Lab 001

Data cleaning and workflow [2/N]

Edward Rubin
15 January 2021

mutate

mutate

Refresher

You already know some of `mutate()`

- **Create** new variables
- **Transform** existing data into more usable forms
- **Fill in** missing values

... but it turns out **there's more!**

mutate

Pipes

But first, a reminder about pipes and the `magrittr` package.

The `%>%` operator is called a **pipe**.

```
# Equivalent:
some_function(a, b)
a %>% some_function(b)
```

The `%<>%` operator is an **assignment pipe**.

```
# Equivalent:
house_df = house_df %>% mutate(n_rooms = bed + bath + other)
house_df %<>% mutate(n_rooms = bed + bath + other)
```

mutate

'Regular' `mutate`

Transform a numeric variable...

```
# Convert kg to lbs: 1 kg is ~2.205 pounds
starwars %<>% mutate(weight = mass * 2.205)
# Check results
starwars %>% select(name, mass, weight)
```

name	mass	weight
Luke Skywalker	77	169.785
C-3PO	75	165.375
R2-D2	32	70.56

`mutate()` receives data and returns both old and new columns.

`transmute()` returns only the new columns.

`mutate` and family

`mutate` and family

Multiple columns

To `mutate()` multiple variables, you'll want one of its cousins:

- `mutate_at()` applies your function(s) to all **specified** columns.
- `mutate_if()` applies your function(s) to **"matched"** columns.
- `mutate_all()` applies your function(s) to **all** columns.

The main difference between the `mutate_*()` functions:

> How do you want to select your columns?

Note: These `mutate_*()` functions will **replace** variables.

`mutate` and family

`_at`

`mutate_at()` wants three arguments:

1. **`.tbl`** the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. **`.vars`** the specific variables you want to transform (*e.g.*, `mass`)
3. **`.funs`** the function(s) you are using for the transformation (*e.g.*, `log`)

For example, to take the `log` of `height` and `mass`

```
# Option 1: Character vector of variable names
starwars %>% mutate_at(.vars = c("height", "mass"), .funs = log)
```

Above: You can specify which variables using a character vector.

Q Where is `.tbl`?

A Implicitly defined as `starwars` via the pipe (`starwars %>%`).

`mutate` and family

`_at`

`mutate_at()` wants three arguments:

1. **`.tbl`** the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. **`.vars`** the specific variables you want to transform (*e.g.*, `mass`)
3. **`.funs`** the function(s) you are using for the transformation (*e.g.*, `log`)

For example, take the `log` of `height` and `mass`.

```r
# Option 1a: Character vector of variable names
starwars %>% mutate_at(.vars = c("height", "mass"), .funs = log)
# Option 1b: Same, but don't name the arguments
starwars %>% mutate_at(c("height", "mass"), log)
```

Above: You don't have to name the arguments `.vars` and `.funs`.

`mutate` and family

`_at`

`mutate_at()` wants three arguments:

1. **`.tbl`** the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. **`.vars`** the specific variables you want to transform (*e.g.*, `mass`)
3. **`.funs`** the function(s) you are using for the transformation (*e.g.*, `log`)

For example, take the `log` of `height` and `mass`.

```
# Option 2a: A "list" via the 'var' function
starwars %>% mutate_at(.vars = vars(height, mass), .funs = log)
```

Above: You can "list" the target variables inside `vars()`.

`mutate` and family

`_at`

`mutate_at()` wants three arguments:

1. **`.tbl`** the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. **`.vars`** the specific variables you want to transform (*e.g.*, `mass`)
3. **`.funs`** the function(s) you are using for the transformation (*e.g.*, `log`)

For example, take the `log` of `height` and `mass`.

```r
# Option 2b: Variable "span" inside 'var'
starwars %>% mutate_at(.vars = vars(height:mass), .funs = log)
```

Above: **Inside** `vars()`, define a "span" of (sequential) variables with `:`.

`mutate` and family

`_at`

`mutate_at()` wants three arguments:

1. `.tbl` the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. `.vars` the specific variables you want to transform (*e.g.*, `mass`)
3. `.funs` the function(s) you are using for the transformation (*e.g.*, `log`)

For example, take the `log` of `height`-related variables.

```
# Option 2c: Matching names inside 'var'.
starwars %>% mutate_at(.vars = vars(starts_with("height")), .funs = log)
```

Above: **Inside** `vars()`, variables whose names start with `"height"`.

"Predicate" functions: `starts_with`, `ends_with`, `contains`, `matches`.

`mutate` and family

`_at`

Q How might we use `mutate_at` to help with missing values?

Ex. C-3PO and R2-D2 (among others) have `NA` for `hair_color`, but let's apply our technique for **all `"_color"` variables**.

We need two things:

1. Which **`.vars`** we will target.

```
.vars = vars(contains("_color"))
```

2. The **function(s)** (`.funs`) for the mutation/transformation.

```
# We'll write our own function but can rely upon 'replace_na'
na_to_other = function(d) replace_na(data = d, replace = "other")
```

`mutate` and family

`_at`

The function `replace_na(data, replace)` comes from `tidyr`.

- `data` can be a vector or data frame
- `replace` is the replacement for those pesky `NA`s

Be careful: Just because you *can* replace `NA`s doesn't mean you *should*.

mutate and family

_at

Let's put it all together now.

```r
# Define the function to replace NAs with "other"
na_to_other = function(d) replace_na(data = d, replace = "other")
# Use mutate_at to transform "_color" variables
starwars %<>% mutate_at(
  .vars = vars(contains("_color")),
  .funs = na_to_other
)
# Check if it worked!
starwars %>% select(name, contains("_color"))
```

name	hair_color	skin_color	eye_color
Luke Skywalker	blond	fair	blue
C-3PO	other	gold	yellow
R2-D2	other	white, blue	red

`mutate` and family

`_at`

Note: We don't actually have to define our own function.

`replace_na()` just wants to know what should `replace` the `NA`s.

```r
# Use mutate_at to transform "_color" variables
starwars %<>% mutate_at(
  .vars = vars(contains("_color")),
  .funs = replace_na,
  replace = "other"
)
```

name	hair_color	skin_color	eye_color
Luke Skywalker	blond	fair	blue
C-3PO	other	gold	yellow
R2-D2	other	white, blue	red

`mutate` and family

`_if`

`mutate_if` is very similar to `mutate_at`.

- `_if` uses logical statements to select columns.
- *Recall:* `_at` uses more direct statements to select columns.

`mutate` and family

`_if`

`mutate_if()` wants three arguments:

1. **`.tbl`** the dataset you're trying to mutate/transform (*e.g.*, `starwars`)
2. **`.predicate`** logical statement to select variables (*e.g.*, `is_character`)
3. **`.funs`** function(s) for the transformation (*e.g.*, `replace_na`)

For example, let's change `NA`s to `"other"` for all `character` variables.

```r
# Replace NAs with "other" for character variables
starwars %>% mutate_if(
  .predicate = is.character,
  .funs = replace_na, replace = "other"
)
```

Above: R finds all `character` variables and runs `replace_na` on them.

`mutate` and family

`_if`

Let's change `NA`s to variables' means for `numeric` variables.

Step 1 Create a function that replaces `NA`s with the mean:

```r
# Function for mean imputation/replacement
mean_replace = function(d) replace_na(d, replace = mean(d, na.rm = T))
```

Notice: We need `na.rm = T` in `mean()` to calculate the mean when `NA`s exist.

Step 2 Use the new function `mean_replace()` with `mutate_if()`:

```r
# Replace NAs with means for numeric variables
starwars %>% mutate_if(
  .predicate = is.numeric,
  .funs = mean_replace
)
```

What if we want to apply a function to **all** of our columns?

`mutate` and family

`_all`

`mutate_all( .tbl , .funs )` does exactly what its name suggests.

You supply the data (`.tbl`).

`mutate_all()` transforms **all** variables in `.tbl` using `.funs` function(s).

Now you just need a function that you want to apply to every variable...

The `dplyr` family

There's more!

Just as `mutate` has `mutate_at`, `mutate_if`, and `mutate_all`,

- `transmute` ➡ `transmute_at`, `transmute_if`, and `transmute_all`
- `select` ➡ `select_at`, `select_if`, and `select_all`
- `filter` ➡ `filter_at`, `filter_if`, and `filter_all`
- `summarize` ➡ `summarize_at`, `summarize_if`, and `summarize_all`
- `group_by` ➡ `group_by_at`, `group_by_if`, and `group_by_all`

Note: `dplyr` is working to replace the "scoped verbs" (`_if`, `_at`, `_all`) with the function `across()`. But it's a bit less clean.

Finishing up

What's next?

Check out the **wine reviews**(https://www.kaggle.com/zynicide/wine-reviews) dataset. It has a lot of opportunities for data cleaning.

Coming soon:

- simulation, resampling, and cross validation
- `tidymodels`, `recipes` (more here), and `parsnip`
- *Related:* tidy modeling with `R`

Table of contents

`mutate`

- Refresher
- Pipes

`mutate_at`

- Intro
- Fundamentals
- Example

`mutate_if`

- Intro
- Fundamentals
- Example

`mutate_all`

- Intro