Data Cleaning Pipeline 2: Fixing Inconsistencies

Introduction

In the previous lesson, we learned a range of functions for diagnosing data issues. Now, let's focus on some common techniques and functions for fixing those issues. Let's get started!

Learning Objectives

By the end of this lesson, you will be able to:

- Understand how to clean column names, both automatically and manually.
- Remove empty columns and rows with ease.
- Effectively eliminate duplicate entries.
- Correct and fix string values in your data.
- Convert and modify data types as required.

Packages

Load the following packages for this lesson:

Dataset

The dataset we will be using for this lesson is a slightly modified version of the dataset we used in the first `Data Cleaning` lesson; here, we've added slightly more errors to clean! Check out lesson 1 for an explanation of this dataset.

```
non_adherence <- read_csv(here("data/non_adherence_messy.csv"))
```

```
non_adherence
```

```
## # A tibble: 5 × 15
##   patient_id District `Health unit` Sex   Age_35
##        <dbl>    <dbl>         <dbl> <chr> <chr>
## 1      10037        1             1 Male  over 35
## 2      10537        1             1 F     over 35
## 3       5489        2             3 F     Under 35
## 4       5523        2             3 Male  Under 35
## 5       4942        2             3 F     over 35
## # ℹ 10 more variables: `Age at ART initiation` <dbl>,
## #   EDUCATION_OF_PATIENT <chr>, …
```

Cleaning column names

As a general rule, column names should have a "clean", standardized syntax so that we can easily work with them and so that our code is readable to other coders.

Ideally, column names:

- should be short
- should have no spaces or periods(space and periods should be replaced by underscore "_")
- should have no unusual characters(&, #, <, >)
- should have a similar style

To check out our column names, we can use the `names()` function from base R.

```
names(non_adherence)
```

```
##  [1] "patient_id"                  "District"
##  [3] "Health unit"                 "Sex"
##  [5] "Age_35"                      "Age at ART initiation"
##  [7] "EDUCATION_OF_PATIENT"        "OCCUPATION_OF_PATIENT"
##  [9] "Civil...status"              "WHO status at ART initiaiton"
## [11] "BMI_Initiation_Art"          "CD4_Initiation_Art"
## [13] "regimen.1"                       "Nr_of_pills_day"
## [15] "NA"
```

Here we can see that:

- some names contain spaces
- some names contain special characters such as `...`
- some names are in upper case while some are not

Automatic column name cleaning with `janitor::clean_names()`

A handy function for standardizing column names is the `clean_names()` from the `janitor` {janitor} package. The function `clean_names()` converts all names to consist of only underscores, numbers, and letters, using the snake case style.

```
non_adherence %>%
  clean_names() %>%
  names()
```

```
##  [1] "patient_id"                      "district"
##  [3] "health_unit"                     "sex"
##  [5] "age_35"                          "age_at_art_initiation"
##  [7] "education_of_patient"            "occupation_of_patient"
##  [9] "civil_status"                    "who_status_at_art_initiaiton"
## [11] "bmi_initiation_art"              "cd4_initiation_art"
## [13] "regimen_1"                       "nr_of_pills_day"
## [15] "na"
```

From this output, we can see that:

- upper case variable names were converted to lower case (e.g., `EDUCATION_OF_PATIENT` is now `education_of_patient`)

- spaces inside the variable names have been converted to underscores (e.g., `Age at ART initiation` is now `age_at_art_initiation`)

- periods(`.`) have all been replaced by underscores (e.g., `Civil...status` is now `civil_status`)

Let's save this cleaned dataset as `non_adherence_clean`.

```
non_adherence_clean <-
  non_adherence %>%
  clean_names()
non_adherence_clean
```

```
## # A tibble: 1,420 × 15
##    patient_id district health_unit sex   age_35
##          <dbl>    <dbl>       <dbl> <chr> <chr>
##   1      10037        1           1 Male  over 35
##   2      10537        1           1 F     over 35
##   3       5489        2           3 F     Under 35
##   4       5523        2           3 Male  Under 35
##   5       4942        2           3 F     over 35
##   6       4742        2           3 Male  over 35
##   7      10879        1           1 Male  over 35
##   8       2885        2           3 Male  over 35
##   9       4861        2           3 F     over 35
##  10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: age_at_art_initiation <dbl>, …
```

Q: Automatic cleaning

The following dataset has been adapted from a study that used retrospective data to characterize the tmporal and spatial dynamics of typhoid fever epidemics in Kasene, Uganda.

```
typhoid <- read_csv(here("data/typhoid_uganda.csv"))
```

```
## Rows: 215 Columns: 31
## ── Column specification
────────────────────────────────────────────────────────
## Delimiter: ","
## chr (18): Householdmembers, Positioninthehousehold,
Watersourcedwithinhouseh...
## dbl (11): UniqueKey, CaseorControl, Age, Sex, Levelofeducation,
Below10years...
## lgl  (2): NA, NAN
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

Use the `clean_names()` function from `janitor` to clean the variables names in the `typhoid` dataset.

{stringr} and `dplyr::rename_with()` for renaming columns

To make more manual changes to column names, you should already know how to use the `rename()` function from the `dplyr` package. However, this function requires you to specify the old and new column names. This can be tedious if you have many columns to rename.

An alternative is to use the `rename_with()` function from the `dplyr` package. This function allows you to apply a function to all column names. For example, if we want to convert all column names to upper case, we can use the `toupper()` function inside `rename_with()`.

```
non_adherence_clean %>%
  rename_with(.cols = everything(), .fn = toupper)
```

```
## # A tibble: 1,420 × 15
##    PATIENT_ID DISTRICT HEALTH_UNIT SEX   AGE_35
##         <dbl>    <dbl>       <dbl> <chr> <chr>
##  1      10037        1           1 Male  over 35
##  2      10537        1           1 F     over 35
##  3       5489        2           3 F     Under 35
##  4       5523        2           3 Male  Under 35
##  5       4942        2           3 F     over 35
##  6       4742        2           3 Male  over 35
##  7      10879        1           1 Male  over 35
##  8       2885        2           3 Male  over 35
##  9       4861        2           3 F     over 35
## 10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: AGE_AT_ART_INITIATION <dbl>, …
```

```
# We can also omit the .cols argument, which defaults to everything()
non_adherence_clean %>%
  rename_with(.fn = toupper)
```

```
## # A tibble: 1,420 × 15
##    PATIENT_ID DISTRICT HEALTH_UNIT SEX   AGE_35
##         <dbl>    <dbl>       <dbl> <chr> <chr>
##  1      10037        1           1 Male  over 35
##  2      10537        1           1 F     over 35
##  3       5489        2           3 F     Under 35
##  4       5523        2           3 Male  Under 35
##  5       4942        2           3 F     over 35
##  6       4742        2           3 Male  over 35
##  7      10879        1           1 Male  over 35
##  8       2885        2           3 Male  over 35
##  9       4861        2           3 F     over 35
## 10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: AGE_AT_ART_INITIATION <dbl>, …
```

But of course, this is not a recommended name style. Let's try something more useful. In the `non_adherence` dataset, some of the column names end with `_of_patient`. This is not necessary, as we know that all the variables are of the patient. We can use the `str_replace_all()` function from the `stringr` package to remove the `_of_patient` from all column names.

Here is the syntax for `str_replace_all()`:

```r
test_string <- "this is a test string"
str_replace_all(string = test_string, pattern = "test", replacement = "new")
```

```
## [1] "this is a new string"
```

It has three arguments, `string`, the string to be modified, `pattern`, the pattern to be replaced, and `replacement`, the new pattern to replace the old pattern.

Let's apply this to our column names.

```r
non_adherence_clean_2 <-
  non_adherence_clean %>%
  # manually re-name columns
  rename_with(str_replace_all, pattern = "_of_patient", replacement = "")

non_adherence_clean_2
```

```
## # A tibble: 5 × 15
##   patient_id district health_unit sex   age_35
##        <dbl>    <dbl>       <dbl> <chr> <chr>
## 1      10037        1           1 Male  over 35
## 2      10537        1           1 F     over 35
## 3       5489        2           3 F     Under 35
## 4       5523        2           3 Male  Under 35
## 5       4942        2           3 F     over 35
## # ℹ 10 more variables: age_at_art_initiation <dbl>,
## #   education <chr>, occupation <chr>, …
```

That looks great! We'll use this dataset with the clean columns in the next section.

SIDE NOTE

In the cleaning examples in this lesson, we are assigning a lof of intermediate objects, such as `non_adherence_clean` and `non_adherence_clean_2`. This is not usually necessary; we are doing it here for the sake of clarity in the tutorial. In your own code, you will typically combine multiple cleaning steps into a single pipe chain:

```r
non_adherence_clean <-
  non_adherence %>%
  # cleaning step 1 %>%
  # cleaning step 2 %>%
  # cleaning step 3 %>%
  # and so on
```

Q: Complete cleaning of column names

Standardize the column names in the typhoid dataset with clean_names() then;

- replace or_ with _
- replace of with _

Removing Empty Rows and Columns in R

Empty rows and columns in a dataset, filled entirely with NA values, can be misleading and affect your analysis. It's crucial to identify and remove these to ensure that every row represents a valid data point and every column represents a meaningful variable. Here's a straightforward guide to doing this in R.

How to Remove Empty Columns

First, let's focus on columns. Use the inspect_na() function from the inspectdf package to identify columns filled entirely with NA values.

```
inspectdf::inspect_na(non_adherence_clean_2)
```

If a column shows 100% NA values, it's empty and should be removed. To do this, use the remove_empty() function from the janitor package, specifying "cols" to target columns:

```
# Before removal
ncol(non_adherence_clean_2)
```

```
## [1] 15
```

```
# Removing empty columns
non_adherence_clean_3 <- non_adherence_clean_2 %>%
  remove_empty("cols")

# After removal
ncol(non_adherence_clean_3)
```

```
## [1] 14
```

How to Remove Empty Rows

Identifying empty rows is a little more complicated, as there is no function (that we know of) to do this directly. Here is a construction that works:

```
non_adherence_clean_3 %>%
  mutate(missing_count = rowSums(is.na(.))) %>%
  select(missing_count)
```

```
## # A tibble: 1,420 × 1
##    missing_count
##            <dbl>
##  1             2
##  2             0
##  3             3
##  4             3
##  5             2
##  6             2
##  7             1
##  8             2
##  9             3
## 10             2
## # ℹ 1,410 more rows
```

We can then filter the dataset to show only rows with a `missing_count` of 14, indicating that all values in the row are NA:

```
non_adherence_clean_3 %>%
  mutate(missing_count = rowSums(is.na(.))) %>%
  filter(missing_count == 14)
```

```
## # A tibble: 3 × 15
##   patient_id district health_unit sex   age_35
##        <dbl>    <dbl>       <dbl> <chr> <chr>
## 1         NA       NA          NA <NA>  <NA>
## 2         NA       NA          NA <NA>  <NA>
## 3         NA       NA          NA <NA>  <NA>
## # ℹ 10 more variables: age_at_art_initiation <dbl>,
## #   education <chr>, occupation <chr>, …
```

These rows can then be removed using the same `remove_empty()` function from the `janitor` package. This time, specify `"rows"` to target rows:

```
# Before removal
nrow(non_adherence_clean_3)
```

```
## [1] 1420
```

```
# Removing empty rows
non_adherence_clean_4 <- non_adherence_clean_3 %>%
  remove_empty("rows")

# After removal
nrow(non_adherence_clean_4)
```

```
## [1] 1417
```

Notice the change in the row count, indicating the removal of empty rows.

Q: Removing empty columns and rows

Remove both empty rows and empty columns from the `typhoid` dataset. How many rows and columns are left?

Removing Duplicate Rows

Duplicated rows in datasets can occur due to multiple data sources or survey responses. It's essential to identify and remove these duplicates for accurate analysis.

Use `janitor::get_dupes()` to quickly identify duplicate rows. This function helps you visually inspect duplicates before removal. For example:

```
non_adherence_clean_4 %>%
  get_dupes()
```

```
## No variable names specified - using all columns.
```

This output shows duplicated rows, identifiable by common variables like `patient_id`.

After identifying duplicates, use `dplyr::distinct()` to remove them, keeping only the unique rows. For instance:

```
# Before removal
nrow(non_adherence_clean_4)
```

```
## [1] 1417
```

```
# Removing duplicates
non_adherence_distinct <- non_adherence_clean_4 %>%
  distinct()

# After removal
nrow(non_adherence_distinct)
```

```
## [1] 1413
```

We can see that the number of rows has decreased, indicating the removal of duplicates. We can check this by using `get_dupes()` again:

```
non_adherence_distinct %>%
  get_dupes()
```

```
## No variable names specified - using all columns.
```

```
## No duplicate combinations found of: patient_id, district, health_unit,
sex, age_35, age_at_art_initiation, education, occupation, civil_status, ...
and 5 other variables
```

```
## # A tibble: 0 × 15
## # ℹ 15 variables: patient_id <dbl>, district <dbl>,
## #   health_unit <dbl>, sex <chr>, age_35 <chr>, …
```

Q: Removing duplicates

Identify the duplicates in the `typhoid` dataset using `get_dupes()`, then remove them using `distinct()`.

Homogenize strings

You may remember that with the `skim` output from the first data cleaning pipeline lesson, we had instances where our string characters were inconstent regarding capitalization. For example, for our `occupation` variable, we had both `Professor` and `professor`:

```
non_adherence_distinct %>%
  count(occupation, name = "Count") %>%
  arrange(-str_detect(occupation, "rofessor")) # to show the professor rows
first
```

```
## # A tibble: 51 × 2
##    occupation                Count
##    <chr>                     <int>
##  1 Professor                    35
##  2 professor                    11
##  3 Accountant                    1
##  4 Administrator                 1
##  5 Agriculture technician        3
##  6 Artist                        1
##  7 Basic service agent           2
##  8 Boat captain                  1
##  9 Business                      3
## 10 Commercial                   18
## # ℹ 41 more rows
```

In order to address this, we can transform our character columns to a specific case. Here we will use title case, since that looks better on graphics and reports.

```
non_adherence_case_corrected <-
  non_adherence_distinct %>%
  mutate(across(.cols = c(sex, age_35, education, occupation, civil_status),
                .fns = ~ str_to_title(.x))) #
```

If we count the number of unique levels for the `occupation` variable, we can see that we have reduced the number of unique levels from 49 to 47:

```
non_adherence_case_corrected %>%
  count(occupation, name = "Count")
```

```
## # A tibble: 49 × 2
##    occupation                Count
##    <chr>                     <int>
##  1 Accountant                    1
##  2 Administrator                 1
##  3 Agriculture Technician        3
##  4 Artist                        1
##  5 Bartender                     1
##  6 Basic Service Agent           2
##  7 Boat Captain                  1
##  8 Business                      3
##  9 Commercial                   18
## 10 Cook                          3
## # ℹ 39 more rows
```

As we can see, we have gone from 51 to 49 unique levels for the `occupation` variable.

Q: Transforming to lowercase

Transform all the strings in the `typhoid` dataset to lowercase.

`dplyr::case_match()` for String Cleaning

For more bespoke string cleaning, we can use the `case_match()` function from the {dplyr} package. This function allows us to specify a series of conditions and values to be applied to a vector.

Here is a simple example demonstrating how to use `case_match()`:

```
test_vector <- c("+", "-", "NA", "missing")
case_match(test_vector,
           "+" ~ "Positive",
           "-" ~ "Negative",
           .default = "Other")
```

```
## [1] "Positive" "Negative" "Other"    "Other"
```

As you can see, the `case_match()` function takes a vector as its first argument, followed by a series of conditions and values. The `.default` argument is optional and specifies the value to be returned if none of the conditions are met.

We'll apply this first on the `sex` column in the `non_adherence_distinct` dataset. First, observe the levels in this variable:

```
non_adherence_case_corrected %>% count(sex, name = "Count")
```

```
## # A tibble: 2 × 2
##   sex   Count
##   <chr> <int>
## 1 F      1084
## 2 Male    329
```

In this variable, we can see that there are inconsistencies in the way the levels have been coded. Let's use the `case_match()` function so that F is changed to `Female`.

```
non_adherence_case_corrected %>%
  mutate(sex = case_match(sex, "F" ~ "Female", .default = sex)) %>%
  count(sex, name = "Count")
```

```
## # A tibble: 2 × 2
##   sex    Count
##   <chr>  <int>
## 1 Female  1084
## 2 Male     329
```

The utility is more obvious when you have a lot of values to change. For example, let's make the following modifications on the occupation column:

- Replace "Worker" with "Laborer"
- Replace "Housewife" with "Homemaker"
- Replace "Truck Driver" and "Taxi Driver" with "Driver"

```
non_adherence_recoded <-
  non_adherence_case_corrected %>%
  mutate(sex = case_match(sex, "F" ~ "Female", .default = sex)) %>%
  mutate(occupation = case_match(occupation,
                                 "Worker" ~ "Laborer",
                                 "Housewife" ~ "Homemaker",
                                 "Truck Driver" ~ "Driver",
                                 "Taxi Driver" ~ "Driver",
                                 .default = occupation))
```

WATCH OUT

If you don't specify the `.default=column_name` argument then all of the values in that column that don't match the ones those you are changing and explicitly mentioned in your `case_match()` function will be returned as NA.

Q: Fixing strings

The variable `householdmembers` from the `typhoid` dataset should represent the number of individuals in a household. There is a value `01-May` in this variable. Recode this value to `1-5`.

Converting data types

Columns containing values that are numbers, factors, or logical values (TRUE/FALSE) will only behave as expected if they are correctly classified. As such, you may need to redefine the type or class of your variable.

REMINDER

R has 6 basic data types/classes.

- `character`: strings or individual characters, quoted
- `numeric` : any real numbers (includes decimals)

REMINDER

- `integer`: any integer(s)/whole numbers
- `logical`: variables composed of TRUE or FALSE
- `factor`: categorical/qualitative variables
- `Date/POSIXct`: represents calendar dates and times

You may recall in the last lesson that our dataset contains 5 character variables and 9 numeric variables. Let's quickly take a look at our variables using the `skim()` function from last lesson:

```
skim(non_adherence_recoded) %>%
  select(skim_type) %>%
  count(skim_type)
```

Looking at our data, all our variables are categorical, except `age_at_art_initation`, `bmi_initiation_art`, `cd4_initiation_art`, and `nr_of_pills_day`. Let's change all the others to factor variables using the `as.factor()` function!

```
non_adherence_distinct %>%
  mutate(across(
    .cols = !c(
      age_at_art_initiation,
      bmi_initiation_art,
      cd4_initiation_art,
      nr_of_pills_day
    ),
    .fns = as.factor
  )) %>%
  skim() %>%
  select(skim_type) %>%
  count(skim_type)
```

Great, that's exactly what we wanted!

Q: Changing data types

Convert the variables in position 13 to 29 in the `typhoid` dataset to factor.

Wrap Up!

Congratulations on completing the two-part lesson on the data cleaning piepline! You are now better equipped to tackle the intricacies of real-world datasets. Remember, data cleaning is not just about tidying up messy data; it's about ensuring the reliability and accuracy of your analyses. By mastering techniques like handling column names, eliminating empty entries, addressing duplicates, refining string values, and managing data types, you've honed your abilities to transform raw health data into a clean foundation for meaningful insights!

Answer Key

Q: Automatic cleaning

```
clean_names(typhoid)
```

Q: Complete cleaning of column names

```
typhoid %>%
  clean_names() %>%
  rename_with(str_replace_all, pattern = "or_",replacement = "_") %>%
  rename_with(str_replace_all, pattern = "of",replacement = "_") %>%
  names()
```

```
##  [1] "unique_key"                 "case_control"
##  [3] "age"                        "sex"
##  [5] "level_education"            "householdmembers"
##  [7] "below10years"               "n1119years"
##  [9] "n2035years"                 "n3644years"
## [11] "n4565years"                 "above65years"
## [13] "positioninthehousehold"     "watersourcedwithinhousehold"
## [15] "borehole"                   "river"
## [17] "tap"                        "rainwatertank"
## [19] "unprotectedspring"          "protectedspring"
## [21] "pond"                       "shallowwell"
## [23] "stream"                     "jerrycan"
## [25] "bucket"                     "county"
## [27] "subcounty"                  "parish"
## [29] "village"                    "na"
## [31] "nan"
```

Q: Removing empty columns and rows

```
typhoid_empty_removed <-
  typhoid %>%
  remove_empty("cols") %>%
  remove_empty("rows")

# identify how many empty columns were removed
ncol(typhoid) - ncol(typhoid_empty_removed)
# identify how many empty rows were removed
nrow(typhoid) - nrow(typhoid_empty_removed)
```

Q: Removing duplicates

```r
# Identify duplicates
get_dupes(typhoid)
```

```
## No variable names specified - using all columns.
```

```r
# Remove duplicates
typhoid_distinct <- typhoid %>%
  distinct()

# Ensure all distinct rows left
get_dupes(typhoid_distinct)
```

```
## No variable names specified - using all columns.
```

```
## No duplicate combinations found of: UniqueKey, CaseorControl, Age, Sex,
Levelofeducation, Householdmembers, Below10years, N1119years, N2035years, ...
and 22 other variables
```

Q: Transforming to lowercase

```r
typhoid %>%
  mutate(across(where(is.character),
                ~ tolower(.x)))
```

Q: Fixing strings

```r
typhoid %>%
  mutate(Householdmembers = case_match(Householdmembers, "01-May" ~ "1-5",
.default=Householdmembers)) %>%
  count(Householdmembers)
```

Q: Changing data types

```r
typhoid %>%
  mutate(across(13:29, ~as.factor(.)))
```

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education