Lesson Notes | Data Cleaning Pipeline 2: Exploratory Data Analysis

Introduction

In this session, we'll dive into essential techniques to clean and prepare your dataset for analysis. You'll learn to standardize column names, identify and remove empty data, handle duplicates, and correct data types and string values. By the end of this lesson, you'll be equipped to apply a comprehensive data cleaning workflow, ensuring your datasets are accurate and analysis-ready. Let's get started!

Learning Objectives

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

1. Recognize the importance of standardized column names for data analysis.
2. Learn automated and manual methods for cleaning column names.
3. Identify and eliminate empty columns and rows in a dataset.
4. Master techniques to remove duplicate rows and assign correct data types.
5. Correct string inconsistencies, like typos and capitalization errors.
6. Apply a comprehensive data cleaning process on a dataset for analysis.

Packages

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 columns names

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"                      "Health
unit"
##  [4] "Sex"                           "Age_35"                        "Age at
ART initiation"
##  [7] "EDUCATION_OF_PATIENT"          "OCCUPATION_OF_PATIENT"
"Civil...status"
## [10] "WHO status at ART initiaiton" "BMI_Initiation_Art"
"CD4_Initiation_Art"
## [13] "regimen.1"                    "Nr_of_pills_day"               "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 cleaning

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.
- Parses letter cases and separators to a consistent format. (default is snake_case)
- Handles special characters(&, #, <, >) or accented characters.

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

```
##  [1] "patient_id"                    "district"
"health_unit"
##  [4] "sex"                           "age_35"
"age_at_art_initiation"
##  [7] "education_of_patient"          "occupation_of_patient"
"civil_status"
## [10] "who_status_at_art_initiaiton" "bmi_initiation_art"
"cd4_initiation_art"
## [13] "regimen_1"                    "nr_of_pills_day"              "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`)

INTRODUCE TYPHOID DATASET

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

Manually cleaning column names

We can also rename columns manually, either as an alternative to the automatic procedure described above or **in addition to the automatic step above**.

Let's first illustrate how to manually clean names with `rename()` and `rename_with`.

`rename()`

The `rename()` function is simply a way to change variable names. It was covered in our data wrangling chapter. As a reminder, its syntax is so: `rename(new_name = old_name)`

Here, manually, we could rename `Civil...status` to `civil_status` and `Age at ART initiation` to `age_at_art_initiation`.

```
        non_adherence %>% rename(civil_status = `Civil...status`,
                                 age_at_art_initiation = `Age at ART
initiation`) %>%
                          names()
```

```
##  [1] "patient_id"                    "District"                      "Health
unit"
##  [4] "Sex"                           "Age_35"
"age_at_art_initiation"
##  [7] "EDUCATION_OF_PATIENT"          "OCCUPATION_OF_PATIENT"
"civil_status"
## [10] "WHO status at ART initiaiton" "BMI_Initiation_Art"
"CD4_Initiation_Art"
## [13] "regimen.1"                     "Nr_of_pills_day"               "NA"
```

Evidently, this would make a long list if we have a dataset with many columns. Let's see how to do this quicker with `rename_with()`.

`rename_with()`

Instead of changing each column individually, `rename_with` is a handy shortcut to apply the same modification to all (or a subset of) column names.

The arguments of `rename_with` are:

- a data frame
- a function to apply to each column name
- a selection of columns to rename (by default all columns)

Say, for example, we want all the column names to be in lower case, we can use `rename_with` in the following way:

```
        non_adherence %>%
          rename_with(tolower) %>%
          names()
```

```
##  [1] "patient_id"                    "district"                      "health
unit"
##  [4] "sex"                           "age_35"                        "age at
art initiation"
##  [7] "education_of_patient"          "occupation_of_patient"
```

```
"civil...status"
## [10] "who status at art initiaiton" "bmi_initiation_art"
"cd4_initiation_art"
## [13] "regimen.1"                    "nr_of_pills_day"              "na"
```

Here we see that the columns `EDUCATION_OF_PATIENT` and `DISTRICT` have been transformed to lower case. If we want to perform more complex operations, such as modifying column names with unwanted characters (e.g., whitespaces and special characters), we can combine it with the `str_replace` function.

This function allows us to replace matched patterns in a string. The arguments are:

- the current pattern
- a replacement pattern

WATCH OUT

To replace special characters like the period (`.`), we adjust the syntax slightly by adding \\ before the special character.

Let's build on what we did before. We corrected for upper cases, now let's correct whitespaces and periods (`.`).

```
        non_adherence %>%
          rename_with(tolower) %>%
          rename_with(str_replace, pattern = " ",replacement = "_") %>%
          rename_with(str_replace, pattern = "\\.",replacement = "_") %>%
          names()
```

```
##  [1] "patient_id"                  "district"
"health_unit"
##  [4] "sex"                         "age_35"                       "age_at
art initiation"
##  [7] "education_of_patient"        "occupation_of_patient"
"civil_..status"
## [10] "who_status at art initiaiton" "bmi_initiation_art"
"cd4_initiation_art"
## [13] "regimen_1"                   "nr_of_pills_day"              "na"
```

As we can see here, `str_replace` only corrects for the first instance of our specified pattern, so `age at art initiation` has become `age_at art initiation` and `civil...status` has become `civil_..status`. If we want to replace all instances, we need to use the `str_replace_all` function.

```
        non_adherence %>%
          rename_with(tolower) %>%
          rename_with(str_replace_all, pattern = " ",replacement = "_") %>%
          rename_with(str_replace_all, pattern = "\\.",replacement = "_") %>%
          names()
```

```
##  [1] "patient_id"                    "district"
"health_unit"
##  [4] "sex"                           "age_35"
"age_at_art_initiation"
##  [7] "education_of_patient"          "occupation_of_patient"
"civil___status"
## [10] "who_status_at_art_initiaiton" "bmi_initiation_art"
"cd4_initiation_art"
## [13] "regimen_1"                    "nr_of_pills_day"              "na"
```

But even with this, we still have a weird column name, `civil___status`. These types of issues are cleaned automatically by {janitor}'s `clean_names()`: remember to use this function first! **It will save you a lot of trouble/manipulations.**

Manually rename the following column names in the `typhoid` dataset; `CaseorControl`and `Levelofeducation` to `case_control`, `education_level`, respectively.

Automatic then manual cleaning of column names

Evidently, manually cleaning column names can be a cumbersome task, particularly when cleaning a large dataset with many variables. A combination of the automatic and manual procedure is more desirable as it not only saves on time, but can help make our column names more readable. We can start with the automatic clean-up, then check there are no weird column names remaining (which would need manual cleanup).

Here is an example of combining automatic and manual cleaning: a more appropriate cleanup would be to standardise the column names, first, and then remove the ending `_of_patient` from all columns. Let's create a new dataframe called `non_adh_cols`.

```
        non_adh_clean_names <- non_adherence %>%
                  # standardize column name syntax
                  clean_names() %>%
                  # manually re-name columns
                  rename_with(str_replace_all, pattern =
"_of_patient",replacement = "")
        non_adh_clean_names
```

```
## # 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.

Standardize the column names in the `typhoid` dataset then;

- replace `or_` with _
- replace `of` with _
- rename variables `below10years` `n1119years` `n2035years` `n3644years`, `n4565years` `above65years` to `num_below_10_yrs` `num_11_19_yrs` `num_20_35_yrs` `num_36_44_yrs`, `num_45_65_yrs` `num_above_65_yrs`

Removing empty columns and rows

An **EMPTY** row/column is one where all values are NA values. When you load a dataset, you always want to check if it has any empty rows or columns and remove them. The goal is that **every row is a meaningful data point** and that **every column is a meaningful variable**. Let's start with our colmuns.

Removing empty columns

To identify empty colmuns, we're going to use the `inspect_na()` function from the package `inspectdf` to identify empty columns.

```
        inspectdf::inspect_na(non_adh_clean_names)
```

From the output we see that the `pcnt` indicates 100% emptiness (i.e. NA values) for our `na` column: there is a NA values in every row.

In order to remove empty columns from the data frame, we will use the `remove_empty()` function from the `janitor` package. This function removes all columns from a data frame that are composed entirely of NA values.

We will apply the function on the `non_adh_clean_names` dataset and remove the empty column identified earlier.

```
        ncol(non_adh_clean_names)
```

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

```
        non_adh_clean_names <- non_adh_clean_names %>%
          remove_empty("cols")

        ncol(non_adh_clean_names)
```

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

Here, we can see that the column `na` has been removed from the data.

Remove the empty columns from the `typhoid` dataset.

Removing empty rows

While it is relatively easy to identify empty columns from the `skim()` output, its not as easy to do so for empty rows. Fortunately, the `remove_empty()` also works if there are empty rows in the data. The only change in the syntax is specifying `"rows"`instead of `"cols"`.

```
        nrow(non_adh_clean_names)
```

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

```
        non_adh_clean_names <- non_adh_clean_names %>%
          remove_empty("rows")

        nrow(non_adh_clean_names)
```

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

The number of rows has gone from 1420 to 1417 suggesting there were empty rows in the data that have been removed.

Remove both empty rows and empty columns from the `typhoid` dataset.

Removing duplicate rows

Very often in your datasets there are situations where you have duplicated values of data, when one row has the exact same values as another row. This can occcur when you combine data from multiple sources, or have received multiple survey responses.

It's therefore necessary to identify and remove any duplicate values from your data in order to ensure accurate results. For this we will be using two functions: `janitor::get_dupes()` and `dplyr::distinct()`.

`janitor::get_dupes()`

An easy way to quickly review rows that have duplicates is the `get_dupes()` function from the `janitor` package.

The syntax is `get_dupes(x)`, where x is **a dataframe**.

Let's try it on our `non_adh_clean_names` dataframe.

```
non_adh_clean_names %>%
  get_dupes()
```

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

The output is made up of 8 rows: there are 2 rows for each pair of duplicates. You can easily see they are duplicates based on the `patient_id` variable which uniquely identifies our observations.

Identify the elements that are duplicates from the `typhoid` dataset.

The `get_dupes()` is a useful function to extract your duplicated data before you remove it. Sometimes if you have lots of duplicates, it can indicate a problem that requires further investigation, such as an issue with data collection or merging. Once you've reviewed your duplicates, you can remove them with the following function.

`dplyr::distinct()`

`distinct()` is a function from the `dplyr` package that only keeps unique/distinct rows from a dataframe. If there are duplicate rows, only the first row is preserved.

```
        nrow(non_adh_clean_names)
```

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

```
        non_adh_distinct <- non_adh_clean_names %>%
          distinct()

        nrow(non_adh_distinct)
```

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

Initially, our dataset had 1417 rows. Applying the `distinct()` function reduces the dimensions of the dataset to 1413 rows. This makes sense, because as we saw earlier, 4 of our values were duplicates.

Remove the duplicate rows from the `typhoid` dataset. Ensure only unique rows remain in the dataset

---

Transformations

Fixing strings

There are often times when you need to correct some inconsistencies in strings that might interfere with data analysis, including typos and capitalization errors.

These issues can be fixed manually in the raw data source or we can make the change in the cleaning pipeline. The latter is more transparent and reproducible to anyone else seeking to understand or repeat your analysis.

Changing string values with `recode()`

We can use the `recode()` function within the `mutate()` function to change specific values and to reconcile values not spelled the same.

The syntax is `recode(column_name, old_column_value = new_column_value)`

First, let's have a look at the `sex` column:

```
        non_adh_distinct %>% 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 `recode()` function so that `F = Female`.

```
        non_adh_distinct %>%
          mutate(sex = recode(sex, `F` = "Female")) %>%
          count(sex, name = "Count")
```

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

That looks great! `recode()` is an easy way to fix mismatching values in your dataset. Sometimes we have more intricate patterns that we need to fix, such as conditional recoding and removing special characters. For more in-depth information on how to deal with strings, check out the strings lesson!

The variable `householdmembers` from the `typhoid` dataset should represent the number of individuals in a household. Display the different values in the variable.

There is a value `01-May` in the `householdmembers` variable in the `typhoid` dataset. Recode this value to `1-5`.

Homogenize all strings throughout the dataset

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`.

In order to address this, we can transform all our strings to lowercase using the `tolower()` function. We select all character type columns with `where(is.character)`

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

```
        non_adh_distinct %>%
          mutate(across(where(is.character),
                        ~ tolower(.x))) %>%
          count(occupation, name = "Count")
```

As we can see, we have gone from 51 to 49 unique levels for the `occupation` variable when we transform everything to lowercase!

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

Cleaning 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.

R has 6 basic data types/classes.

- `character`: strings or individual characters, quoted
- `numeric` : any real numbers (includes decimals)
- `integer`: any integer(s)/whole numbers
- `logical`: variables composed of TRUE or FALSE
- `factor`: categorical/qualitative variables
- `Date/POSIXct`: represents calendar dates and times

In addition to the ones listed above, there is also `raw` which will not be discussed in this lesson.

You may recall in the last lesson that our dataset contained 5 character variables and 9 numeric variables (as well as 1 logical variable NA that has since been removed as it was completely empty). Let's quickly take a look at our variables using the `skim()` function from last lesson:

```
        skim(non_adh_distinct) %>%
        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_adh_distinct %>%
  mutate(across(!c(age_at_art_initiation,
                   bmi_initiation_art,
                   cd4_initiation_art,
                   nr_of_pills_day),
                ~ as.factor(.x))) %>%
  skim() %>%
  select(skim_type) %>%
  count(skim_type)
```

SIDE NOTE

As a reminder from the data wrangling lesson: we use ~ to indicate that we're supplying an anonymous function and use `.x` to indicate where the variables supplied in `across()` are used.

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

Great, that's exactly what we wanted!

Putting it all together

Now, let's apply the strings and data type tranformations together!

```
non_adherence_clean <- non_adh_distinct %>%
  # Recoding sex values
  mutate(sex = recode(sex, `F` = "Female")) %>%
  # Changing all to lowercase
  mutate(across(where(is.character),
                ~ tolower(.x))) %>%
  # Changing variable types
    mutate(across(!c(age_at_art_initiation,
                   bmi_initiation_art,
                   cd4_initiation_art,
                   nr_of_pills_day),
                ~ as.factor(.x)))

non_adherence_clean
```

Contributors

The following team members contributed to this lesson:

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

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network