

Nested vs. Splitted Structures in Data Simulation and Inference

Yingqi Jing

July 22, 2025

Contents

| | | |
|-------|--|----|
| 0.1 | 1. Linear regression via nested structures | 3 |
| 0.1.1 | (1) Function | 3 |
| 0.1.2 | (2) Simulation with given parameters | 4 |
| 0.1.3 | (3) Run the linear model | 5 |
| 0.1.4 | (4) Visualization | 7 |
| 0.2 | 2. Linear regression via splitted structures | 9 |
| 0.3 | 3. Concluding remarks | 11 |
| 0.4 | Useful links | 12 |

List of Figures

| | | |
|---|--|----|
| 1 | nest and split operations for a data.frame | 2 |
| 2 | column-wise and row-wise operations in dplyr | 2 |
| 3 | pmap function to each row of a list in purrr package | 3 |
| 4 | Simulated data plot | 5 |
| 5 | Linear model results | 7 |
| 6 | Splitted simulation | 9 |
| 7 | Linear model with split data | 10 |

List of Tables

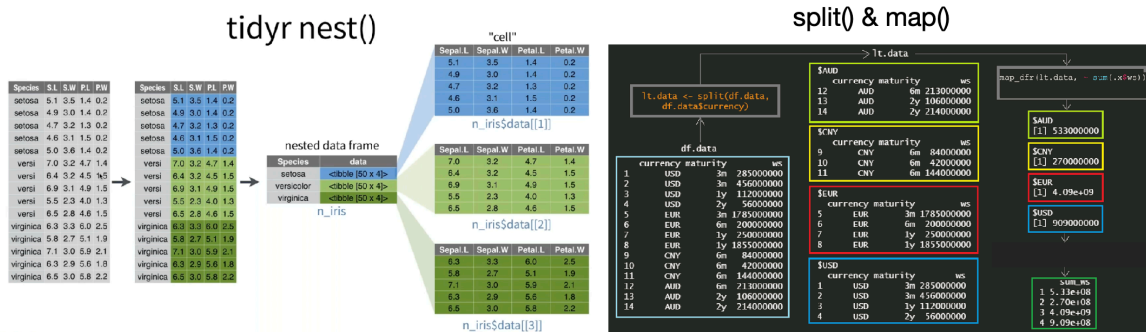


Figure 1: nest and split operations for a data.frame

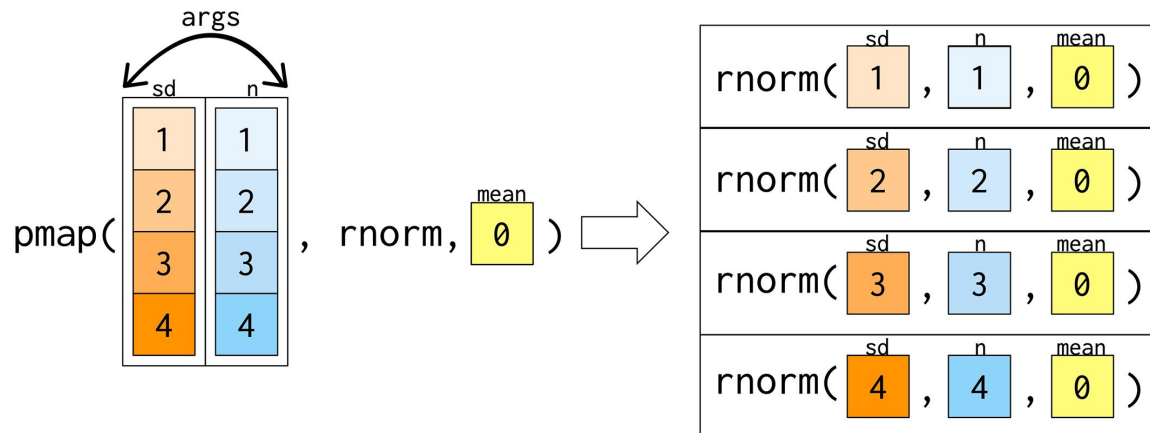
The development of **dplyr** and **purrr** packages makes the workflow of R programming more smooth and flexible. The **dplyr** package provides an elegant way of manipulating data.frames or tibbles in a column-wise (e.g., **select**, **filter**, **mutate**, **arrange**, **group_by**, **summarise** and **case_when**) or row-wise (**rowwise**, **c_across**, and **ungroup**) manner. It also fits well with the **map** function by applying an anonymous function to each column, or applying a user-defined function to each row.



Figure 2: column-wise and row-wise operations in dplyr

The **purrr** package allows us to map a function to each element of a list. You can also **select**, **filter**, **modify**, **combine** and **summarise** a list (see [this blog](#) for an overview). Note that the default output from **map** function is a list of the same length as the input data, though you can easily reformat the output into a data.frame via **map_dfr** and **map_df** functions.

```
args <- list(sd = c(1, 2, 3, 4), n = c(1, 2, 3, 4))
purrr::pmap(args, rnorm, mean = 0)
```



src: @sauer_sebastian × @hadleywickham

Figure 3: pmap function to each row of a list in purrr package

Here we focus on comparing and understanding the `nest` and `split` functions in data simulation and inference. Specifically, two data structures (**nested data** vs. **split data**) are used for simulating and fitting linear regression models across different types.

0.1 1. Linear regression via nested structures

0.1.1 (1) Function

We first define a function to generate the response (y) by providing the predictor (x), *intercept* and *slope*.

```
# function to generate the response
generate_response <- function(x, intercept, slope) {
  x * slope + intercept + rnorm(length(x), 0, 30)
}
```

0.1.2 (2) Simulation with given parameters

To simulate data for each type (A, B or C), we put the parameters in a tibble, since it allows nested objects with a list of vectors as a column. To generate the response, we use a **rowwise** function by applying the **generate_response** function to each row.

```
# it is recommended to use tibble format
parameters <- tibble(
  type = c("A", "B", "C"),
  x = list(1:100, 1:100, 1:100),
  intercept = c(1, 3, 5),
  slope = c(2, 4, 3)
)
# note: convert the vector responses into a list
simulated_df <- parameters %>%
  rowwise() %>%
  mutate(y = list(generate_response(x, intercept, slope))) %>%
  ungroup() %>%
  unnest(c(x, y))
```

```

# It is recommended to use tibble format
parameters <- tibble(
  type = c("A", "B", "C"),
  x = list(1:100, 1:100, 1:100),
  intercept = c(1, 3, 5),
  slope = c(2, 4, 3)
)
# A tibble: 3 × 4
  type     x      intercept slope
  <chr> <list>      <dbl> <dbl>
1 A     <int [100]>         1     2
2 B     <int [100]>         3     4
3 C     <int [100]>         5     3

# rowwise operation to generate response
simulated_df <- parameters %>%
  rowwise() %>%
  mutate(y = list(generate_response(x, intercept, slope))) %>%
  ungroup() %>%
  unnest(c(x, y))
# A tibble: 300 × 5
  type     x intercept slope     y
  <chr> <int>      <dbl> <dbl> <dbl>
1 A         1         1     2 -9.04
2 A         2         1     2 45.5
3 A         3         1     2  9.65

```

Figure 4: Simulated data plot

0.1.3 (3) Run the linear model

With the simulated data.frame or tibble, we can create a nested data and map the linear model for each type. After that, we can extract the predicted values and 95% credible intervals.

```

# nesting data by each type and run lm via map
lm_results <- simulated_df %>%
  group_by(type) %>%

```

```

nest() %>%
mutate(
  models = map(data, ~ lm(y ~ x, data = .x)),
  summaries = map(models, ~ broom::glance(.x)),
  model_coef = map(models, ~ broom::tidy(.x)),
  pred = map(models, ~ predict(.x, interval = "confidence"))
)
# extract the predicted results
pred_ci <- lm_results %>%
  dplyr::select(type, pred) %>%
  unnest(pred) %>%
  pull(pred) %>%
  set_colnames(c("fit", "lwr", "upr"))

```

```

lm_results <- simulated_df %>%
  group_by(type) %>%
  nest() %>%
  # group_nest(type) %>%
  mutate(
    models = map(data, ~ lm(y ~ x, data = .x)),
    summaries = map(models, ~ broom::glance(.x)),
    model_coef = map(models, ~ broom::tidy(.x))
    pred = map(models, ~ predict(.x, interval = "confidence"))
  )
# A tibble: 3 × 6
  type          data models summaries model_coef pred
<chr> <list<tibble[,4]>> <list> <list>      <list>      <list>
1 A      [100 × 4] <lm>    <tibble>  <tibble>    <dbl[...]>
2 B      [100 × 4] <lm>    <tibble>  <tibble>    <dbl[...]>
3 C      [100 × 4] <lm>    <tibble>  <tibble>    <dbl[...]>

# extract the predicted results
pred_ci <- lm_results %>%
  dplyr::select(type, pred) %>%
  unnest(pred) %>%
  pull(pred) %>%
  set_colnames(c("fit", "lwr", "upr"))
      fit      lwr      upr
1 10.60067 -1.8463723 23.04771
2 12.50523  0.2453462 24.76511
3 14.40979  2.3360579 26.48353

```

Figure 5: Linear model results

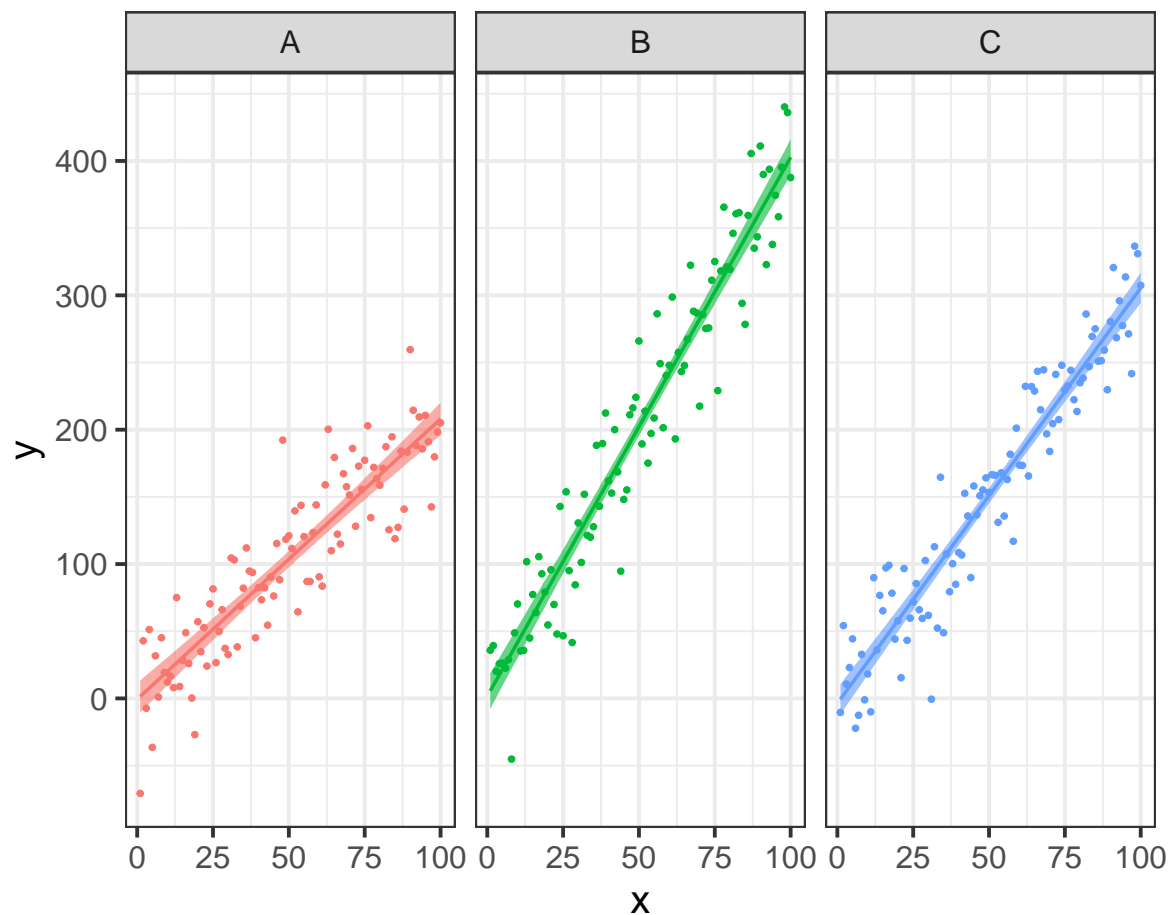
0.1.4 (4) Visualization

To visualize the raw data and the fitted lines, we need to combine them by row, and draw the fitted line and credible intervals via *geom_line* and *geom_ribbon* functions from *ggplot2*.

```

cbind(simulated_df, pred_ci) %>%
  ggplot(., aes(x = x, y = y, color = type)) +
  geom_point() +
  geom_ribbon(aes(ymin = lwr, ymax = upr, fill = type, color = NULL),
    alpha = .6
  ) +
  geom_line(aes(y = fit), size = 1) +
  facet_wrap(~type) +
  theme(legend.position = "none")

```



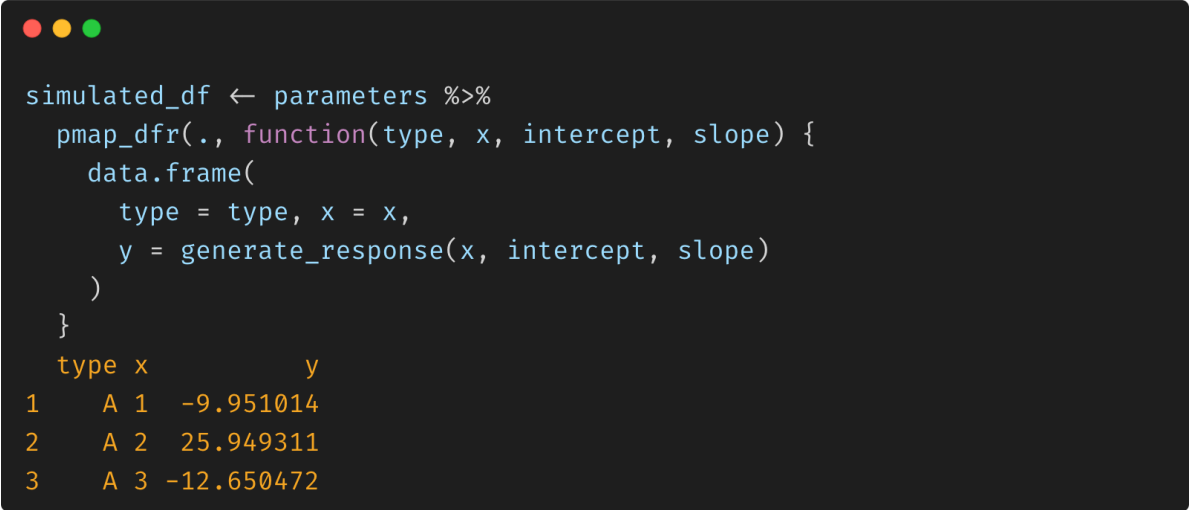
![Fitted results from the linear regression models](pics/lm via dplyr-1.png)

0.2 2. Linear regression via splitted structures

Alternatively, we can split the simulated data by each type and replicate the whole analysis by using `map` functions. Note that you may still need to convert the data into a `data.frame` so as to combine them by row.

```
parameters <- list(  
  type = c("A", "B", "C"),  
  x = list(1:100, 1:100, 1:100),  
  intercept = c(1, 3, 5),  
  slope = c(2, 4, 3)  
)
```

```
simulated_df <- parameters %>%  
  pmap_dfr(., function(type, x, intercept, slope) {  
    data.frame(  
      type = type, x = x,  
      y = generate_response(x, intercept, slope)  
    )  
  })
```



```
simulated_df <- parameters %>%  
  pmap_dfr(., function(type, x, intercept, slope) {  
    data.frame(  
      type = type, x = x,  
      y = generate_response(x, intercept, slope)  
    )  
  })  
type x      y  
1    A 1 -9.951014  
2    A 2 25.949311  
3    A 3 -12.650472
```

Figure 6: Splitted simulation

```
pred_ci <- simulated_df %>%  
  split(.$type) %>%
```

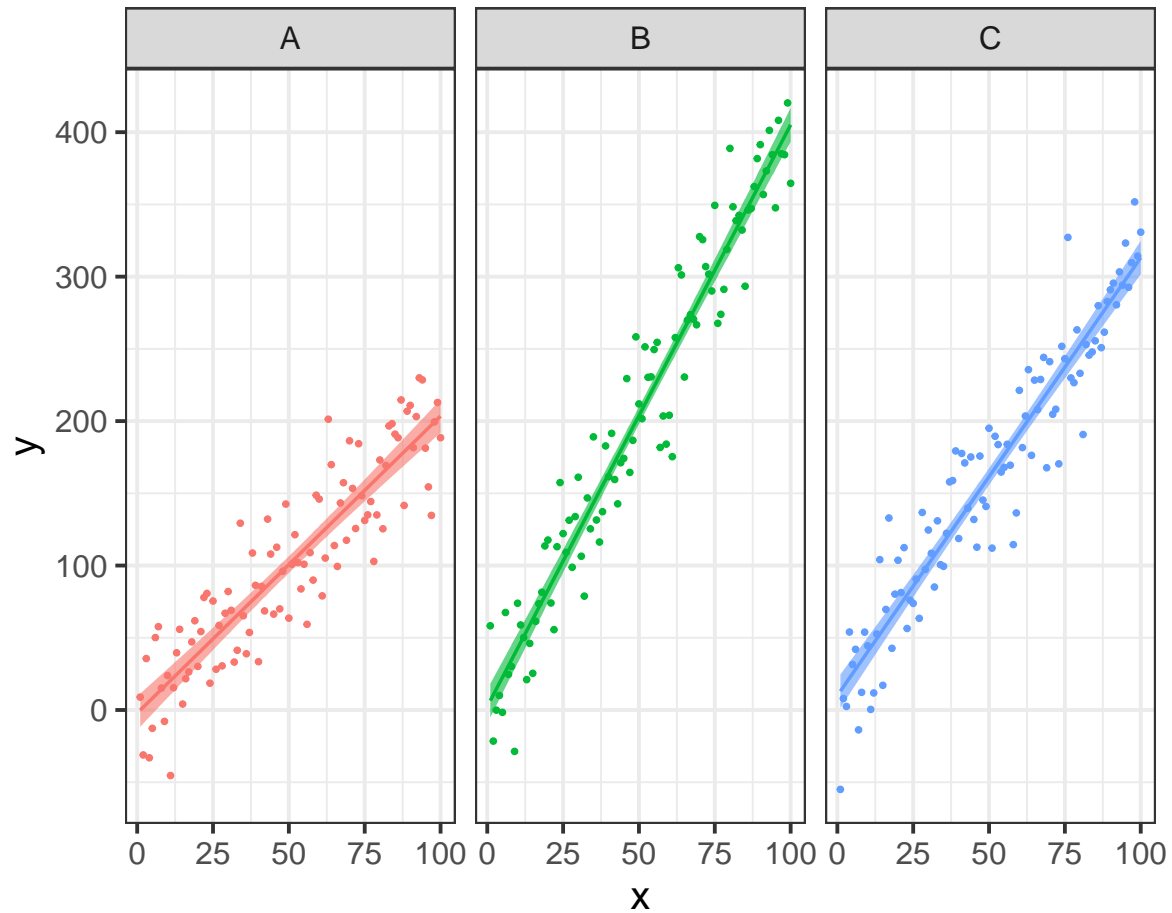
```
map(~ lm(y ~ x, data = .x)) %>%
map_dfr(~ as.data.frame(predict(.x, interval = "confidence")))
```

```
# split the data and run lm
pred_ci <- simulated_df %>%
  split(.$type) %>%
  map(~ lm(y ~ x, data = .x)) %>%
  map_dfr(~ as.data.frame(predict(.x, interval = "confidence")))

      fit      lwr      upr
1  7.073178 -3.9495546 18.09591
2  9.032314 -1.8246799 19.88931
3 10.991449  0.2993031 21.68359
```

Figure 7: Linear model with split data

```
cbind(simulated_df, pred_ci) %>%
  ggplot(., aes(x = x, y = y, color = type)) +
  geom_point() +
  geom_ribbon(aes(ymin = lwr, ymax = upr, fill = type, color = NULL),
    alpha = .6
  ) +
  geom_line(aes(y = fit), size = 1) +
  facet_wrap(~type) +
  theme(legend.position = "none")
```



0.3 3. Concluding remarks

- The combination of `split` and `map` functions seems to be a bit more intuitive and easier to follow. You do not need to constantly `nest` and `unnest` the data. More importantly, nested objects are somehow compressed in tibble objects and make them less tractable.
- Nested data structure is useful when you want to apply several functions to each model object and the results can be directly saved in a tidy data.frame. For the splitted data structure, you need to apply the function to each model object separately.
- Nested structure often relies on lists for data simulation and packing model outputs, whereas the splitted structure needs to convert and combine the outputs into data.frames for visualization.

0.4 Useful links

- <https://towardsdev.com/a-gentle-introduction-to-purrr-4cfe78e92392>
- <https://medium.com/p/da3638b5f46c>
- <https://adv-r.hadley.nz/functionals.html>