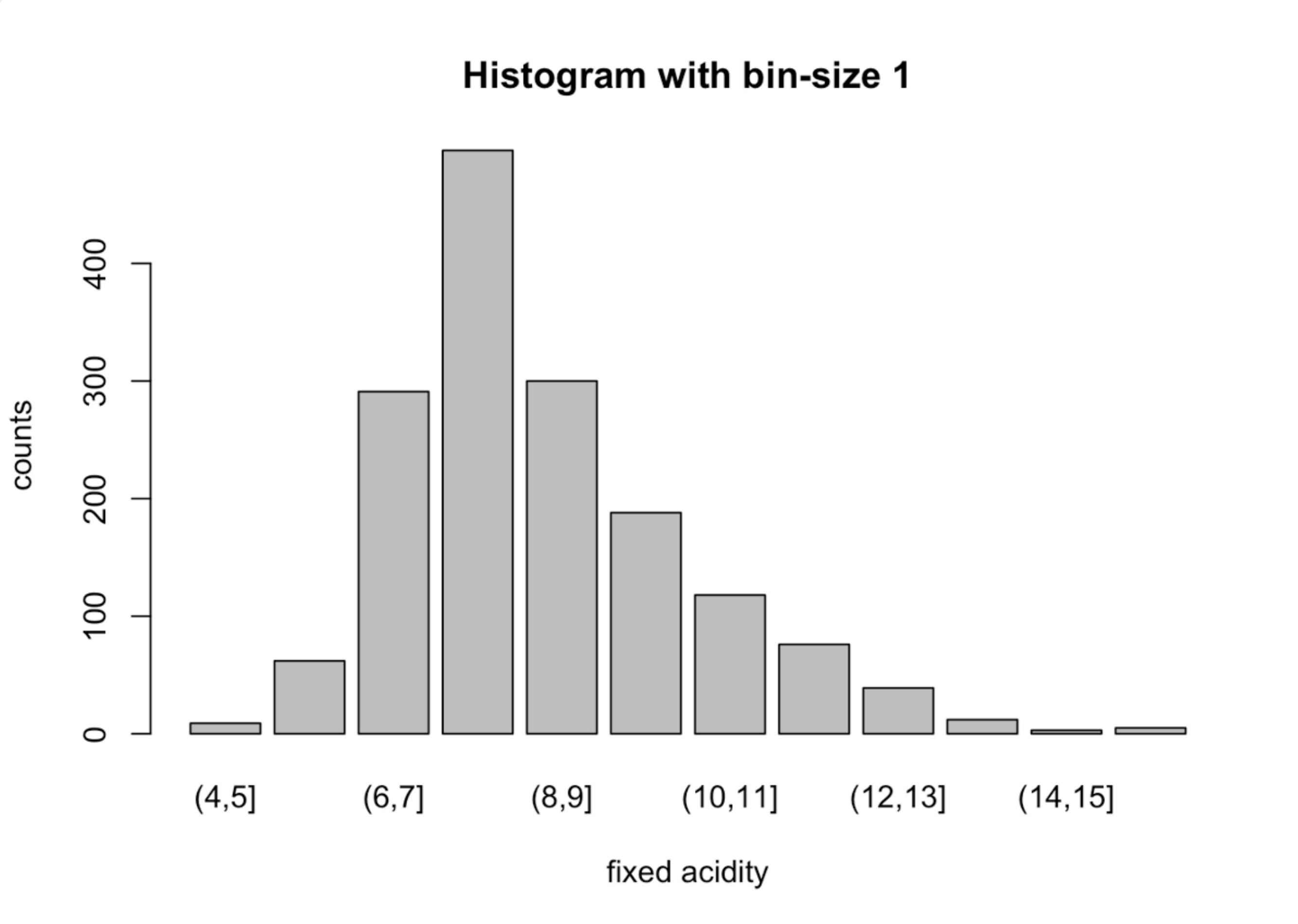
Hint: You can use the following block of code to create a zero vector of length n (in our case, we want n=12:

```
count <- c()
n <- 12
for (i in 1:n) count <- c(count, 0)
```

Alternatively and more efficiently, since the `counts` vector is going to store integer values only, we can simply do:

```
counts <- integer(12)
```

Plot your `counts` vector using the R built-in `barplot()` function. Set some appropriate parameters so that your plot has a meaningful title, axis labels and tick marks, like the diagram shown below:



Note that R automatically omits some tick marks to avoid over-crowding the axis in the plot. To get the label below each histogram bin as shown in the diagram above, try

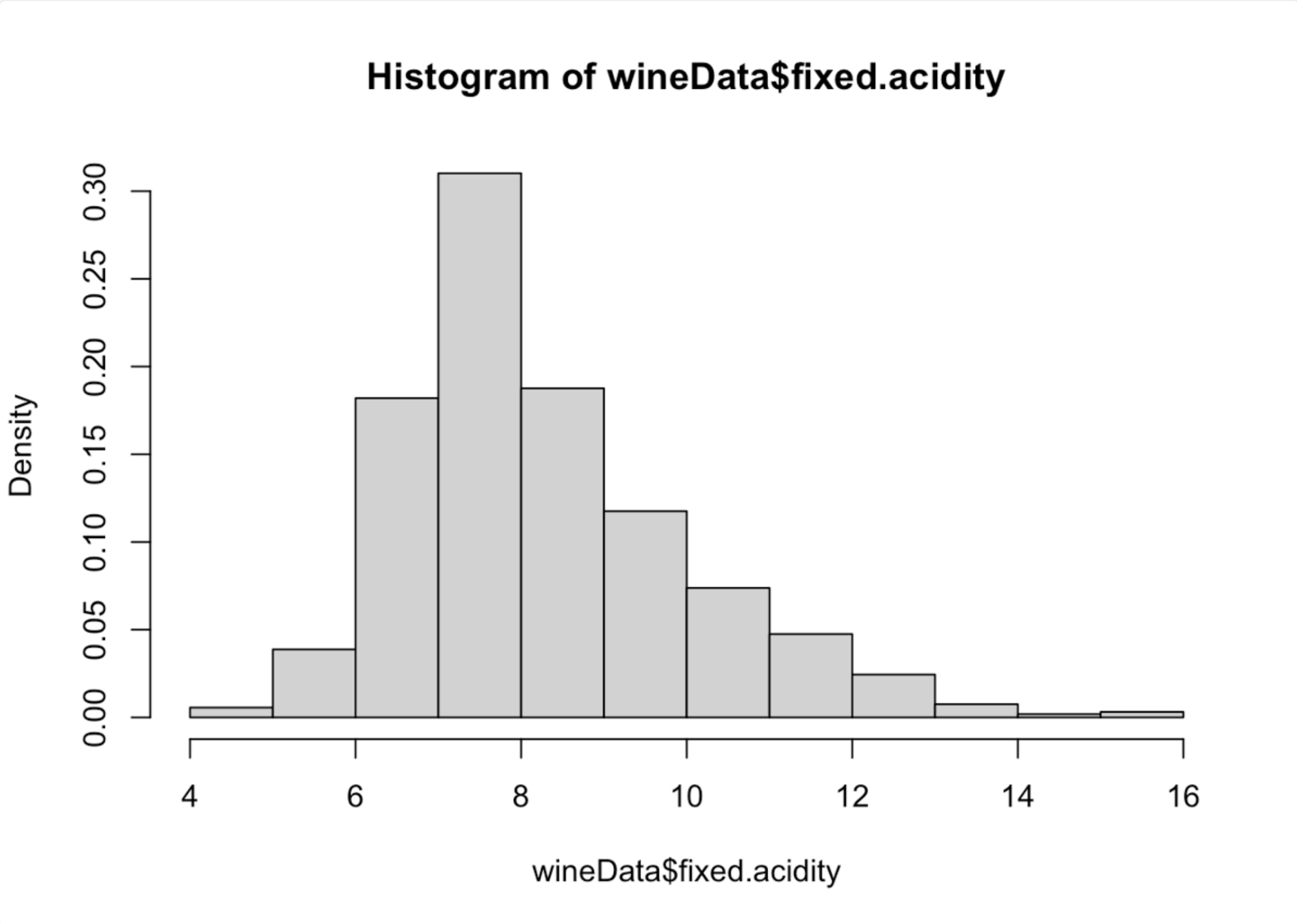
```
bins <- seq(4,15)
name_str <- paste("(", bins, ",", bins+1, "]", sep="")
```

and use the variable `name_str` appropriately when constructing your barplot.

Compare your plot with the histogram from `hist()`.

Excercise 3: Take a look at values on the vertical axis in the following R built-in Density Histogram, when `freq` is set to `FALSE`.

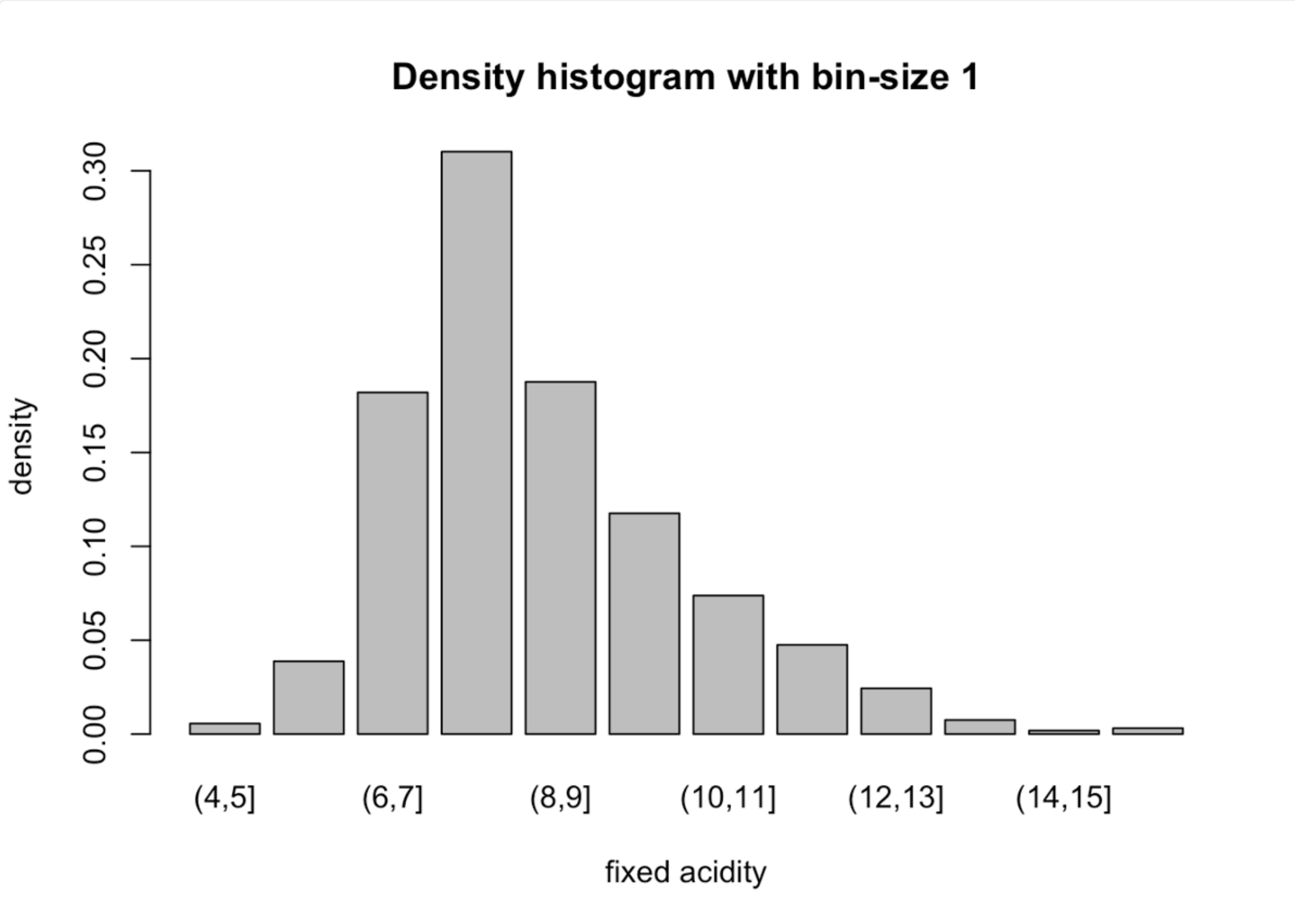
```
hist(wineData$fixed.acidity, freq=FALSE)
```



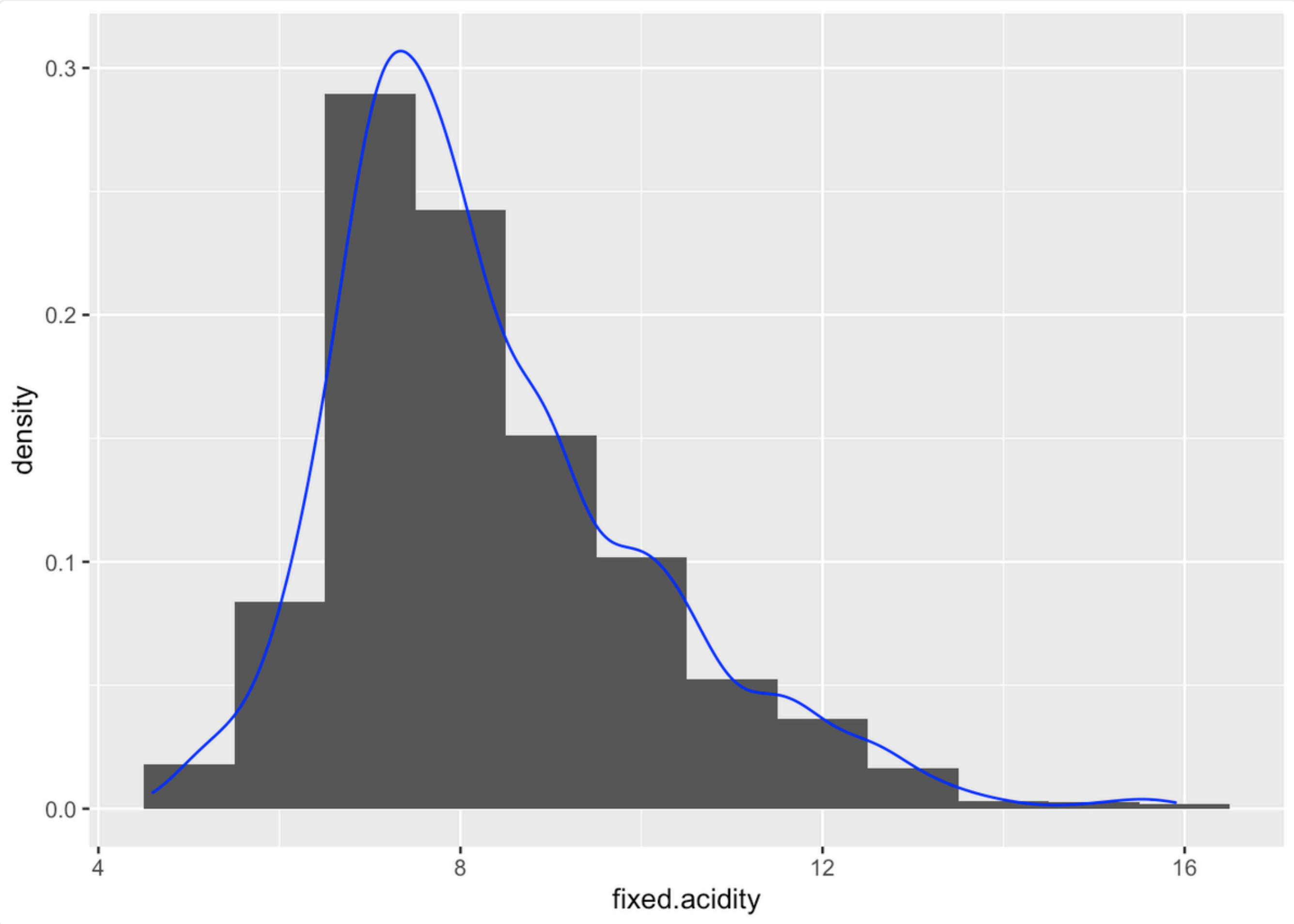
Reproduce the values along the *density* axis for each bin based on your `counts` vector that you worked out earlier:

```
freqs <- counts / sum(counts)
```

and plot these values using the built-in `barplot()` function. Your plot should look like this:

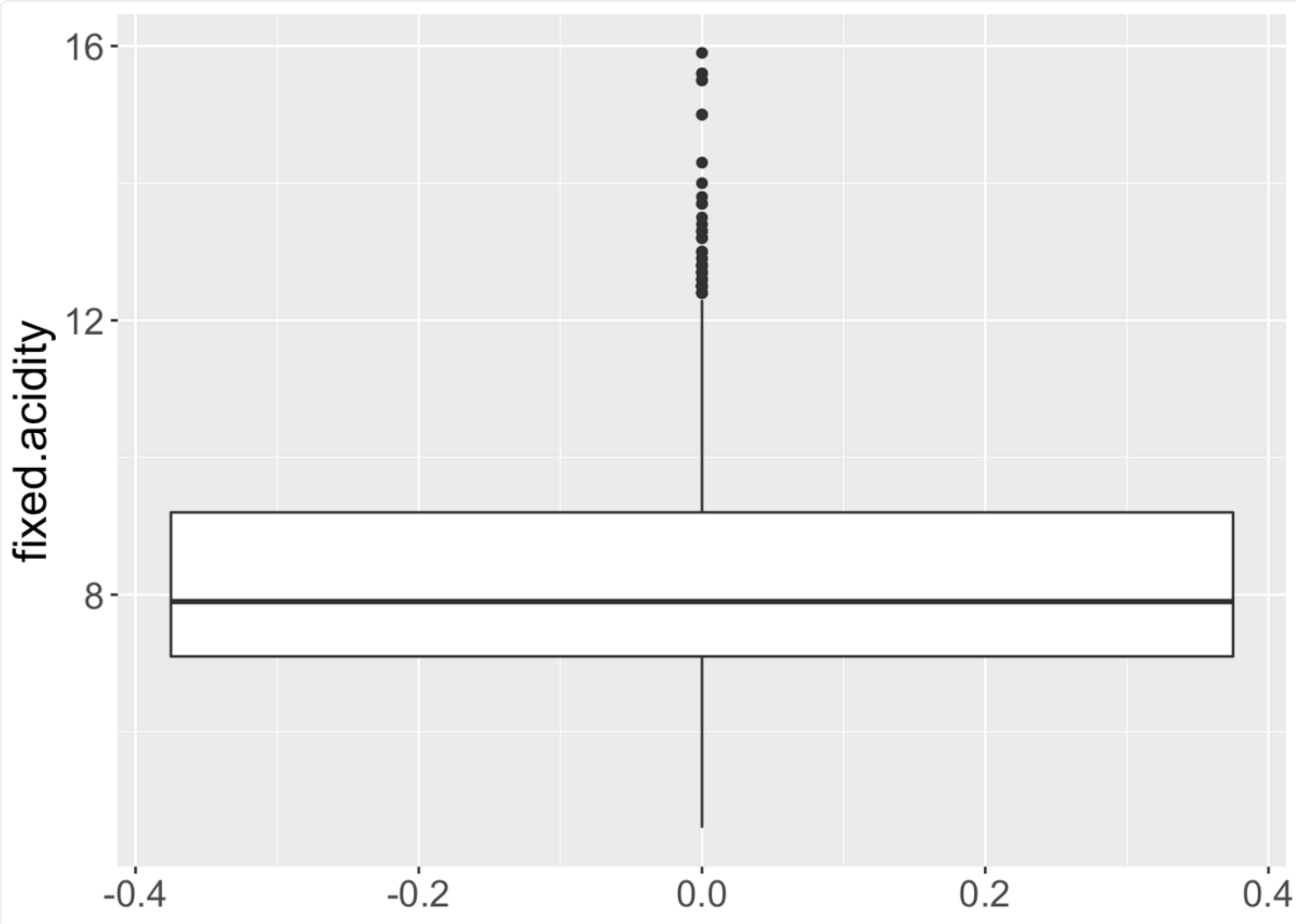


Exercise 4: Plot the histogram and density diagrams using the `geom_histogram` and `geom_density` functions from the *ggplot2* library into one figure. Use `binwidth = 1` for the histogram. You can make `geom_histogram` to plot the density instead of raw counts by using `mapping = aes(x = fixed.acidity, y=..density..)`, where `..density..` is `ggplot` calculated stat.



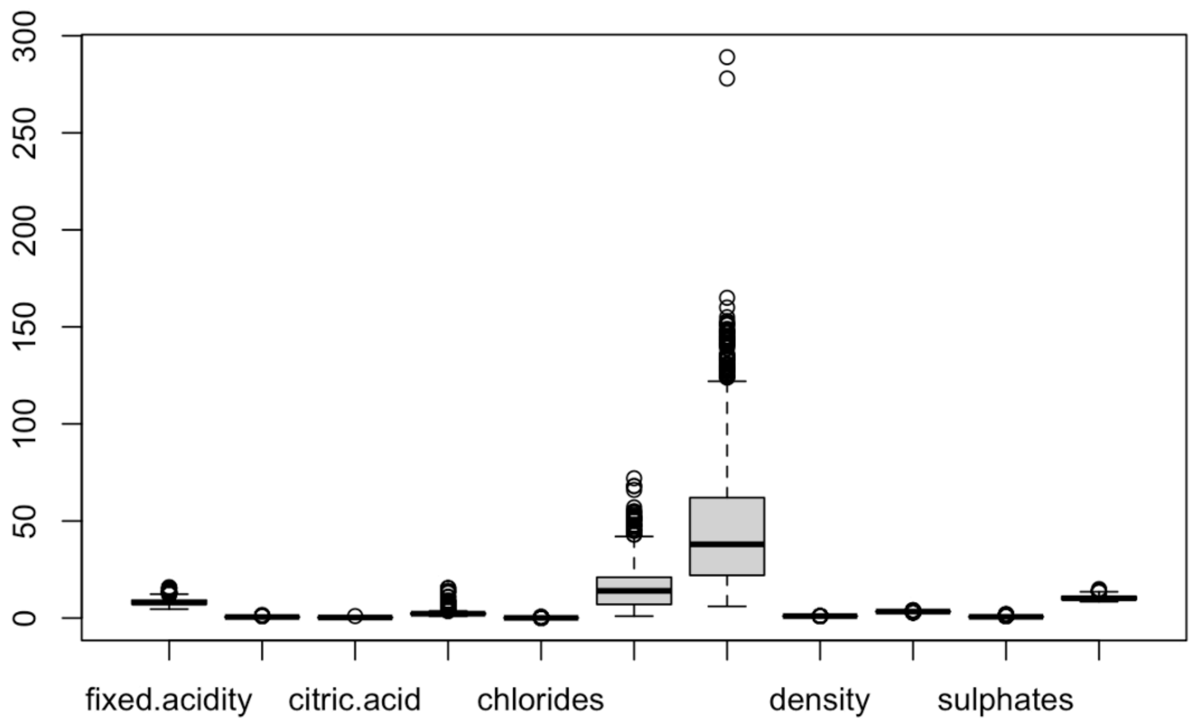
Box plots

Exercise 1: Use box plot (either the basic `boxplot()` function or the `geom_boxplot()` from the *ggplot2* library to plot the `wineData$fixed.acidity` variable. You should get a diagram that looks like this (if you use `geom_boxplot()` from *ggplot2*):



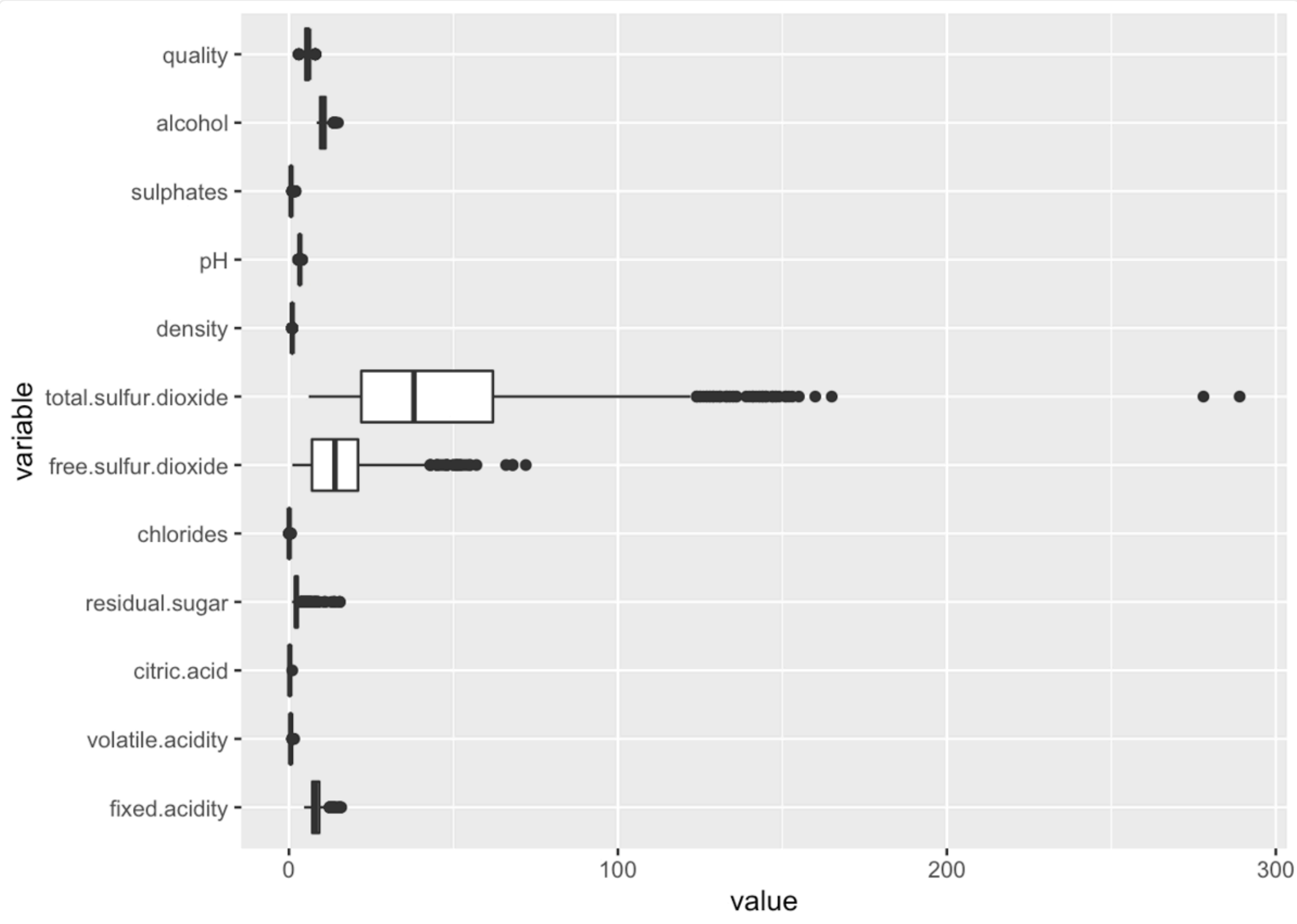
What does this plot tell you?

Exercise 2: Write R code to produce the box plots of all the variables (except for the `quality` column,) side-by-side in the same figure, like the diagram show below (if you use the basic `boxplot()`):



To produce the above side-by-side box plot using `ggplot` and flip the coordinate axes using `coord_flip()`, you can consider the code below, where `stack(wineData)` creates a new two-column data frame from `wineData` by stacking the 12 columns of `wineData` row-wise. The column names of this new data frame are `ind` and `values`. In the code below, we rename the axes to *variable* and *value*.

```
ggplot(stack(wineData)) +
  geom_boxplot(mapping = aes(x = ind, y = values)) +
  labs(x='variable', y='value') +
  coord_flip()
```



Try saving `stack(wineData)` to a variable `df` and inspect `df`. How many rows are there in the data frame `df`? Verify that `nrow(df)` is equal to `nrow(wineData) * ncol(wineData)`.

Exercise 3: Read the help page for `boxplot.stats` and write a line of R code to obtain the five value stats for `fixed.acidity`. You should get a vector that looks like this:

```
## [1]  4.6  7.1  7.9  9.2 12.3
```

Exercise 4: Try to find the median using your own understanding rather than using the build-in functions. **Hint:** sort the vector of values in `fixed.acidity` and find the index that splits the sorted values in half. Try to work out the code yourself before looking at the solution below.

```
median_index <- floor(length(wineData$fixed.acidity)/2) + 1
sorted <- sort(wineData$fixed.acidity)
cat("The median value of fixed.acidity is", sorted[median_index])
```

```
## The median value of fixed.acidity is 7.9
```

Optional exercise 1: Put your R code into a function called `myMedian` that would find and return the median of any vector of numerical values. So `myMedian(wineData$fixed.acidity)` should give the following (same as `median(wineData$fixed.acidity)`):

```
print(myMedian(wineData$fixed.acidity))
## [1] 7.9
```

Optional exercise 2: Modify your `myMedian` function to have an optional argument `q`. Let's call this new function `myQuantile`. It should return the appropriate quantile value of the input vector. For example, if the input vector is `wineData$fixed.acidity`, then the lower quartile (when `q` = 0.25) or upper quartile (when `q` = 0.75) should output the following:

```
cat("The lower quartile is", myQuantile(wineData$fixed.acidity, q=0.25))
## The lower quartile is 7.1
cat("The upper quartile is", myQuantile(wineData$fixed.acidity, q=0.75))
## The upper quartile is 9.2
```

Optional exercise 3: Write a function called `computeQuantiles` which should take in a data frame (e.g., variable `wineData` above) and, depending on the value of the optional argument `q`, output a vector of numbers containing the lower quartile, median, or upper quantile values. The length of the vector should be equal to the number of numeric columns in the data frame. **Hint:** Use `ncol()` to find out the number of variables (columns) in the data frame, `is.numeric()` to find out if a given column contains numerical values. For instance, the lower quartiles and upper quartiles of all the numerical columns of the *red wine* data (variable `wineData`) returned by the function should be

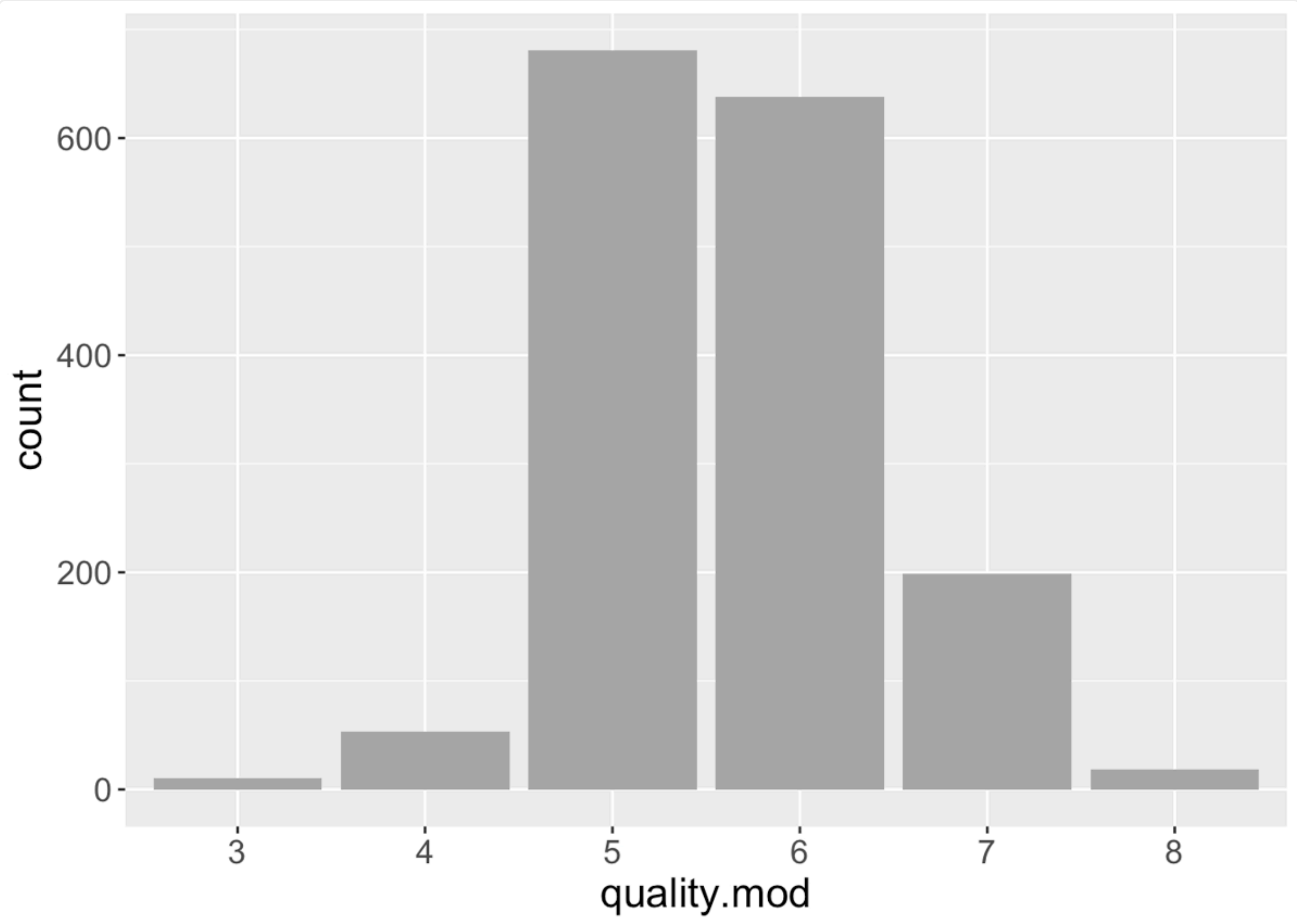
```
print(computeQuantiles(wineData, q=0.25))
##  [1]  7.1000  0.3900  0.0900  1.9000  0.0700  7.0000 22.0000  0.9956  3.210
## [10]  0.5500  9.5000  5.0000
print(computeQuantiles(wineData, q=0.75))
##  [1]  9.20000  0.64000  0.42000  2.60000  0.09000 21.00000 62.00000  0.9978
##  [9]  3.40000  0.73000 11.10000  6.00000
```

Bar chart

Use `str()` to look at the type of each column of the imported *red wine* data. The last column `quality` is of type integer, which should really be factors.

-

- Extract the `quality` column, turn it into factors, and store the results as a new column called `quality.mod` in the data frame.
- Use `ggplot`'s `geom_bar()` to plot the distribution of `quality.mod`. Repeat this operation for the distribution of `quality`. How different are the two plots? Repeat this operation again for another numerical variable, such as `fixed.acidity`, and see how the plot looks like.



Continue the learning journey on Swirl

Type

```
library(swirl)
swirl()
```

and complete lessons 1-6 and the **Exploratory Data Analysis** course. You may need to uninstall the R Programming course. Inside `swirl`, type

```
uninstall_course("R Programming")
```

`swirl()` can get tricky at times, if you run into problem, please contact the Lab Facilitator.

Try out R Markdown

Try out the R Markdown (.Rmd) if you haven't done so. You need to compose a R Markdown file for your EDA for project 1.

- **Cheat Sheet:** https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf

Previous
Markdown Basics

Next
Solution to Lab 03 Part C