Unit 02: Hermione Granger and the billboard dataset

Applied AI with R

Ferdinand Ferber and Wolfgang Trutschnig

Paris Lodron Universität Salzburg

3/4/24

Table of contents I

Hermione Granger and the billboards dataset

AI generated image for the prompt "Hermione Granger listening to music with a computer in the background at Hogwards."

Hermione Granger and the billboards dataset

- Using her analytical prowess, Hermione discovers that Voldemort's weakness lies not only in his Horcruxes, but also in his taste for music.
- She learns that the Dark Lord secretly loves Muggle music, particularly cheesy 2000s pop hits.
- Armed with this knowledge and access to the `billboard` dataset[1], Hermione comes up with the plan to study the rankings to find the perfect tune to infiltrate Voldemort's mind and distract him by some irresistable beats.

[1] The *Billboard Hot 100* is the music industry standard record chart in the US for songs, based on sales, streaming and radio airplay.

Section 1

Programming in R

What is R and where does it came from

- In the mid-1970s: Statistics was done using specialized FORTRAN libraries
- This is cumbersome for repetitive tasks
- And makes *exploratory data analysis* hard
- To combat this: The S language was designed at Bell Laboratories around 1975

What is R and where does it came from

- First version: Just a bunch of macros to transform `S` statements to FORTRAN subroutine calls.
- Operated in a read-eval-print-loop (REPL), i.e. interactively.
- Unique selling point (USP): A device-independent graphics system (for various printers, plotters, microfilm recorders and text terminals).
- Over the time, `S` grew into a proper, standalone programming language, specializing in statistical computing.
- The `R` programming language is an open source implementation of `S`.

Naming

- Naming of the `S` language/system: A pun on the `C` programming langue (also from Bell Labs).
- The `R` language refers to the `S` language in the same manner

Language design principles

- ...imperative (you tell the CPU what to do).
- ...interpreted (machine code is generated on the fly by the runtime, no compilation).
- ...dynamically typed (type errors will be caught at runtime).
- ...lexically scoped[2] (identifier resolution refers to regions of the source code).
- ...garbage collected (you don't need to allocate/free memory yourself).
- ...lazy (values are only computed when actually needed).

[2] With the exception of non-standard evaluation

R as a functional language

- Functions are first-class citicens, they can be stored in variables and passed around.
- Most R functions treat data as immutable. Function arguments are not passed by reference, but passed by value (as a deep copy).
- R supports anonymous functions (lambdas).
- But: No algebraic data types, no optimization for recursion, no sophisticated pattern matching, all variables are mutable.

R as an object-oriented language

- The R language has not one, but three major classes: S3, R6 and S4.
- The language is dynamic enough to allow you writing your own, if you like.
- Normal R users rarely interact with the object systems directly. It usually stays in the background and does its magic there.

If..then..else

- In R the `if..then..else` syntax is an expression that returns a value.
- The `else` part can be omitted (R will silently return a `NULL` value for that missing branch). Of course, it is also not neccessary to bind the value of the if-expression to a name.

```
x <- 5
res <- if (x < 10) {
    2 * x
} else {
    1 + x
}

res
```

```
[1] 10
```

For loops

- `for` loops are used to iterate over items in a vector or list and perform an action (i.e. side effect).
- The `next` statement skips the rest of the current iteration and the `break` statement exits the entire loop.

```
for (i in 1:10) {
  if (i < 3)
    next
  print(i)
  if (i >= 5)
    break
}
```

```
[1] 3
[1] 4
[1] 5
```

Remarks on control flow and imperative programming

- For most data analysis tasks, imperative programming (looping over a set of indices and fiddling around with arrays) is not the most elegant way.
- Instead, functional programming (i.e. filtering and mapping over lists) is advised.
- So this was the last time you see a `for` loop in this lecture and we will now dive into functional programming.

Defining functions

- Functions consist of three parts:
- A list of formal arguments, a function body, an environment.
- The environment will be created implicitly (and R is the only programming language that allows to manipulate it[3]).

```
# Define a function that adds two inputs and
# assign it to the name `myfun`
myfun <- function(x, y) {
    x + y
}

# Call the function
myfun(10, 20)
```

```
[1] 30
```

[3] See the lecture *Tom Riddle and the Dark Arts of R*

The return statement

- Usually, the last expression of a function will determine the return value of that function.
- But we can control this behaviour with the `return()` keyword:

```
myfun <- function(x, y) {
    if(x) {
        return(y)  # Early return
    }
    # This is the last expression and will
    # otherwise be returned
    y + 1
}
# Call the function
myfun(TRUE, 20)
```

```
[1] 20
```

do.call

- If the arguments of a function are already in a data structure, you can use `do.call()` to call the function:

```
myfun <- function(x, y) {
    x + y
}

args <- list(x = 10, y = 20)
do.call(myfun, args)
```

```
[1] 30
```

- `do.call()` is particularly useful, e.g., for binding dataframes of a list in a joint dataframe.

Pipe

- You already know the pipe. You can override first-argument-injection by using the *pipe placeholder* _ for it. It only works for named arguments:

```
myfun <- function(x, y) paste(x, y)

# First-argument-injection
"hello" |> myfun("world")
#> [1] "hello world"

# Explicit argument injection
"world" |> myfun("hello", y = _)
# [1] "hello world"

# Illegal:
"world" |> myfun("hello", _)
```

Lexical scoping

- We can use the assignment operator (e.g. `x <- 10`) to assign a name to a value. The reverse, finding the value to a given name, is called *identifier resolution*.
- One key concept is *scoping*: Every identifier is only valid in its scope. In *lexical scoping* the scope depends on the region of the source code.

```
x <- 10
myfun <- function() {
    y <- 20
    y
}

myfun()  # `myfun()` will set `y`
y        # But, we can't access `y` here!
```

Shadowing

- *Shadowing* allows you to re-use a variable name. Identifier resolution always starts in the current scope. If a value is found, it will be used. Otherwise, the search will continue in the outer scope. This process continues until the *global scope* is reached. An error is thrown if the value can't be found there.

```
x <- 1 ; y <- 2
myfun <- function() {
    # `x` shadows the other `x` in the outer scope
    x <- 10

    # `y` is not found in this scope,
    # proceed in the outer scope.
    c(x, y)
}
myfun()  #> [1] 10  2
```

Exercise

- After being correct so many times, Prof. Snape is furious about Hermione.
- He wants to challenge her and asks these questions. Help her answering them correctly:
- What does the following code return? Describe how each of the three c's is interpreted:

```
c <- 10
c(c = c)
```

Exercise

- What does the following function return?

```
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x^2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
```

Section 2

Lists and maps

Lists and maps

Lists are like atomic vectors, but with three differences:

- They can contain heterogenic data (a number, a string, a dataframe).
- They can be nested.
- They can be named.[4]

[4] Atomic vectors could also be named, but this is very uncommon. Lists, however, are most of the time named.

Creating lists

- You can create lists using the `list()` function:

```
list(name = "Rubeus Hagrid", birthyear = 1928)
```

```
$name
[1] "Rubeus Hagrid"

$birthyear
[1] 1928
```

Accessing lists

- Two ways for accessing list elements:

```
x <- list(name = "Rubeus Hagrid", birthyear = 1928)
x$name
```

```
[1] "Rubeus Hagrid"
```

and

```
x[["birthyear"]]
```

```
[1] 1928
```

- Note the analogy to accessing columns in dataframes.

Sublists

- It's possible to select a sublist via single brackets:

```
x["name"]
```

```
$name
[1] "Rubeus Hagrid"
```

Mapping over lists

- In data analysis, we often want to apply the same transformation to all elements of a list.
- We could use a `for` loop, but `map` is more elegant. The syntax `\(x) expr(x)` defines an *anonymous function*.

```
library(tidyverse)
mylist <- list(c(1,2,3), c(9,8,7))
mylist |> map_int(\(lst) sum(lst))
```

```
[1]  6 24
```

- There are different variants for maps: The `map()` version takes a list and returns a list.
- The variants `map_int()`, `map_dbl()`, `map_char()`, `map_lgl()` take lists and return integer vectors, float vectors, character vectors or boolean vectors resp.

imap

- Often it is necessary to iterate over the elements and their indices/names at the same time. The `imap()` function does this:

```
unnamed_list <- list("some argument", "another point")
imap(unnamed_list, \(elem, idx) paste0(idx, ") ", elem))
```

```
[[1]]
[1] "1) some argument"

[[2]]
[1] "2) another point"
```

imap

- This also works for named lists:

```
named_list <- list(first = "some argument",
                   second = "another point")
imap(named_list, \(elem, name) paste0(name, ": ", elem))
```

```
$first
[1] "first: some argument"

$second
[1] "second: another point"
```

map_if

- It is always possible to use `if...else` in the function that gets mapped over a container. But for simple cases there is a special case `map_if(cond, fn)`:

```
x <- list(c(1, 2, 3), c("a", "b", "c"))

# Apply the function `as.character` only for elements
# satisfying the condition `is.numeric`
x |> map_if(is.numeric, as.character)
```

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

[[2]]
[1] "a" "b" "c"
```

Filter

- We can use a filter to keep (or discard) all elements of a container that satisfy a given predicate. `keep(lst, pred)` keeps all elements of `lst` that satisfy `pred`. `discard(lst, pred)` discards them.

```
rep(10, times = 10) |>
    map(\(to) sample(1:to, size = 5)) |>
    keep(\(x) mean(x) > 6)
```

```
[[1]]
[1]  8  9  6  2 10

[[2]]
[1] 4 5 7 9 6

[[3]]
[1]  7  9  1 10  6
```

Filter

- Instead of testing elements for filtering, we can also test the whole list:

```
is_even <- function(x) x %% 2 == 0

3:10 |> every(is_even)
#> [1] FALSE

3:10 |> some(is_even)
#> [1] TRUE

3:10 |> none(is_even)
#> [1] FALSE
```

Pluck

- The base-`R` indexing operator doesn't work naturally in pipes.
- Therefore the `{purrr}` package provides the `pluck()` function, that also supports indexing into deeply nested structures:

```
obj <- list(
    list("a", list(1, foo = "bar")),
    list("b", list(2, foo = "baz"))
)
pluck(obj, 1, 2, "foo")  # same as obj[[1]][[2]][["foo"]]
```

```
[1] "bar"
```

```
pluck(obj, 10)
```

```
NULL
```

Pluck and map

- All map functions take also pluck-locations instead of a function.

```
obj <- list(
    list("a", list(1, foo = "bar")),
    list("b", list(2, foo = "baz"))
)

obj |> map_chr(1)
```

```
[1] "a" "b"
```

Exercise

- Download the `billboard.json` file (see 02_Hermione.R)
- Install the `jsonlite` package
- Use `jsonlite::fromJSON("billboard.json", simplifyDataFrame = F)` to parse the JSON as a list.
- Hermione wants to know which songs performed extraordinarily well:
    - Get all Muggle songs that got a rank 1-30 in just the first week.
    - What is the highest week one ranking ever achieved?
    - Which Muggle track stayed in the charts for the longest time?

Section 3

Timestamps

Handling timestamps

Time is sometimes complicated due to the following obstacles:

- UTC has leap seconds at irregular intervals.
- Calendars have leap years.
- Time zones.
- Daylight saving time.

Unix time

- To store UTC timestamps, one usually saves them as the number of SI-seconds (i.e., non-leap seconds) since the *UNIX epoch*, 1970-01-01 00:00:00 UTC+0.
- In `R`, the datatype holding UTC timestamps is called `POSIXct` and saves this number as a double.
- It is important to note that `R` will print `POSIXct` timestamps as a string using the local timezone, e.g. `"2024-02-15 15:24:23 CET"`.
- Hence, the same timestamp can show up different on other people's computers (if their timezone is not the same).

The lubridate package

- The `{lubridate}` package is part of the `tidyverse` and provides a number of verbs to work with timestamps.
- It proves useful in many situations and simplifies many tasks when working with timestamps or dates.

Parsing timestamps

- Timestamps are often stored as a string that needs to be parsed.
- {lubridate} provides a family of helpers to parse various kinds of strings.

```
ymd_hms("2017-11-28T14:03:00Z")
ymd_hms("2017-11-28T14:03:00.683+0230")
mdy_h("11/28/2017 2pm", tz = "US/Pacific")
dmy_hm("28.11.2017 14:03", tz = "Europe/Vienna")
mdy("November 28th, 2017")
```

- If no timezone is supplied, the local timezone is assumed. Seconds can be fractional.

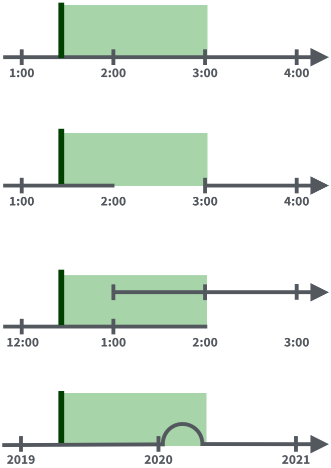
Timestamp components

- Once you have a timestamp in numeric format, you can decompose it into its components (not a complete list):

```
ts <- ymd_hms("2017-11-28T14:03:00.683+0230")

year(ts)      #> [1]  2017
day(ts)       #> [1]    28
wday(ts)      #> [1]     3
hour(ts)      #> [1]    11
tz(ts)        #> [1]   UTC
```

- Notice that you get the time in UTC. Weekdays start from Sunday.

Periods[5]



- In `{lubridate}` a *period* tracks changes in clock time, ignoring leap seconds/years.

[5] Image taken from the *lubridate cheat sheet*, Posit Software, PBC

Periods

```
ts <- ymd_hms("2017-11-28T14:03:00.683+0230")

ts + years(1)     #> [1] "2018-11-28 11:33:00 UTC"
ts + months(2)    #> [1] "2018-01-28 11:33:00 UTC"
ts + weeks(3)     #> [1] "2017-12-19 11:33:00 UTC"
ts + seconds(4)   #> [1] "2017-11-28 11:33:04 UTC"

years(1) + months(2) + weeks(3) + seconds(4)
#> [1] "1y 2m 21d 0H 0M 4S"
```

Periods

- This ususally works well... until it doesn't. Using periods, one can create nonexisting timestamps.
- The + operator returns NA in this case.
- The %m+% operator rolls imaginary dates back to the last day of the previous month.

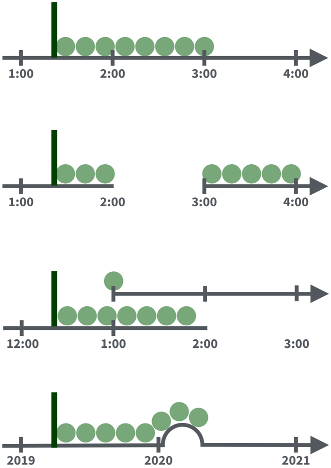
```
ts <- ymd("2024-01-31")
ts + months(1)
```

```
[1] NA
```

```
ts %m+% months(1)
```

```
[1] "2024-02-29"
```

Durations[6]



- In {lubridate} a *duration* tracks physical time, including leap seconds/years.
- All functions to create durations can be prepended with an d to create a duration instead. Durations are always in terms of SI-seconds.

[6]Image taken from the *lubridate cheat sheet*, Posit Software, PBC

Intervals

- You can construct intervals by given two timestamps as the borders and test if a third timestamp is inside the interval.

```
from <- ymd("2017-01-01")
to <- ymd("2017-01-31")
ts <- ymd("2017-01-15")

ts %within% interval(from, to)
```

```
[1] TRUE
```

Exercise

- Download the `billboard.csv` file (see 02_Hermione.R) and parse it into a dataframe.
- The `date.entered` column is stored as a string. Parse the string into a timestamp.
- The `week` is relative to the `date.entered`. Use the `{lubridate}` verbs to calculate the actual date.

Section 4

More on ggplot

More on ggplot

- In the last unit we already got an introduction to `{ggplot2}`.
- We will extend our knowlegde a bit and learn about how to add multiple layers to a ggplot, how to add titles and label and how to handle factorial variables.

Multiple layers

- Consider the following dataset

```
df <- billboard |>
  mutate(first_letter = str_sub(artist, 1, 1)) |>
  select(artist, track, first_letter, wk1)
```

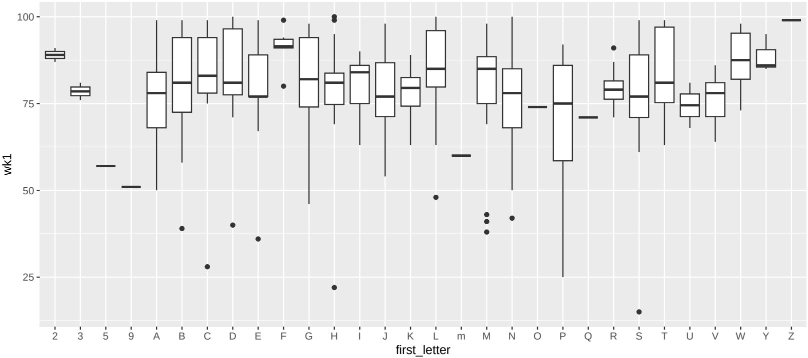
```
df |> slice_sample(n = 5)
```

artist	track	first_letter	wk1
Houston, Whitney	My Love Is Your Love	H	81
Offspring, The	Original Prankster	O	74
Lonestar	What About Now	L	78
Lonestar	Amazed	L	81
Jay-Z	I Just Wanna Love U …	J	58

Multiple layers

- We might wonder if artists starting with early letters perform on average better than artists with late letters in the alphabet. Let's draw a boxplot:

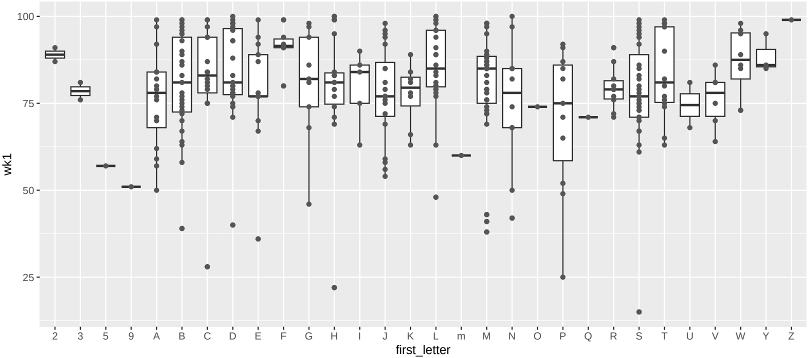
```
df |> ggplot(aes(x = first_letter, y = wk1)) +
  geom_boxplot()
```



Multiple layers

- Let's add another layer showing the actual points:

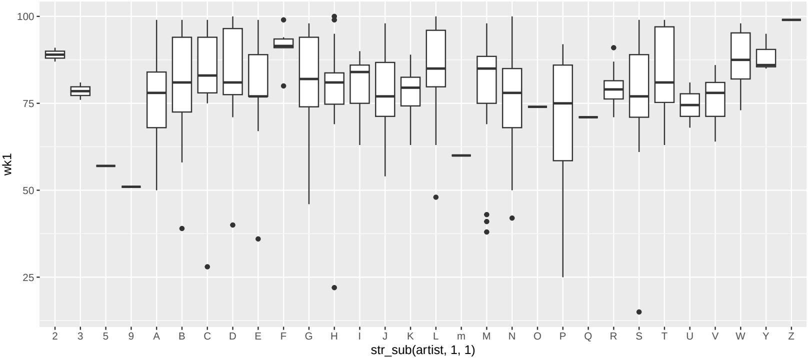
```
df |> ggplot(aes(x = first_letter, y = wk1)) +
  geom_boxplot() +
  geom_point(colour = "#555555")
```



Titles and labels

- Let's modify the code a little bit. Notice that {ggplot} magically determined the axis labels.

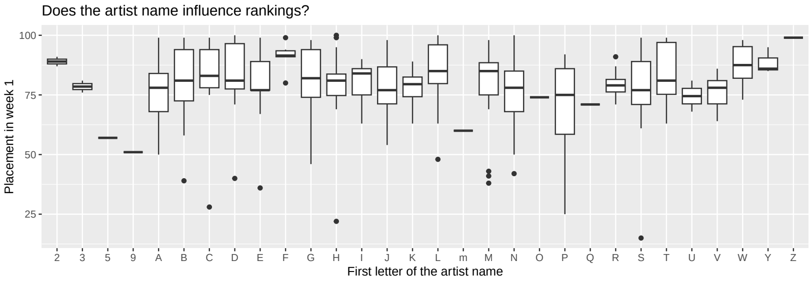
```
billboard |> ggplot(aes(x = str_sub(artist,1,1), y = wk1))
  geom_boxplot()
```



Titles and labels

- We might want to change that (and add a title).

```
billboard |> ggplot(aes(x = str_sub(artist,1,1), y = wk1))
  geom_boxplot() +
  labs(x = "First letter of the artist name",
       y = "Placement in week 1",
       title = "Does the artist name influence rankings?")
```



Forcats

- Consider the following dataframe (intentionally more complicated than necessary[7], but we want to make a point later on):

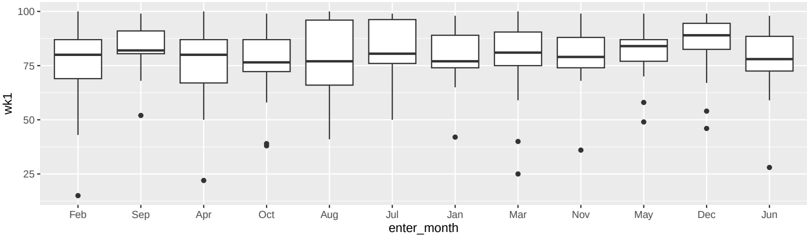
```
months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")

df <- billboard |>
  mutate(enter_month = month(date.entered)) |>
  rowwise() |>
  mutate(enter_month = months[[enter_month]]) |>
  mutate(enter_month = as.factor(enter_month)) |>
  select(artist, track, enter_month, wk1)
```

[7] We could use `month(date.entered, label=T)` instead

Forcats

Let's prpduce a boxplot again to see if we can spot a pattern between the month of the release and the distribution of ranks in the first week:

```
df |> ggplot(aes(x = enter_month, y = wk1)) +
  geom_boxplot()
```

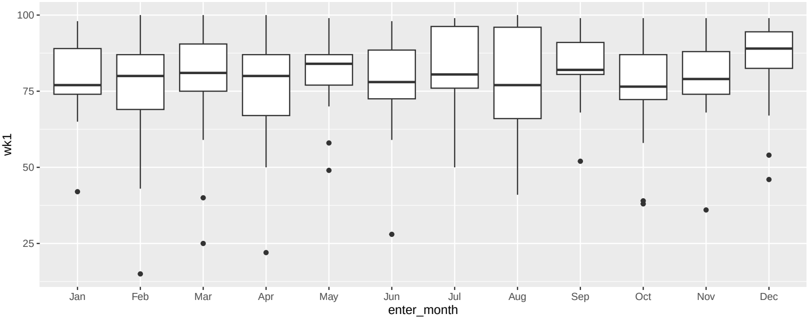


...the factors are sorted by their first occurence in the dataset.

Forcats

- Luckily, the {forcats} package (included in the tidyverse) comes to a rescue.

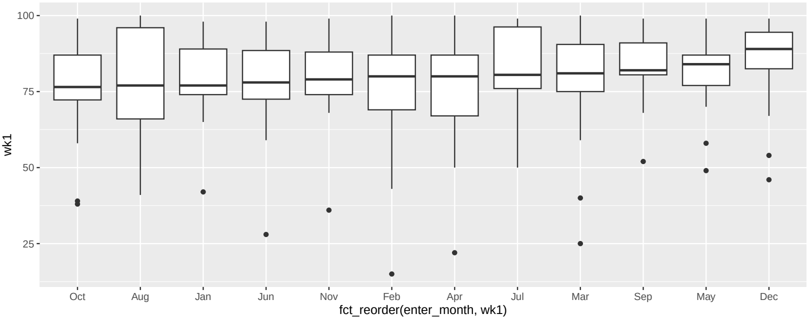
```
df |>
  mutate(enter_month = fct_relevel(enter_month, months)) |>
  ggplot(aes(x = enter_month, y = wk1)) +
  geom_boxplot()
```



Forcats

- Or we simply reorder a factor based on another variable in the dataset:

```
df |>
  ggplot(aes(x = fct_reorder(enter_month, wk1), y = wk1))
  geom_boxplot()
```



Exercise

- Use the `billboard.csv` file and parse it into a dataframe.
- Come up with a way to randomly sample 10 tracks.
- Produce a nice plot to show how the rankings changed over time. Include axis labels and a title.
- Hint: You may work with `geom_line()` and `geom_point()`.