R for Research

Data Cleaning

Md. Jubayer Hossain

CHIRAL Bangladesh

Founder & Exectutive Director

Last Updated on May 26, 2024

Agenda

- Introduction to Data Cleaning
- Dealing with Missing Data
- Recoding Variables
- Working with Strings
- Separating and Uniting Data

Introduction to Data Cleaning

What is data cleaning?

- In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.
- Data cleaning is an essential part of data analysis, because dirty data can lead to misleading results.

Why data cleaning is so important?

There are several reasons why data cleaning is crucial, especially in health research:

- **Better Decision-Making:** Dirty data, like data with errors or inconsistencies, can lead to misleading results in analysis. Imagine trying to decide where to send flu vaccines based on faulty data - you could end up sending them to the wrong areas, leaving vulnerable populations at risk. Clean data ensures your analysis is accurate, leading to more informed public health decisions.
- **Improved Efficiency:** Dirty data can waste time and resources. For instance, public health officials might chase down inaccurate leads from messy disease outbreak data, delaying the real response. Cleaning data eliminates these roadblocks and allows for a more efficient allocation of resources.

Why data cleaning is so important?

- **Trustworthy Insights:** Public health initiatives rely on public trust. Inaccurate information due to unclean data can erode that trust. Clean data ensures the information being communicated to the public about health risks or vaccination rates is accurate and reliable.
- **Fair and Equitable Outcomes:** Unclean data can lead to biased results in public health analysis. For example, missing data on certain demographics could skew results and lead to unequal distribution of resources. Data cleaning helps ensure everyone is considered fairly in public health efforts.

Dealing with Missing Data

Missing data types

One of the most important aspects of data cleaning is missing values.

Types of “missing” data:

- `NA` - general missing data
- `NaN` - stands for “**N**ot **a** **N**umber”, happens when you do `0/0`.
- `Inf` and `-Inf` - Infinity, happens when you divide a positive number (or negative number) by `0`.

Finding Missing data

- `is.na` - looks for `NAN` and `NA`
- `is.nan`- looks for `NAN`
- `is.infinite` - looks for Inf or -Inf

```
test <- c(0,NA, -1)
test/0
test <- test/0
is.na(test)
is.nan(test)
is.infinite(test)
```

Useful checking functions

- `any` will be `TRUE` if ANY are true
  - `any(is.na(x))`
  - do we have any `NA`'s in `x`?

```
1 test
2 any(is.na(test))
```

Finding NA values with `count()`

Check the values for your variables, are they what you expect?

`count()` is a great option because it gives you:

1. The unique values
2. The amount of these values

```
1 starwars |> count(sex)
```

naniar

Sometimes you need to look at lots of data though... the naniar package is a good option.

```
# install.packages("naniar")
library(naniar)
```

naniar: `pct_complete()`

The `pct_complete()` function shows the percentage that is complete for a given data object.

```
1 pct_complete(starwars)
```

Or for a particular variable:

```
1 starwars |>
2   select(gender) |>
3   pct_complete()
```

naniar::miss_var_summary()

To get the percent missing (and counts) for each variable as a table, use this function.

```
1 miss_var_summary(starwars)
```

naniar plots

The `gg_miss_var()` function creates a nice plot about the number of missing values for each variable (need a data frame).

```
1 gg_miss_var(starwars)
```

Missing Data Issues

- Recall that mathematical operations with `NA` often result in `NA`s.
- This is also true for logical data. Recall that `TRUE` is evaluated as 1 and `FALSE` is evaluated as 0.

```
sum(c(1,2,3,NA))
mean(c(1,2,3,NA))
median(c(1,2,3,NA))
```

filter() and missing data

Be careful with missing data using subsetting:

`filter()` **removes missing values by default.** Because R can't tell for sure if an `NA` value meets the condition. To keep them need to add `is.na()` conditional.

Think about if this is OK or not - it depends on your data!

filter() and missing data

What if NA values represent values that are so low it is undetectable?

Filter will drop them from the data.

```
starwars |>
  filter(gender == "feminine")
```

filter() and missing data

is.na() can help us keep them.

```
1 starwars |>
2   filter(gender == "feminine" | is.na(gender))
```

To remove rows with NA values for a variable use `drop_na()`

A function from the `tidyr` package.

Disclaimer: Don't do this unless you have thought about if dropping NA values makes sense based on knowing what these values mean in your data. **Also consider if you need those rows for values for other variables.**

```
1 dim(starwars)
2 starwars_drop <- starwars |> drop_na(gender)
3 dim(starwars_drop)
```

Let's take a look

Can still have NAs for other columns

```
1 starwars_drop
```

To remove rows with NA values for a data frame use drop_na()

This function of the tidyr package drops rows with **any** missing data in **any** column when used on a df.

```
1 starwars_drop <- starwars |> drop_na()
2 starwars_drop
```

Drop columns with any missing values

Use the `miss_var_which()` function from `naniar`

```
1 miss_var_which(starwars) # which columns have missing values
```

Drop columns with any missing values

`miss_var_which` and function from `naniar` (need a data frame)

```
1 starwars_drop <- starwars |> select(!miss_var_which(starwars))
2 starwars_drop
```

Change a value to be NA

Let's say we think that all 0 values should be NA.

```
count(starwars, gender)
```

Change a value to be NA

The `na_if()` function of `dplyr` can be helpful for this. Let's say we think that all `0` values should be `NA`.

```
starwars <- starwars |>
  mutate(gender = na_if(gender, 99))

count(starwars, gender)
```

Change NA to be a value

- Sometimes removing NA values leads to distorted math - be careful!
- Think about what your NA means for your data (are you sure ?).
- Is an NA for values so low they could not be reported?
- Or is it if it was too low and also if there was a different issue (like no one reported)?

Think about NA

- If it is something more like a zero then you might want it included in your data like a zero instead of an NA.
- Example: - survey reports 0 if student has tried cigarettes but did not smoke that week
- You might want to keep the NA values so that you know the original sample size.

Word of caution

Calculating percentages will give you a different result depending on your choice to include NA values.

Word of caution - Percentages with NA

```
count(starwars, gender)  |>  mutate(percent = (n/(sum(n)) *100))
```

Word of caution - Percentages with NA

```
starwars  |>
  drop_na(gender) |>
  count(gender)  |>
  mutate(percent = (n/(sum(n)) *100))
```

Should you be dividing by the total count with NA values included?

It depends on your data and what NA might mean.

Pay attention to your data and your NA values!

Don’t forget about the common issues

- Extra or Missing commas
- Extra or Missing parentheses
- Case sensitivity
- Spelling

Summary

- `is.na()`,`any(is.na())`, `count()`, and functions from `naniar` like `gg_miss_var()` can help determine if we have `NA` values
- `filter()` automatically removes `NA` values - can't confirm or deny if condition is met (need `| is.na()` to keep them)
- `drop_na()` can help you remove `NA` values from a variable or an entire data frame
- `NA` values can change your calculation results
- think about what `NA` values represent

Recoding Variables

Example of Recoding

```r
set.seed(124)
data_diet <- tibble(Diet = rep(c("A", "B", "B"),
                               times = 4),
                    Treatment = c("Ginger",
                                  "Ginger",
                                  "Other",
                                  "peppermint",
                                  "peppermint",
                                  "Ginger",
                                  "Mint",
                                  "O",
                                  "Ginger",
                                  "mint",
                                  "Mint",
                                  "O"),
                    Weight_start = sample(100:250, size = 12),
                    Weight_change = sample(-10:20, size = 12))
data_diet
```

Reading in the data if it were an excel sheet

```
1 library(readxl)
2 data_diet<- read_excel(here::here("data", "cleaning_diet_data.xlsx"))
```

Count

This needs lots of recoding.

```
1 data_diet |>
2   count(Treatment)
```

dplyr can help!

Using Excel to find all of the different ways `Treatment` has been coded,could be hectic! In `dplyr` you can use the `case_when` function.

Or you can use `case_when()`

The `case_when()` function of `dplyr` can help us to do this as well.

It is more flexible and powerful.

```
data_diet |>
  mutate(Treatment_recoded = case_when(
                                  Treatment == "O" ~ "Other",
                                  Treatment == "Mint" ~ "Peppermint",
                                  Treatment == "mint" ~ "Peppermint",
                                  Treatment == "peppermint" ~ "Peppermint"))
  count(Treatment, Treatment_recoded)
```

What happened?

We seem to have `NA` values!

We didn't specify what happens to values that were already `Other` or `Ginger`.

```
data_diet %>%
  mutate(Treatment = case_when(
                                  Treatment == "O" ~ "Other",
                                  Treatment == "Mint" ~ "Peppermint",
                                  Treatment == "mint" ~ "Peppermint",
                                  Treatment == "peppermint" ~ "Peppermint"))
```

`case_when()` drops unspecified values

Note that automatically values not reassigned explicitly by `case_when()` will be `NA` unless otherwise specified.

```
# General Format - this is not code!
{data_input}  |>
  mutate({variable_to_fix} = case_when({Variable_fixing}
              /some condition/ ~ {value_for_con},
                           TRUE ~ {value_for_not_meeting_condition})
```

case_when with TRUE ~ original variable name

```
data_diet |>
  mutate(Treatment_recoded = case_when(
                                  Treatment == "O" ~ "Other",
                                  Treatment == "Mint" ~ "Peppermint",
                                  Treatment == "mint" ~ "Peppermint",
                                  Treatment == "peppermint" ~ "Peppermint",
                                   TRUE ~ Treatment))  |>
  count(Treatment, Treatment_recoded)
```

Typically it is good practice to include the TRUE statement

You never know if you might be missing something - and if a value already was an NA it will stay that way.

```
data_diet  |>
  mutate(Treatment_recoded = case_when(
                                  Treatment == "O" ~ "Other",
                                  Treatment == "Mint" ~ "Peppermint",
                                  Treatment == "mint" ~ "Peppermint",
                                  Treatment == "peppermint" ~ "Peppermint",
                                   TRUE ~ Treatment))  |>
  count(Treatment, Treatment_recoded)
```

But maybe we want NA?

Perhaps we want values that are O or Other to actually be NA, then `case_when` can be helpful for this. We simply specify everything else.

```
data_diet |>
  mutate(Treatment_recoded = case_when(Treatment == "Ginger" ~ "Ginger",
                                       Treatment == "Mint" ~ "Peppermint",
                                       Treatment == "mint" ~ "Peppermint",
                                       Treatment == "peppermint" ~ "Peppermint"))
  count(Treatment, Treatment_recoded)
```

case_when() can also overwrite/update a variable

You need to specify what we want in the first part of `mutate`.

```
data_diet |>
  mutate(Treatment = case_when(Treatment == "Ginger" ~ "Ginger",
                               Treatment == "Mint" ~ "Peppermint",
                               Treatment == "mint" ~ "Peppermint",
                               Treatment == "peppermint" ~ "Peppermint"))
  count(Treatment)
```

More complicated case_when()

`case_when` can do complicated statements and can match many patterns at a time.

```
data_diet |>
  mutate(Treatment_recoded = case_when(
    Treatment == "Ginger" ~ "Ginger", # keep it the same!
    Treatment %in% c("Mint", "mint", "Peppermint", "peppermint") ~ "Pepperm
    Treatment %in% c("O", "Other") ~ "Other"))  |>

  count(Treatment, Treatment_recoded)
```

Another reason for `case_when()`

`case_when` can do very sophisticated comparisons!

Here we create a new variable called `Effect`.

```
1 data_diet <- data_diet |>
2       mutate(Effect = case_when(Weight_change > 0 ~ "Increase",
3                                 Weight_change == 0 ~ "Same",
4                                 Weight_change < 0 ~ "Decrease"))
5 head(data_diet)
```

Now it is easier to see what is happening

```
data_diet |>
  count(Diet, Effect)
```

Working with Strings

Strings in R

- R can do much more than find exact matches for a whole string!

The `stringr` package

The `stringr` package:

- Modifying or finding **part** or all of a character string
- We will not cover `grep` or `gsub` - base R functions
  - are used on forums for answers
- Almost all functions start with `str_*`

stringr

`str_detect`, and `str_replace` search for matches to argument pattern within each element of a **character vector** (not data frame or tibble!).

- `str_detect` - returns `TRUE` if `pattern` is found
- `str_replace` - replaces `pattern` with `replacement`

str_detect()

The `string` argument specifies what to check

The `pattern` argument specifies what to check for (case sensitive)

```
1 Effect <- pull(data_diet)  |>  head(n = 6)
2 Effect
3
4 str_detect(string = Effect, pattern = "d")
5 str_detect(string = Effect, pattern = "D")
```

`st_replace()` only replaces the first instance of the pattern in each value

`str_replace_all()` can be used to replace all instances within each value

```
1 str_replace(string = Effect, pattern = "e", replacement = "E")
```

```
1 str_replace_all(string = Effect, pattern = "e", replacement = "E")
```

Subsetting part of a string

- `str_sub()` allows you to subset part of a string
- The `string` argument specifies what strings to work with
- The `start` argument specifies position of where to start
- The `end` argument specifies position of where to end

```
1 str_sub(string = Effect, start = 1, end = 3)
```

filter and stringr functions

```
1 head(data_diet, n = 4)
2 data_diet |>
3   filter(str_detect(string = Treatment,
4                     pattern = "int"))
```

OK back to our original problem

```
1 count(data_diet, Treatment)
```

`case_when()` made an improvement

But we still might miss a strange value

```
data_diet |>
  mutate(Treatment_recoded = case_when(
    Treatment %in% c("G", "g", "Ginger", "ginger") ~ "Ginger",
    Treatment %in% c("Mint", "mint", "Peppermint", "peppermint") ~ "Pepperm
    Treatment %in% c("O", "Other") ~ "Other",
    TRUE ~ Treatment))
```

`case_when()` improved with `stringr`

^ indicates the beginning of a character string $ indicates the end

```
data_diet |>
  mutate(Treatment_recoded = case_when(
    str_detect(string = Treatment, pattern = "int") ~ "Peppermint",
    str_detect(string = Treatment, pattern = "^o|^O") ~ "Other",
    TRUE ~ Treatment)) %>%
  count(Treatment, Treatment_recoded)
```

This is a more robust solution! It will catch typos as long as first letter is correct or there is part of the word mint.

Summary

- `case_when()` requires `mutate()` when working with dataframes/tibbles
- `case_when()` can recode **entire values** based on **conditions** (need quotes for conditions and new values)
  - remember `case_when()` needs `TRUE ~ varaible` to keep values that aren't specified by conditions, otherwise will be `NA`

Note: you might see the `recode()` function, it only does some of what `case_when()` can do, so we skipped it, but it is in the extra slides at the end.

Separating and Uniting Data

Uniting columns

- The `unite()` function can help combine columns
- The `col` argument specifies new column name
- The `sep` argument specifies what separator to use when combining - default is "_"
- The `remove` argument specifies if you want to drop the old columns

```
1 diet_comb <- data_diet |>
2   unite(Diet, Effect, col = "change", remove = TRUE)
```

Separating columns based on a separator

- The `separate()` function from `tidyr` can split a column into multiple columns.
- The `col` argument specifies what column to work with
- The `into` argument specifies names of new columns
- The `sep` argument specifies what to separate by

```
diet_comb <- diet_comb |>
  separate(col = change, into = c("Diet", "Change"), sep = "_" )
diet_comb
```

Summary

- `stringr` package has great functions for looking for specific **parts of values** especially `filter()` and `str_detect()` combined
- `stringr` also has other useful string functions like `str_detect()` (finding patterns in a column or vector), `str_subset()` (parsing text), `str_replace()` (replacing the first instance in values), `str_replace_all()` (replacing all instances in each value) and **more**!
- `separate()` can split columns into additional columns
- `unite()` can combine columns
- `:` can indicate when you want to start and end with columns next to one another