R Tutorial

Hoang Bao Long, MD MPH

Table of contents

Giới thiệu

Đây là tập hợp các tutorial cho ngôn ngữ R. Trong khi pytutor cung cấp tutorial để tiếp cận Python một cách toàn diện, rtutor tập trung vào đơn giản hóa các khái niệm phức tạp nhưng có tính ứng dụng cao trong R.

1 So sánh data frame và tibble

```
library(dplyr)
library(knitr)

opts_chunk$set(
    message = FALSE
)

d <- data.frame(
    a = c(1, 0, 1, 0, 0),
    b = seq(5)
)

d
```

```
  a b
1 1 1
2 0 2
3 1 3
4 0 4
5 0 5
```

1.1 Tên hàng

Trong khi data frame có tên hàng, tibble không có.

```
rownames(d) <- c("a", "b", "c", "d", "e")
d
```

```
  a b
a 1 1
b 0 2
c 1 3
```

```
d 0 4
e 0 5
```

```
d_tib <- tibble(d)
d_tib
```

```
# A tibble: 5 x 2
      a     b
  <dbl> <int>
1     1     1
2     0     2
3     1     3
4     0     4
5     0     5
```

Muốn thêm tên hàng cho tibble, chúng ta tạo một cột mới.

```
d_tib <- tibble(tibble::rownames_to_column(d, "id"))
d_tib
```

```
# A tibble: 5 x 3
  id        a     b
  <chr> <dbl> <int>
1 a         1     1
2 b         0     2
3 c         1     3
4 d         0     4
5 e         0     5
```

Cũng vì lí do này, data frame có thể slice theo tên hàng, còn tibble thì không.

```
d[c("a", "b"), ]
```

```
  a b
a 1 1
b 0 2
```

```
d_tib %>% filter(id %in% c("a", "b"))
```

```
# A tibble: 2 x 3
  id        a     b
  <chr> <dbl> <int>
1 a         1     1
2 b         0     2
```

1.2 Slice cột

Để truy cập vào dữ liệu của một cột trong data frame, chúng ta có những cách sau.

```
d$a
```

```
[1] 1 0 1 0 0
```

```
d[, "a"]
```

```
[1] 1 0 1 0 0
```

```
d[, c("a")]
```

```
[1] 1 0 1 0 0
```

Những cách này trả về vector nếu chỉ slice một cột. Muốn giữ nguyên định dạng data frame (gọi là subset), chúng ta làm như sau.

```
d["a"]
```

```
  a
a 1
b 0
c 1
d 0
e 0
```

Thêm một cặp ngoặc vuông nữa, bạn cũng sẽ lấy được vector giá trị của cột.

```
d[["a"]]
```

```
[1] 1 0 1 0 0
```

Slicing bằng giá trị chuỗi kí tự của tên cột thuận lợi cho lập trình.

```
col_name <- "a"
d[[col_name]]
```

```
[1] 1 0 1 0 0
```

Đối với tibble, slicing luôn trả về subset.

```
d_tib["a"]
```

```
# A tibble: 5 x 1
      a
  <dbl>
1     1
2     0
3     1
4     0
5     0
```

```
d_tib[, "a"]
```

```
# A tibble: 5 x 1
      a
  <dbl>
1     1
2     0
3     1
4     0
5     0
```

```
d_tib[, c("a")]
```

```
# A tibble: 5 x 1
      a
  <dbl>
1     1
2     0
3     1
4     0
5     0
```

Để lấy vector, bạn có thể dùng các cách sau.

```
d_tib$a
```

```
[1] 1 0 1 0 0
```

```
d_tib[["a"]]
```

```
[1] 1 0 1 0 0
```

```
d_tib %>% pull(a)
```

```
[1] 1 0 1 0 0
```

Do vậy, muốn lập trình với tibble, bạn có thể làm như sau.

```
d_tib[[col_name]]
```

```
[1] 1 0 1 0 0
```

Hoặc

```
d_tib %>% pull(!!rlang::sym(col_name))
```

```
[1] 1 0 1 0 0
```

Tương tự, nếu muốn subset, bạn có thể làm như trên với hàm `select()`.

```
d_tib %>% select(!!rlang::sym(col_name))
```

```
# A tibble: 5 x 1
      a
  <dbl>
1     1
2     0
3     1
4     0
5     0
```

2 Lặp trong R

```
library(knitr)
opts_chunk$set(
    message = FALSE
)
```

2.1 Ví dụ 1

2.1.1 Cách lặp truyền thống

Chúng ta sẽ đến với một bài toán đơn giản để mô phỏng cho việc lặp trong R. Chúng ta có một vector `v` chứa các phần tử là các dữ liệu số, và chúng ta muốn cộng thêm 1 cho mỗi phần tử. Chúng ta sẽ lưu kết quả cộng vào vector `v_a`.

```
v <- c(1, 2, 3)
v_a1 <- v

for (i in v) {
    v_a1[i] <- v[i] + 1   # cộng 1 cho mỗi phần tử thứ i của vector v
}

print(v)
```

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

```
print(v_a1)
```

```
[1] 2 3 4
```

Thao tác trên đây mô phỏng việc sử dụng vòng lặp `for` để thực hiện các tính toán giống nhau lặp đi lặp lại (thao tác cơ bản thì giống nhau, tham số đầu vào có thể khác nhau). Thông thường chúng ta sẽ gói gọn các công việc này vào trong một **hàm** để dễ tái sử dụng cho nhiều lần khác nhau, cũng như dễ quản lí mã lệnh và chỉnh sửa khi cần. Chẳng hạn, bạn có thể viết hàm `add_y()` để cộng thêm `y` vào mỗi phần tử trong vector, và mặc định `y` bằng 1.

```
add_y <- function(x, y = 1) {
    x + y
}

add_y(1)
```

```
[1] 2
```

Vòng lặp của chúng ta trở thành như sau.

```
for (i in v) {
    v_a1[i] <- add_y(v[i])
}

print(v_a1)
```

```
[1] 2 3 4
```

2.1.2 Sử dụng `purrr::map_dbl()`

Bạn có thể đơn giản hóa vòng lặp này bằng hàm `map_dbl()` trong thư viện `purrr`. Hàm này sẽ lặp qua từng phần tử của vector `v`, thực hiện hàm `add_y()` trên từng phần tử đó, và trả về một vector là kết quả thực hiện trên toàn bộ vector `v`. Tốc độ lặp của hàm này nhanh hơn so với việc sử dụng vòng lặp `for`, do không phải truy xuất bộ nhớ liên tục và các tối ưu về vectorization khác.

```
library(purrr)

v_a2 <- map_dbl(v, add_y)

print(v_a2)
```

```
[1] 2 3 4
```

Nếu bạn chỉ muốn thực hiện một phép cộng đơn giản, chúng ta có thể làm nhanh hơn nữa như dưới đây. Cách viết thẳng hàm vào trong câu lệnh mà không khai báo hàm gọi là hàm lambda hay anonymous function. Hệ sinh thái Tidyverse cho phép bạn viết tắt việc khai báo hàm lambda như dòng lệnh tiếp theo (tính ra vector `v_a4`), `.x` là đại diện cho đối số của hàm lambda (tương tự `x` trong `function(x)`).

```
v_a3 <- map_dbl(v, function(x) x + 1)
v_a4 <- map_dbl(v, ~ .x + 1)

print(v_a3)
```

```
[1] 2 3 4
```

```
print(v_a4)
```

```
[1] 2 3 4
```

2.2 Ví dụ 2

Trong ví dụ phức tạp hơn dưới đây, chúng ta sẽ cùng thao tác trên một data frame.

```
library(dplyr)

set.seed(0)
d <- data.frame(
    id = seq(10),
    a = rnorm(10),
    b = rgamma(10, 1)
)

d %>% kable()
```

id	a	b
1	1.2629543	1.1857109
2	-0.3262334	0.0946191
3	1.3297993	0.1572015
4	1.2724293	0.3108054
5	0.4146414	0.4687319

id	a	b
6	-1.5399500	0.0681973
7	-0.9285670	1.2492921
8	-0.2947204	1.0081313
9	-0.0057672	1.3609450
10	2.4046534	1.2059882

Chúng ta sẽ tính tổng bình phương giá trị của tất cả các bản ghi trong một cột. Để làm việc này, chúng ta sẽ viết hàm `ssq()`. Hàm `sapply()` mà chúng ta sử dụng có tính năng tương tự hàm `map_dbl()`, và là một hàm sẵn có trong R.

```
ssq <- function(v) {
    sum(sapply(v, function(x) x ^ 2))
}

ssq(d$a)
```

```
[1] 14.36379
```

Bài toán phức tạp hơn là chúng ta muốn chạy hàm này cho nhiều cột khác nhau. Bên cạnh đó mình cũng muốn trả về trung bình của tổng bình phương, và thêm tên cột vào để dễ theo dõi. Vì vậy, mình tạo thêm một hàm `calc_ssq()` với đối số `name` là tên cột mà mình muốn thực hiện các phép tính toán. Hàm này sẽ trả về một data frame có một dòng, là kết quả tính toán tương ứng với cột trong `name`.

```
calc_ssq <- function(d, name) {
    data.frame(
        name = name,
        ssq = ssq(d[name])
    ) %>%
        mutate(
            mean_ssq = ssq / nrow(d)
        )
}

calc_ssq(d, "a")
```

```
  name      ssq mean_ssq
1    a 14.36379 1.436379
```

Cái hay của `purrr` là nó cung cấp hàm `map_df()`, tự động gộp các data frame sau mỗi lần chạy vào với nhau, và tạo thành một data frame duy nhất.

```
vars_to_calc <- c("a", "b")
map_df(vars_to_calc, ~ calc_ssq(d, .x)) %>% kable()
```

name	ssq	mean_ssq
a	14.363786	1.4363786
b	7.644174	0.7644174

3 Tạo nhóm từ biến liên tục

```
library(knitr)
opts_chunk$set(
    message = FALSE
)
```

3.1 Mục đích

Trong nghiên cứu, bạn thường thu thập một số biến liên tục, sau đó dựa vào các điểm cắt để phân thành các nhóm. Chẳng hạn, chúng ta thường phân loại chỉ số khối cơ thể (BMI) thành các nhóm nhẹ cân (BMI < 18.5), bình thường (18.5 - <25), thừa cân (25 - <30), và béo phì (>30). Chúng ta không nên thu thập ngay phân loại BMI, mà nên thu thập các chỉ số chiều cao và cân nặng, sau đó sử dụng phần mềm để tính ra BMI và phân nhóm.

Chúng ta sẽ cùng xem một bộ số liệu như sau.

```
library(dplyr)

set.seed(0)
n <- 10

d <- data.frame(
    id = seq(n),
    sex = sample(c(1, 2), n, replace = TRUE),   # 1=Nam 2=Nữ
    bmi = runif(n, 16.5, 35)
)

d %>% kable()
```

id	sex	bmi
1	2	17.64305
2	1	20.31053
3	2	19.76630

id	sex	bmi
4	1	29.20992
5	1	23.60592
6	2	30.74207
7	1	25.70744
8	1	29.77594
9	1	34.85026
10	2	23.53065

3.2 Phân nhóm từ một biến liên tục

Để phân loại các nhóm BMI, chúng ta chỉ cần sử dụng một biến BMI là đủ. Đặc điểm của các nhóm phân loại từ BMI là mỗi cá thể chỉ được phân loại vào đúng một nhóm. Thông thường, với cách phân loại này, chúng ta sẽ mã hóa các nhóm tăng dần từ giá trị 1 (ví dụ, 1 đến 4 cho 4 nhóm BMI).

Bạn có thể làm rất nhanh việc phân nhóm này trong R bằng việc sử dụng hàm `cut()`. Cung cấp cho hàm này một đối số là các khoảng giá trị điểm cắt (bao gồm cả giá trị thấp nhất và cao nhất), hàm sẽ trả về cho bạn một factor của phân nhóm tạo ra từ biến liên tục. Các đối số khác trong hàm `cut()` bạn tự tham khảo trong phần documentation của R nhé (gõ `?cut` trong R console và ấn Enter).

```
d$bmi %>% cut(c(0, 18.5, 25, 30, 100),
    labels = seq(4), right = FALSE, ordered_result = TRUE)
```

```
 [1] 1 2 2 3 2 4 3 3 4 2
Levels: 1 < 2 < 3 < 4
```

Một cách khác tuy nhìn không thuận tiện nhưng lại thuận lợi hơn về mặt tính toán là chỉ sử dụng các biểu thức logic và số học. Chúng ta sẽ xem kết quả trước (hãy tập trung vào nội dung của hàm `mutate()`), sau đó mình sẽ giải thích chi tiết.

```
d %>%
    select(id, bmi) %>%
    mutate(
        bmi_group = 1 + (bmi >= 18.5) + (bmi >= 25) + (bmi >= 30)
    ) %>%
    kable()
```

id	bmi	bmi_group
1	17.64305	1
2	20.31053	2
3	19.76630	2
4	29.20992	3
5	23.60592	2
6	30.74207	4
7	25.70744	3
8	29.77594	3
9	34.85026	4
10	23.53065	2

Cách làm này dựa trên nguyên tắc về phân loại mình nêu trên, đấy là mỗi bệnh nhân chỉ được phân vào một trong 4 nhóm, và các nhóm đánh số từ 1 đến 4. Theo công thức trong hàm `mutate()`, những người có BMI cao sẽ thỏa mãn cả các điều kiện của BMI thấp hơn, và do đó, "tổng điểm" sẽ cao hơn. Nếu chưa mường tượng ra, bạn hãy nhìn bảng dưới đây và tự suy ngẫm.

```
d %>%
    select(id, bmi) %>%
    mutate(
        bmi_2 = as.numeric(bmi >= 18.5),
        bmi_3 = as.numeric(bmi >= 25),
        bmi_4 = as.numeric(bmi >= 30),
        bmi_group = 1 + (bmi >= 18.5) + (bmi >= 25) + (bmi >= 30)
    ) %>%
    kable()
```

id	bmi	bmi_2	bmi_3	bmi_4	bmi_group
1	17.64305	0	0	0	1
2	20.31053	1	0	0	2
3	19.76630	1	0	0	2
4	29.20992	1	1	0	3
5	23.60592	1	0	0	2
6	30.74207	1	1	1	4
7	25.70744	1	1	0	3
8	29.77594	1	1	0	3
9	34.85026	1	1	1	4
10	23.53065	1	0	0	2

3.3 Phân loại từ nhiều biến

Nếu tiêu chí phân loại của bạn như sau:

- 1=Nam BMI >30
- 2=Nam BMI >28
- 3=Nam BMI >25 và <=30 hoặc Nữ BMI >23 và <=28
- 4=Còn lại

thì bạn sẽ gặp khó khăn trong việc dùng công thức ở trên. Tuy nhiên, chúng ta vẫn có thể tổng quát hóa công thức toán học đó như sau.

```
d %>%
    mutate(
        ploai = 1 * (sex == 1) * (bmi > 30) +
            2 * (sex == 2) * (bmi > 28) +
            3 * (
                (sex == 1) * (bmi > 25) * (bmi <= 30) +
                (sex == 2) * (bmi > 23) * (bmi <= 28)
            ),
        ploai = ploai * (ploai > 0) + 4 * (ploai == 0)
    ) %>%
    kable()
```

id	sex	bmi	ploai
1	2	17.64305	4
2	1	20.31053	4
3	2	19.76630	4
4	1	29.20992	3
5	1	23.60592	4
6	2	30.74207	2
7	1	25.70744	3
8	1	29.77594	3
9	1	34.85026	1
10	2	23.53065	3

Tư duy trong giải pháp này là bạn có thể tạo ra các phân nhóm chỉ từ phép cộng và phép nhân. Phép nhân sẽ tương đương với toán tử `AND` (nếu `A` và `B` chỉ là 0 hoặc 1 thì `A AND B` $= 1$ khi và chỉ khi `A` $=$ `B` $= 1$), còn phép cộng sẽ tương đương với toán tử `OR` nếu như `A` và `B` không bao giờ đồng thời bằng 1 (khi đó `A OR B` $= 0$ khi và chỉ khi `A` $=$ `B` $= 0$). Và khi một trường hợp là đúng (`TRUE`, =1), bạn chỉ cần nhân với code tương ứng của nó, thế là xong. Để thực

hiện theo cách này, chúng ta cần đảm bảo các chuỗi điều kiện chỉ đúng cho một trong các phân nhóm; nếu có nhiều phân nhóm cùng đúng (một người có thể thuộc về nhiều nhóm) thì lệnh sẽ tạo ra các code mới ngoài dự kiến (là tổng của các phân nhóm thỏa mãn điều kiện).

Bạn có thể dùng hàm `case_when()` để đơn giản hóa biểu thức tính toán ở trên.

```
d %>%
    mutate(
        ploai = case_when(
            (sex == 1) & (bmi > 30) ~ 1,
            (sex == 2) & (bmi > 28) ~ 2,
            (sex == 1) & (bmi > 25) | (sex == 2) & (bmi > 23) ~ 3,
            TRUE ~ 4
        )
    ) %>%
    kable()
```

id	sex	bmi	ploai
1	2	17.64305	4
2	1	20.31053	4
3	2	19.76630	4
4	1	29.20992	3
5	1	23.60592	4
6	2	30.74207	2
7	1	25.70744	3
8	1	29.77594	3
9	1	34.85026	1
10	2	23.53065	3

Cách làm của hàm `case_when()` sẽ giúp bạn bớt đi được một số điều kiện (ví dụ, ở nhóm 3 bạn không cần thêm điều kiện BMI <=30 cho nam và <=28 cho nữ). Khi các điều kiện cho nhóm 1 và 2 không thỏa mãn, thì điều kiện BMI <= 30 hoặc 28 đã tự động được thỏa mãn.

4 Tư duy thao tác dữ liệu trong R

```
library(knitr)
opts_chunk$set(
    message = FALSE
)
```

4.1 Bài toán

Hôm nay mình sẽ giới thiệu với các bạn một bài toán phân tích số liệu đơn giản trong R. Bài toán này nhằm giúp các bạn hiểu rõ hơn cách thức tư duy khi giải quyết một bài toán bằng lập trình.

Chúng ta có một bộ số liệu với 4 biến là `stt` (số thứ tự), `gioi` (giới: Nam, Nữ), `do_tuoi` (độ tuổi: <18, 18-45, >45), `hailong` (điểm hài lòng, từ 0 đến 100), và `qol` (điểm chất lượng cuộc sống, từ 0 đến 100). Đây là một bộ số liệu do mình tạo ra ngẫu nhiên ra thôi.

```
library(dplyr)
library(tidyr)

set.seed(0)
n <- 1000

d <- data.frame(
    stt = seq(n),
    gioi = factor(sample(c(1, 2), n, TRUE),
        levels = c(1, 2), labels = c("Nam", "Nu")),
    do_tuoi = factor(sample(c(1, 2, 3), n, TRUE),
        levels = c(1, 2, 3), labels = c("<18", "18-45", ">45")),
    hailong = round(runif(n, 0, 100), 1),
    qol = round(runif(n, 0, 100), 1)
)

d %>% head(5) %>% kable()
```

stt	gioi	do_tuoi	hailong	qol
1	Nu	>45	33.0	76.3
2	Nam	>45	69.7	23.6
3	Nu	>45	35.4	28.6
4	Nam	<18	40.6	31.8
5	Nam	>45	30.8	92.2

Việc của chúng ta sẽ là tạo ra một bảng phân tích kết quả trông như sau:

Đặc điểm	Nhóm	Hài lòng, mean (SD)
Giới	Nam	...
Giới	Nữ	...
Độ tuổi	<18	...
Độ tuổi	18-45	...
Độ tuổi	>45	...

Có nhiều cách để làm việc này. Cách mà mình giới thiệu hôm nay khá trực tiếp, mặc dù có thể không phải là cách tối ưu.

4.2 Các phép thao tác với số liệu

Có 4 phép thao tác (manipulate) số liệu chính:

- *Tái cấu trúc* (reshaping): chuyển số liệu từ dạng bảng dài sang dạng bảng ngang và ngược lại (pivot giữa long / wide data), xếp chồng các số liệu lên nhau (stacking / unstacking), v.v.. Thư viện sử dụng cho reshaping là `tidyr`.
- *Nhóm* (grouping): các số liệu thuộc cùng một nhóm được xếp chung với nhau để phục vụ một mục đích nào đó. Bạn chắc đã làm quen với hàm `dplyr::group_by()` cho việc này.
- *Chuyển dạng* (transformation): chuyển số liệu cá thể thành các giá trị mới dựa trên một phép biến đổi nào đó như chuẩn hóa (normalization), logarit, chia nhóm (categorization), v.v.. Mọi phép chuyển dạng đều thông qua hàm `dplyr::mutate()` và các biến thể của nó.
- *Tổng hợp* (aggregation): tính toán các chỉ số tổng hợp (trung bình, tỉ lệ phần trăm, v.v.) từ số liệu cá thể. Hầu hết các phép tổng hợp đều thông qua hàm `dplyr::summarize()` và các biến thể của nó.

Bằng những phép thao tác số liệu này, chúng ta có thể tạo ra mọi kết quả mong muốn từ một bộ số liệu gốc.

4.3 Tư duy thao tác số liệu

Nhìn vào bộ số liệu gốc, mình nghĩ rằng sẽ cần tạo ra một (hoặc nhiều) bộ số liệu trung gian để phục vụ việc tính toán như trên. Bộ số liệu trung gian sẽ có cấu trúc như thế nào? Quan sát bảng phân tích kết quả, chúng ta thấy rằng:

- Cột "Đặc điểm" là tên các biến mà chúng ta có trong bộ số liệu gốc.
- Cột "Nhóm" là các giá trị của các biến "Đặc điểm" có trong bộ số liệu gốc.
- Cột "Hài lòng" là kết quả tổng hợp của điểm hài lòng trong bộ số liệu gốc.

Vậy bộ số liệu trung gian của mình có thể là kết quả chuyển từ dạng ngang (các biến xếp thành từng cột) sang dạng dài (các biến xếp chồng lên nhau) của hai biến `gioi` và `do_tuoi`, còn giữ lại biến `hailong`. Ví dụ:

stt	variable	value	hailong
1	gioi	Nu	33.0
1	do_tuoi	>45	33.0
2	gioi	Nam	69.7
2	do_tuoi	>45	69.7

Sau đó mình chỉ việc tạo ra các nhóm của Đặc điểm (`variable`) và Nhóm (`value`) để tổng hợp (aggregate) cột `hailong`. Hãy cùng xem chúng ta thực thi kế hoạch này.

4.3.1 Bước 1: Reshaping

```
d_long <- d %>%
    select(gioi, do_tuoi, hailong) %>%
    pivot_longer(cols = c(gioi, do_tuoi), names_to = "variable")

d_long %>% head() %>% kable()
```

hailong	variable	value
33.0	gioi	Nu
33.0	do_tuoi	>45
69.7	gioi	Nam
69.7	do_tuoi	>45
35.4	gioi	Nu
35.4	do_tuoi	>45

4.3.2 Bước 2: Grouping và Aggregation

```
d_agg <- d_long %>%
    group_by(variable, value) %>%
    summarize(
        mean = mean(hailong),
        sd = sd(hailong)
    )

d_agg %>% kable()
```

variable	value	mean	sd
do_tuoi	<18	49.97685	28.50213
do_tuoi	18-45	50.87988	29.54522
do_tuoi	>45	49.84675	28.96249
gioi	Nam	49.64345	29.21245
gioi	Nu	50.84234	28.77515

4.3.3 Bước 3: Transformation

```
d_agg %>%
    mutate(
        mean_sd = sprintf("%.1f (%.1f)", mean, sd)
    ) %>%
    select(variable, value, mean_sd) %>%
    kable()
```

variable	value	mean_sd
do_tuoi	<18	50.0 (28.5)
do_tuoi	18-45	50.9 (29.5)
do_tuoi	>45	49.8 (29.0)
gioi	Nam	49.6 (29.2)
gioi	Nu	50.8 (28.8)

4.4 Module hóa công việc

Như ở trên, bạn đã thấy chúng ta thống kê được mean (SD) của điểm hài lòng. Nhưng nếu chúng ta muốn làm tương tự như vậy với điểm chất lượng cuộc sống và gộp chung kết quả với điểm hài lòng thì bạn sẽ làm thế nào? Tất nhiên, bạn hoàn toàn có thể thêm tạo ra các biến `mean_qol` và `sd_qol` cho điểm chất lượng cuộc sống trong Bước 2, nhưng nếu không phải là 2 biến mà là 20 biến, thì việc đó sẽ rất phiền toái, hoặc nếu bạn phải thay đổi kế hoạch phân tích, loại bỏ biến `qol` và thêm biến khác vào. Đây là lúc bạn cần dùng đến **hàm**, và chúng ta gọi đây là module hóa công việc.

Ba bước ở trên có thể được tóm gọn trong một hàm như sau.

```
library(rlang)

get_mean_sd <- function(d, group_vars, outcome_var) {
    d %>%
        # Bước 1
        select(all_of(c(group_vars, outcome_var))) %>%
        pivot_longer(cols = all_of(group_vars), names_to = "variable") %>%

        # Bước 2
        group_by(variable, value) %>%
        summarize(
            mean = mean(!!sym(outcome_var)),
            sd = sd(!!sym(outcome_var))
        ) %>%

        # Bước 3
        mutate(
            outcome = outcome_var,
            mean_sd = sprintf("%.1f (%.1f)", mean, sd)
        ) %>%
        select(variable, value, outcome, mean_sd)
}

group_vars <- c("gioi", "do_tuoi")
outcome_var <- "hailong"
get_mean_sd(d, group_vars, outcome_var) %>% kable()
```

variable	value	outcome	mean_sd
do_tuoi	<18	hailong	50.0 (28.5)
do_tuoi	18-45	hailong	50.9 (29.5)

variable	value	outcome	mean_sd
do_tuoi	>45	hailong	49.8 (29.0)
gioi	Nam	hailong	49.6 (29.2)
gioi	Nu	hailong	50.8 (28.8)

Và chúng ta có thể tự động hóa việc tính toán này cho nhiều biến kết cục khác nhau.

```
library(purrr)

outcome_vars <- c("hailong", "qol")
map_df(outcome_vars, ~ get_mean_sd(d, group_vars, .x)) %>% kable()
```

variable	value	outcome	mean_sd
do_tuoi	<18	hailong	50.0 (28.5)
do_tuoi	18-45	hailong	50.9 (29.5)
do_tuoi	>45	hailong	49.8 (29.0)
gioi	Nam	hailong	49.6 (29.2)
gioi	Nu	hailong	50.8 (28.8)
do_tuoi	<18	qol	49.3 (30.0)
do_tuoi	18-45	qol	48.7 (30.4)
do_tuoi	>45	qol	49.3 (30.7)
gioi	Nam	qol	49.9 (29.7)
gioi	Nu	qol	48.4 (31.0)

Tất nhiên, nếu bạn muốn chuyển sang dạng nhiều cột kết quả của các biến kết cục thì cũng rất đơn giản, nó chỉ là pivot từ dạng long sang wide thôi.

```
map_df(outcome_vars, ~ get_mean_sd(d, group_vars, .x)) %>%
    pivot_wider(id_cols = c(variable, value),
        names_from = outcome, values_from = mean_sd) %>%
    kable()
```

variable	value	hailong	qol
do_tuoi	<18	50.0 (28.5)	49.3 (30.0)
do_tuoi	18-45	50.9 (29.5)	48.7 (30.4)
do_tuoi	>45	49.8 (29.0)	49.3 (30.7)
gioi	Nam	49.6 (29.2)	49.9 (29.7)
gioi	Nu	50.8 (28.8)	48.4 (31.0)

Và bạn có thể gói tiếp chức năng này trong một hàm như sau:

```
get_mean_sd_all <- function(d, group_vars, outcome_vars) {
    map_df(outcome_vars, ~ get_mean_sd(d, group_vars, .x)) %>%
        pivot_wider(id_cols = c(variable, value),
            names_from = outcome, values_from = mean_sd)
}

get_mean_sd_all(d, group_vars, outcome_vars) %>% kable()
```

variable	value	hailong	qol
do_tuoi	<18	50.0 (28.5)	49.3 (30.0)
do_tuoi	18-45	50.9 (29.5)	48.7 (30.4)
do_tuoi	>45	49.8 (29.0)	49.3 (30.7)
gioi	Nam	49.6 (29.2)	49.9 (29.7)
gioi	Nu	50.8 (28.8)	48.4 (31.0)

Những tính năng thuộc về lập trình cho `dplyr` và `purrr` như dấu chấm than kép (`!!`), hàm `rlang::sym()`, hàm `dplyr::all_of()`, và hàm `purrr:map_df()` mình sẽ giới thiệu cụ thể trong một bài khác. Chúng ta sẽ dừng lại bài này ở đây, vì hi vọng bạn đã hiểu rõ hơn cách chúng ta tư duy khi lập trình để thao tác với số liệu. Mình tổng hợp lại kết quả ở dưới đây nhé.

```
library(dplyr)
library(tidyr)
library(rlang)
library(purrr)

# Tính mean (SD) cho một biến
get_mean_sd <- function(d, group_vars, outcome_var) {
    d %>%
        # Bước 1
        select(all_of(c(group_vars, outcome_var))) %>%
        pivot_longer(cols = all_of(group_vars), names_to = "variable") %>%

        # Bước 2
        group_by(variable, value) %>%
        summarize(
            mean = mean(!!sym(outcome_var)),
            sd = sd(!!sym(outcome_var))
```

```
        ) %>%

        # Bước 3
        mutate(
            outcome = outcome_var,
            mean_sd = sprintf("%.1f (%.1f)", mean, sd)
        ) %>%
        select(variable, value, outcome, mean_sd)
}

# Tính mean (SD) cho tất cả các biến
get_mean_sd_all <- function(d, group_vars, outcome_vars) {
    map_df(outcome_vars, ~ get_mean_sd(d, group_vars, .x)) %>%
        pivot_wider(id_cols = c(variable, value),
            names_from = outcome, values_from = mean_sd)
}

group_vars <- c("gioi", "do_tuoi")
outcome_vars <- c("hailong", "qol")
get_mean_sd_all(d, group_vars, outcome_vars)
```

5 Sử dụng hàm trong R để tối ưu hóa công việc

```
library(knitr)
opts_chunk$set(
    message = FALSE,
    warning = FALSE
)
```

5.1 Bài toán

Hôm nay chúng ta sẽ nói về việc sử dụng hàm trong R. Chúng ta sẽ cùng tìm hiểu khi nào nên sử dụng hàm và vai trò của nó trong việc tối ưu hóa việc lập trình.

Chúng ta sẽ có một cơ sở dữ liệu gồm các biến `gioi` (Nam/Nữ), `do_tuoi` (ba nhóm độ tuổi), `hb` (nồng độ hemoglobin), và `rbc` (số lượng hồng cầu).

```
library(dplyr)
library(tidyr)

set.seed(0)
n <- 1000

d <- data.frame(
    stt = seq(n),
    gioi = factor(sample(c(1, 2), n, TRUE),
        levels = c(1, 2), labels = c("Nam", "Nu")),
    do_tuoi = factor(sample(c(1, 2, 3), n, TRUE),
        levels = c(1, 2, 3), labels = c("<18", "18-45", ">45"))
) %>%
    mutate(
        hb = round(rnorm(n, 130, 10) + 5 * (gioi == "Nam") - 3.5 * (do_tuoi == ">45"), 1),
        rbc = round(rnorm(n, 4, 0.5) + 0.25 * (gioi == "Nam") - 0.1 * (do_tuoi == ">45"),
    )
```

```
d %>% head(10) %>% kable()
```

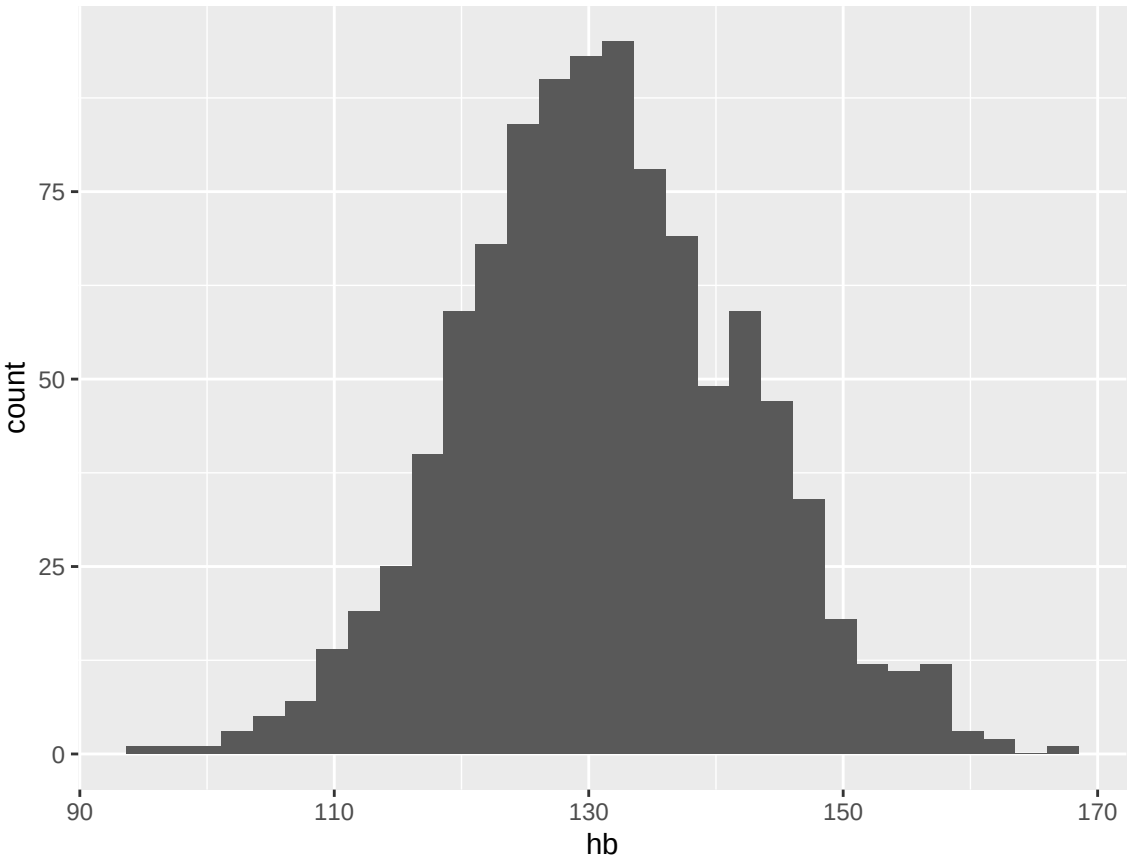
stt	gioi	do_tuoi	hb	rbc
1	Nu	>45	122.1	3.93
2	Nam	>45	127.7	4.95
3	Nu	>45	121.5	3.89
4	Nam	<18	140.0	3.98
5	Nam	>45	146.7	5.14
6	Nu	<18	139.9	3.32
7	Nam	18-45	147.5	4.46
8	Nam	<18	131.7	3.76
9	Nam	18-45	143.4	4.85
10	Nu	18-45	120.2	4.22

Chúng ta sẽ vẽ biểu đồ histogram cho các biến `hb` và `rbc`.

5.2 Vẽ biểu đồ bằng `ggplot2`

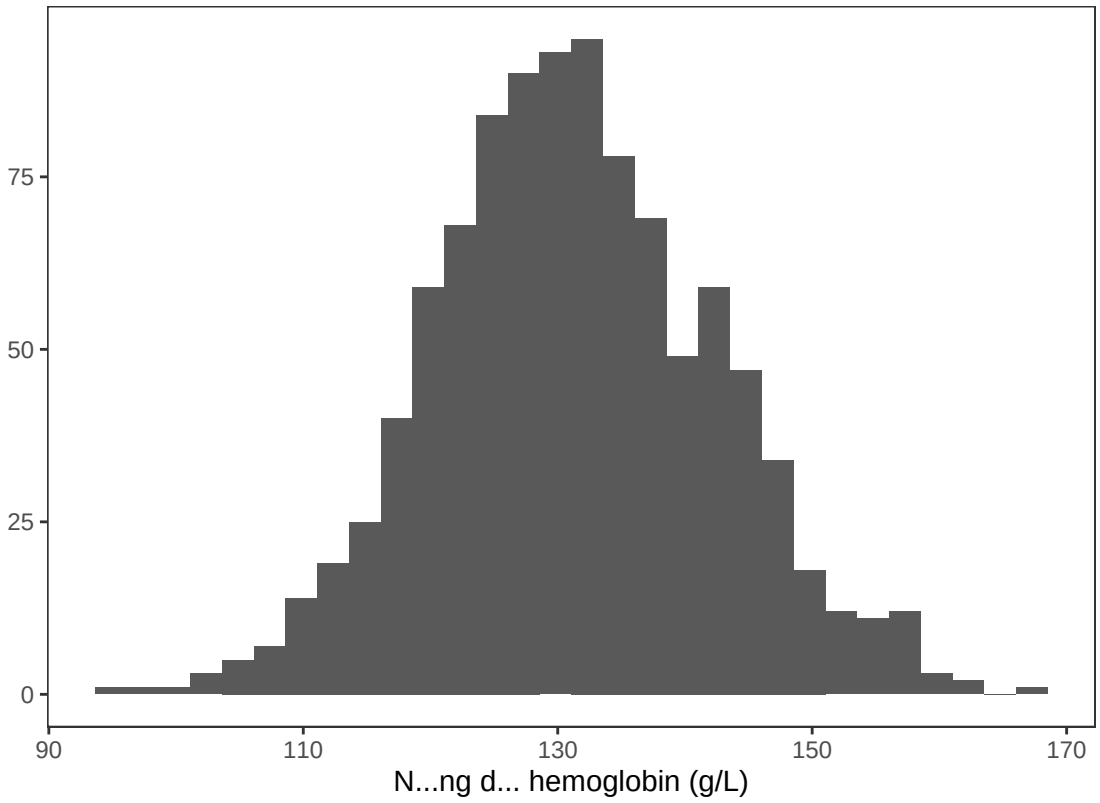
Thư viện `ggplot2` cung cấp cho chúng ta một engine đồ họa mạnh với khả năng tùy biến cao. Hãy cùng nhau vẽ một biểu đồ histogram cho biến `hb` và sau đó tùy biến nó. Đầu tiên, chúng ta sẽ dùng tối thiểu số lệnh cần để vẽ biểu đồ này. Lưu ý: trong bài này mình không giới thiệu tính năng các hàm trong `ggplot2`, bạn sẽ phải tự tìm hiểu ở những khóa học khác.

```
library(ggplot2)

ggplot(d, aes(x = hb)) +
    geom_histogram()
```



Có một số tùy biến mà chúng ta thường sẽ muốn thiết lập để biểu đồ nhìn có thẩm mỹ hơn. Chẳng hạn, mình muốn chuyển sang theme đen-trắng, loại bỏ các đường dóng, đổi tên trục biểu đồ, và thêm ghi chú.

```
ggplot(d, aes(x = hb)) +
    geom_histogram() +
    theme_bw() +
    theme(panel.grid = element_blank()) +
    labs(
        x = "Nồng độ hemoglobin (g/L)",
        y = "",
        caption = "Biểu đồ histogram thể hiện phân bố\ncác giá trị của quần thể nghiên cứu
    )
```

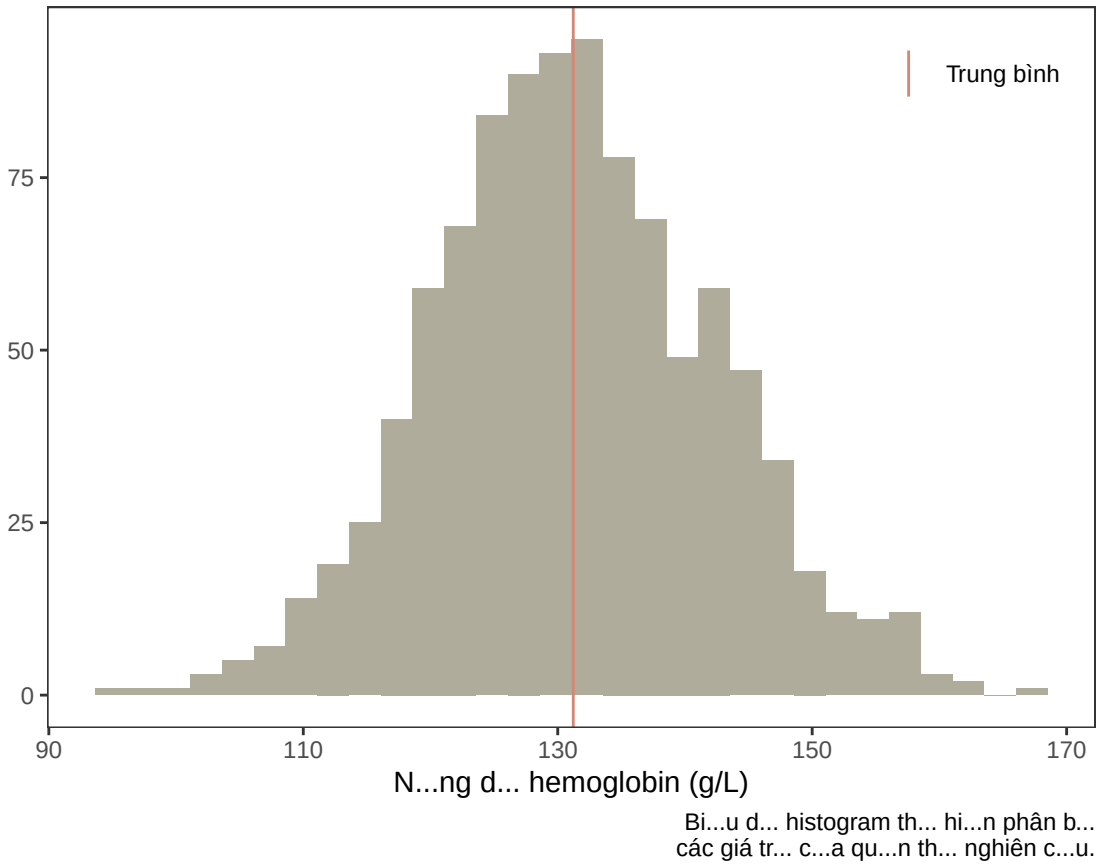


Bi...u d... histogram th... hi...n phân b... các giá tr... c...a qu...n th... nghiên c...u.

Mình chợt nhận ra là mình muốn thêm đường thẳng thể hiện trung bình của nồng độ hemoglobin vào biểu đồ này, và thay đổi màu sắc của histogram.

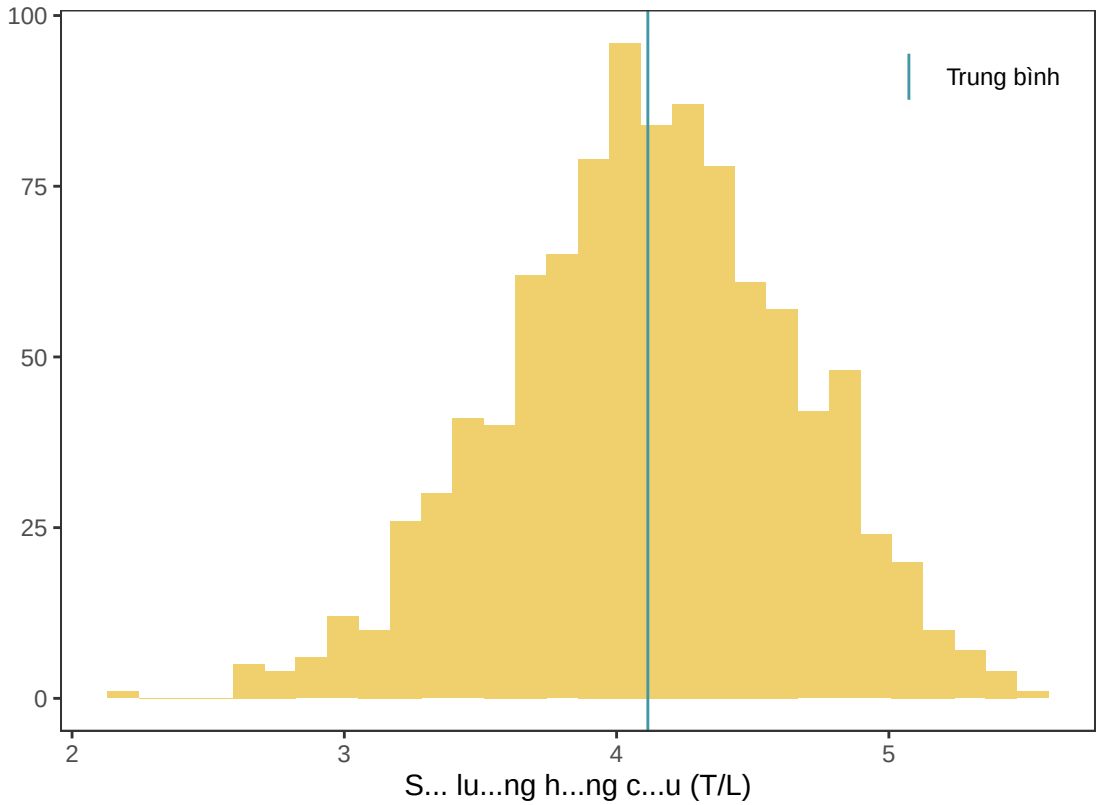
```
ggplot(d, aes(x = hb)) +
    geom_histogram(fill = "#afac9c") +
    geom_vline(
        aes(
            xintercept = d %>% pull(hb) %>% mean(na.rm = TRUE),
            color = "mean"
        )
    ) +
    theme_bw() +
    theme(
        panel.grid = element_blank(),
        legend.position = c(0.98, 0.98),
        legend.justification = c("right", "top"),
```

```
    legend.title = element_blank()
) +
labs(
    x = "Nồng độ hemoglobin (g/L)",
    y = "",
    caption = "Biểu đồ histogram thể hiện phân bố\ncác giá trị của quần thể nghiên cứu
) +
scale_color_manual(
    values = c(mean = "#d28872"),
    labels = c(mean = "Trung bình")
)
```



Bây giờ chúng ta sẽ vẽ tương tự cho biến `rbc`. Bạn nhận ra rằng mình sẽ sao chép lại một đoạn mã lệnh rất dài. Và giả sử mình muốn sử dụng những màu sắc khác cho histogram và đường trung bình, mình sẽ phải chỉnh sửa lại các dòng lệnh liên quan.

```r
ggplot(d, aes(x = rbc)) +
    geom_histogram(fill = "#f0cf6d") +
    geom_vline(
        aes(
            xintercept = d %>% pull(rbc) %>% mean(na.rm = TRUE),
            color = "mean"
        )
    ) +
    theme_bw() +
    theme(
        panel.grid = element_blank(),
        legend.position = c(0.98, 0.98),
        legend.justification = c("right", "top"),
        legend.title = element_blank()
    ) +
    labs(
        x = "Số lượng hồng cầu (T/L)",
        y = "",
        caption = "Biểu đồ histogram thể hiện phân bố\ncác giá trị của quần thể nghiên cứu
    ) +
    scale_color_manual(
        values = c(mean = "#4197a5"),
        labels = c(mean = "Trung bình")
    )
```



5.3 Đóng gói công việc bằng hàm

Chúng ta đã thực hiện các công việc sau đây:

- Khởi tạo một biểu đồ
- Vẽ histogram
- Vẽ đường thẳng trung bình
- Định dạng lại biểu đồ

Bạn có thể thấy rằng tất cả các dòng lệnh cho công việc này nối với nhau bằng toán tử `+`:

```
<công_việc_1> +
    <công_việc_2> +
    <công_việc_3> +
    ...
```

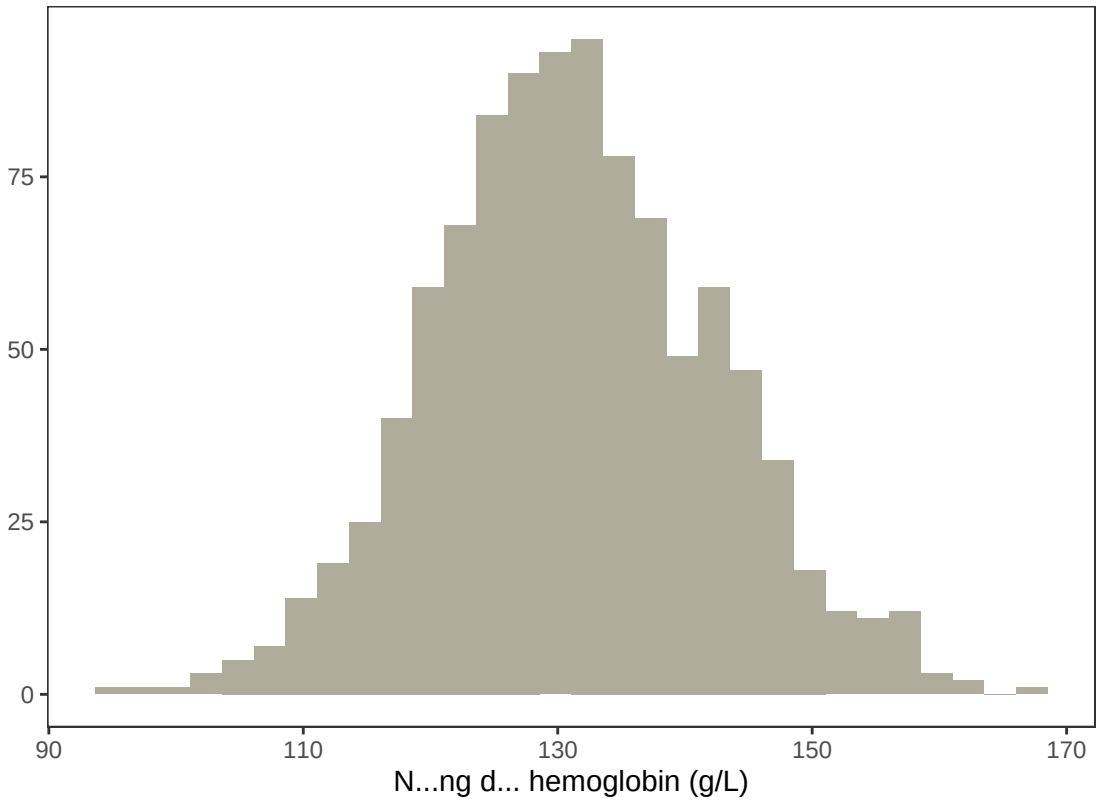
Hãy cùng nhau gói các công việc này vào trong hàm. Để thuận tiện, mình sẽ gói công việc khởi tạo biểu đồ và định dạng chung vào một hàm.

```
library(rlang)

# Công việc 1: Khởi tạo biểu đồ
plot_create <- function(data) {
    ggplot(data) +
        theme_bw() +
        theme(
            panel.grid = element_blank()
        )
}

# Công việc 2: Vẽ histogram
plot_histogram <- function(g, var_name, var_label, bar_fill) {
    g +
        geom_histogram(
            aes(x = !!sym(var_name)),
            fill = bar_fill
        ) +
        labs(
            x = var_label,
            y = "",
            caption = "Biểu đồ histogram thể hiện phân bố\ncác giá trị của quần thể nghiên
        )
}

# Công việc 3: Vẽ đường thẳng trung bình
plot_mean <- function(g, data, var_name, line_color) {
    g +
        geom_vline(
            aes(
                xintercept = data %>% pull(!!sym(var_name)) %>% mean(na.rm = TRUE),
                # Có cách làm khác sử dụng stat_summary(),
                # bạn tự tìm hiểu nhé.
                color = "mean"
            )
        ) +
        scale_color_manual(
            values = c(mean = line_color),
            labels = c(mean = "Trung bình")
```

```
        ) +
        theme(
            legend.position = c(0.98, 0.98),
            legend.justification = c("right", "top"),
            legend.title = element_blank()
        )
}
```

Sau khi đã xây dựng xong các hàm này, chúng ta có thể vẽ như ý muốn. Với biểu đồ cho hemoglobin, mình không muốn vẽ đường thẳng trung bình.

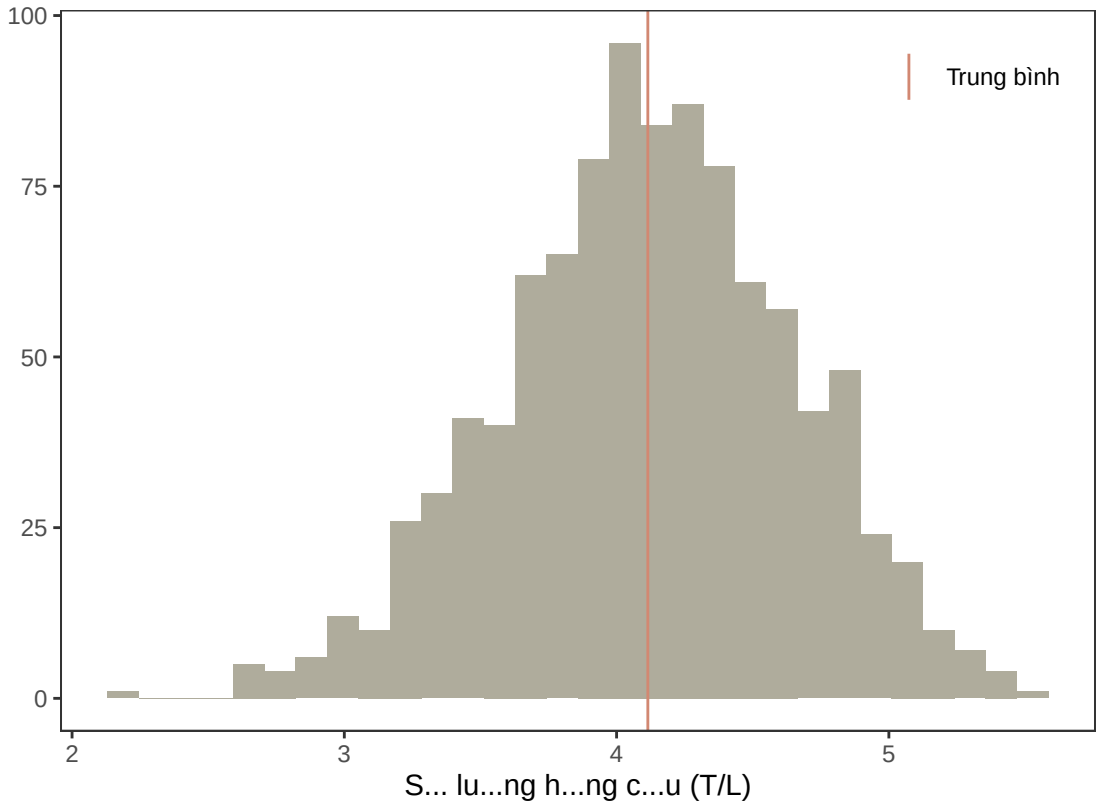
```
var_name <- "hb"
var_label <- "Nồng độ hemoglobin (g/L)"
bar_fill <- "#afac9c"

plot_create(d) %>%
    plot_histogram(var_name, var_label, bar_fill)
```



Bi...u d... histogram th... hi...n phân b... các giá tr... c...a qu...n th... nghiên c...u.

Nhưng với biểu đồ cho hồng cầu, mình sẽ vẽ đường thẳng trung bình, và giữ nguyên thiết lập màu sắc của biểu đồ trước.

```
var_name <- "rbc"
var_label <- "Số lượng hồng cầu (T/L)"
line_color <- "#d28872"

plot_create(d) %>%
    plot_histogram(var_name, var_label, bar_fill) %>%
    plot_mean(d, var_name, line_color)
```



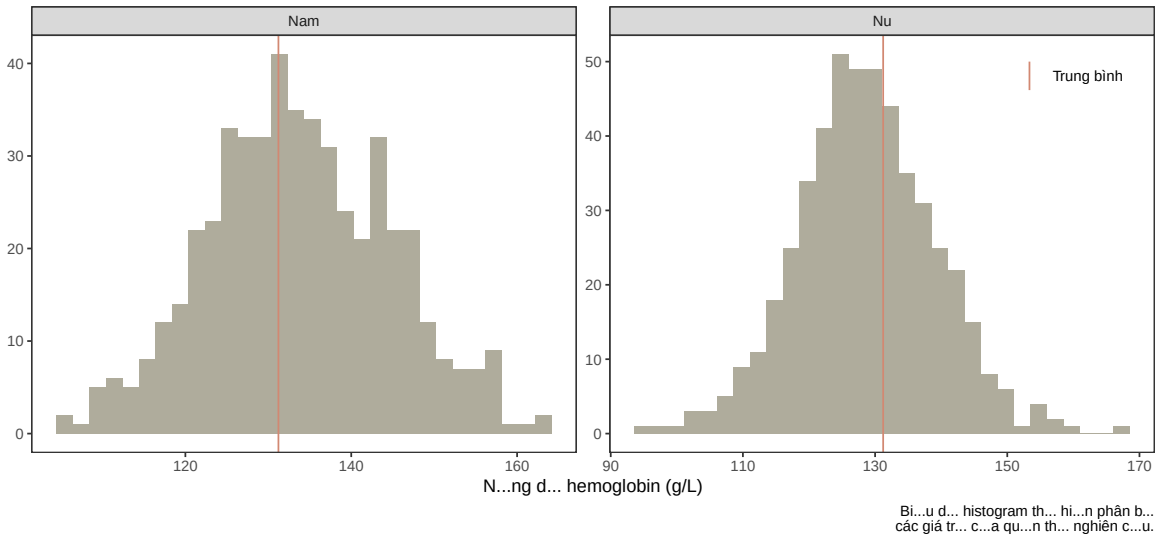
Bi...u d... histogram th... hi...n phân b... các giá tr... c...a qu...n th... nghiên c...u.

5.3.1 Vẽ nhiều biểu đồ cho dưới nhóm

Giả sử bạn muốn vẽ hai biểu đồ histogram của `hb` cho hai nhóm giới tính. Chúng ta dễ dàng được điều này với hàm `facet_wrap()`.

```
plot_subgroup <- function(g, var_subgroup, ncol = 2) {
    g +
        facet_wrap(vars(!!sym(var_subgroup)), ncol = ncol, scales = "free")
}

var_name <- "hb"
var_label <- "Nồng độ hemoglobin (g/L)"
bar_fill <- "#afac9c"
line_color <- "#d28872"
var_subgroup <- "gioi"
```

```
plot_create(d) %>%
    plot_histogram(var_name, var_label, bar_fill) %>%
    plot_mean(d, var_name, line_color) %>%
    plot_subgroup(var_subgroup)
```



5.3.2 Vẽ nhiều biểu đồ cho nhiều biến

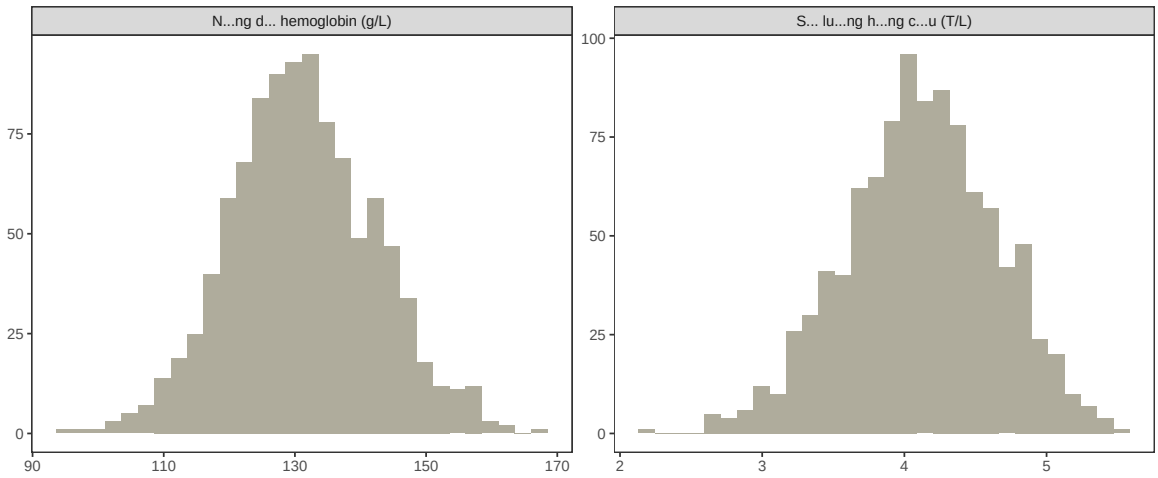
Bạn sẽ tự hỏi liệu có thể làm tương tự nhưng vẽ hai biểu đồ cho hai biến `hb` và `rbc` được không? Câu trả lời là chúng ta sẽ cần chuẩn bị dữ liệu cho việc này bằng `tidyr::pivot_longer()`.

```
prepare_long_data <- function(data, vars_to_long, vars_labels,
    names_to = "variable", values_to = "value") {
    data %>%
        pivot_longer(
            any_of(vars_to_long),
            names_to = names_to, values_to = values_to
        ) %>%
        mutate(
            !!names_to := factor(!!sym(names_to),
                levels = vars_to_long, labels = vars_labels)
        )
}

prepare_long_data(d, c("hb", "rbc"), c("Hb", "RBC")) %>% head() %>% kable()
```

stt	gioi	do_tuoi	variable	value
1	Nu	>45	Hb	122.10
1	Nu	>45	RBC	3.93
2	Nam	>45	Hb	127.70
2	Nam	>45	RBC	4.95
3	Nu	>45	Hb	121.50
3	Nu	>45	RBC	3.89

Với dữ liệu dạng dọc như thế này, chúng ta có thể sử dụng các hàm nêu trên, nhưng đổi tên các biến. Mình bỏ hàm vẽ đường thẳng trung bình vì trong trường hợp này bạn sẽ cần viết lại hàm (sử dụng hàm `ggplot2::stat_summary()`).

```
vars_to_long <- c("hb", "rbc")
vars_labels <- c(
    "Nồng độ hemoglobin (g/L)",
    "Số lượng hồng cầu (T/L)"
)
var_name <- "value"
var_label <- ""
bar_fill <- "#afac9c"
var_subgroup <- "variable"

prepare_long_data(d, vars_to_long, vars_labels) %>%
    plot_create() %>%
    plot_histogram(var_name, var_label, bar_fill) %>%
    plot_subgroup(var_subgroup)
```



Bi...u d... histogram th... hi...n phân b... các giá tr... c...a qu...n th... nghiên c...u.

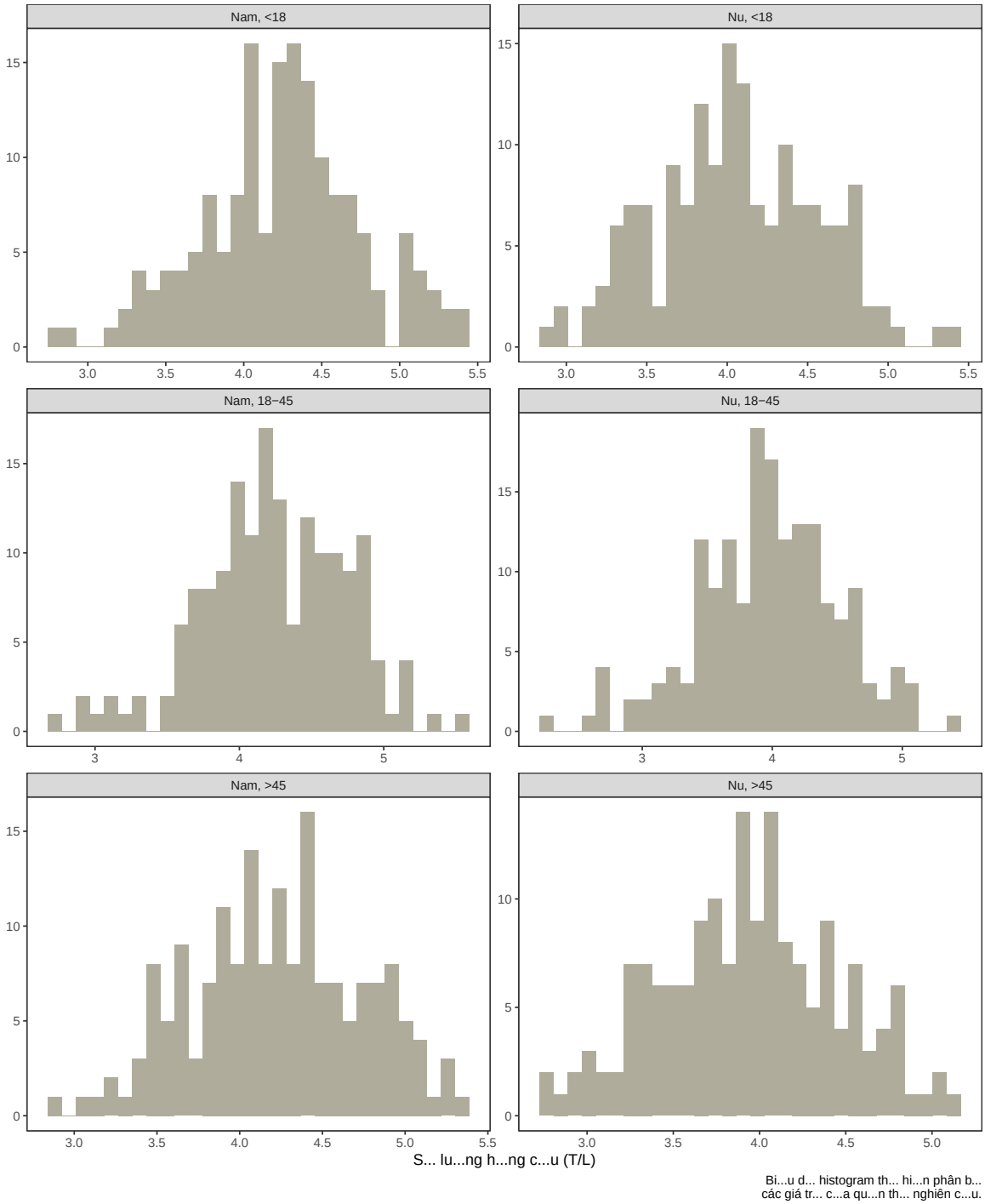
5.3.3 Vẽ nhiều biểu đồ cho nhiều biến dưới nhóm

Trong trường hợp muốn vẽ biểu đồ dưới nhóm cho cả giới và tuổi, chúng ta có thể làm như sau.

```
library(purrr)

prepare_data_multisubgroups <- function(data, vars_subgroup, new_subgroup,
    sep = ", ") {
    paste_list <- lift(paste)
    new_levels <- map(vars_subgroup, ~ levels(pull(d, .x))) %>%
        cross() %>%
        map(~ paste_list(.x, sep = sep)) %>%
        unlist()
    data %>%
        unite(!!new_subgroup, any_of(vars_subgroup), sep = sep) %>%
        mutate(
            !!new_subgroup := factor(!!sym(new_subgroup), levels = new_levels)
        )
}

prepare_data_multisubgroups(d, c("gioi", "do_tuoi"), "gioi_tuoi") %>% head() %>% kable()
```

stt	gioi_tuoi	hb	rbc
1	Nu, >45	122.1	3.93
2	Nam, >45	127.7	4.95
3	Nu, >45	121.5	3.89
4	Nam, <18	140.0	3.98
5	Nam, >45	146.7	5.14
6	Nu, <18	139.9	3.32

```
vars_subgroup <- c("gioi", "do_tuoi")
var_name <- "rbc"
var_label <- "Số lượng hồng cầu (T/L)"
bar_fill <- "#afac9c"
var_subgroup <- "gioi_tuoi"

prepare_data_multisubgroups(d, vars_subgroup, var_subgroup) %>%
    plot_create() %>%
    plot_histogram(var_name, var_label, bar_fill) %>%
    plot_subgroup(var_subgroup)
```



Hi vọng với những ví dụ trên đây, các bạn có thể mường tượng được vai trò của hàm trong việc chia một nhiệm vụ lớn thành nhiều công việc nhỏ. Hàm không chỉ giúp chúng ta viết các

đoạn lệnh gọn gàng hơn và tránh lặp lại các lệnh nhiều lần, nó còn giúp chúng ta tùy biến trong lập trình thông qua việc ghép các công việc khác nhau lại với nhau.