Introduction to R - Young Researchers Fellowship Program

Lecture 7 - Advanced topics in data cleaning - date management and pivoting

Daniel Sánchez Pazmiño

Laboratorio de Investigación para el Desarrollo del Ecuador

October 2024

Date Management with R

Dates in R

- **Why manage dates in R?**
  - Data often involves dates (e.g., time series, logs).
  - R provides powerful tools for handling dates, times, and time zones.
- Key libraries:
  - `base::Date`
  - `lubridate` (for more intuitive handling)

Dates Act Like Numbers

- **Date** objects in R are stored as the number of days since **1970-01-01**.
  - This makes date comparisons and arithmetic straightforward.
- **Examples**:

```
as.Date("2003-02-27") > as.Date("2002-02-27")   # TRUE
as.Date("2003-02-27") + 1                       # "2003-02-28"
as.Date("2003-02-27") - as.Date("2002-02-27")   # Time differenc
```

- Dates can be treated like numbers when performing arithmetic.

ISO 8601 YYYY-MM-DD

- **ISO 8601** is a global standard for dates:
  - Dates ordered from largest to smallest unit: **YYYY-MM-DD**.
  - Each unit is padded with leading zeros if needed.
  - 1st of January 2011 → 2011-01-01.
- **Usage in R**:
  - This format makes date comparison and sorting consistent across systems.

Base R Date Functions

- **Creating dates**

```r
# Date from string
as.Date("2024-10-20")
# Date from numeric values
as.Date(c("2024-10-20", "2022-05-12"))
```

- **Current date and time**

```r
Sys.Date()   # Today's date
Sys.time()   # Current time
```

- **Date arithmetic**

```r
# Adding days
Sys.Date() + 10
# Difference between dates
as.Date("2024-10-20") - as.Date("2022-05-12")
```

lubridate Package

- **Why lubridate?**
  - Simplifies working with dates and times.
  - Easily parse, manipulate, and perform arithmetic with dates.
- **Examples:**

```r
library(lubridate)

# Parsing dates
ymd("20241020")
mdy("10/20/2024")

# Date arithmetic
today() + days(5)
year(today())
```

Parsing Dates with `lubridate`

- **Using `ymd()` and other parsing functions:**

```
# Parsing dates in different formats
ymd("20241020")      # Year-Month-Day
dmy("20-10-2024")    # Day-Month-Year
mdy("10/20/2024")    # Month-Day-Year
```

- **Handling date-time:**

```
ymd_hms("2024-10-20 10:30:00")
```

Further parsing

- Sometimes we need to "get serious" about the parsing:

```
parse_date_time("27-02-2013", order = "dmy")
parse_date_time(c("27-02-2013", "2013 Feb 27th"), order = c("dmy
```

Formatting Characters for Dates

- **Date formatting characters** allow customization of date outputs:
  - `d` = Numeric day of the month.
  - `m` = Month of year.
  - `y` = Year with century.
  - `H` = Hours (24-hour format).
  - `M` = Minutes.
  - `S` = Seconds.
- These can be used to format and extract specific components of dates.

Creating Dates with `make_date()`

- **Create dates from year, month, and day** using `make_date()`:

```
make_date(year = 2013, month = 2, day = 27)
```

- **Create datetimes** using `make_datetime()` for dates with hour, minute, and second components.

Rounding vs. Extracting Dates

- **Rounding**:
  - Use `round_date()`, `floor_date()`, or `ceiling_date()` to round dates to the nearest, up, or down.

```r
floor_date(Sys.time(), unit = "hour")
```

- **Extracting** retains only the desired components:

```r
hour(Sys.time())     # Extract hour
```

Date Arithmetic with `lubridate`

- **Adding and subtracting dates:**

```r
today() + days(10)
today() - months(1)
```

- **Extracting components of dates:**

```r
year(today())
month(today(), label = TRUE)  # Get the month name
day(today())
```

Time Spans in `lubridate`

- **What are Time Spans?**
  - Time spans refer to periods, durations, and intervals, used to measure or manipulate time differences.
- **Creating Periods**:

```
days(2)                 # 2 days
ymd("2023-02-27") + days(1)  # Add 1 day
```

- **Handling Durations** (fixed number of seconds):

```
duration <- ddays(2)    # 2 days in seconds
ymd("2023-02-27") + duration
```

- **Working with Intervals**:

```
interval(ymd("2023-01-01"), ymd("2023-12-31"))
```

Time Zones and Date-Time Conversion

- **Setting time zones:**

```
now(tz = "UTC")                # Current time in UTC
with_tz(now(), "America/Los_Angeles")  # Convert to another time
```

- **Working with `POSIXct` and `POSIXlt`:**

```
as.POSIXct("2024-10-20 10:30:00", tz = "America/New_York")
as.POSIXlt("2024-10-20 10:30:00", tz = "America/New_York")
```

Key Takeaways

- Use `as.Date()` and `Sys.Date()` for basic date management.
- For more advanced date handling, use `lubridate`.
- Be mindful of time zones when working with date-time.

Time Series in R

- **Base `ts` object:**
  - Simple, built-in time series object.
  - Works well for regular, equally spaced data.
- `zoo` **package:**
  - Extends time series to irregular intervals.
  - Great for handling missing or irregularly spaced data.
- `xts` **package:**
  - Built on `zoo` but optimized for financial data.

Graphing Dates with ggplot2

- **Plotting Time Series:**

```r
library(ggplot2)
# Example: Plotting with ggplot2
ggplot(releases, aes(x = date, y = type)) +
  geom_line(aes(group = 1, color = factor(major))) +
  xlim(as.Date('2010-01-01'), as.Date('2014-01-01'))
```

- **Customizing Date Axis:**

```r
ggplot(releases, aes(x = date, y = type)) +
  geom_line(aes(group = 1, color = factor(major))) +
  scale_x_date(date_breaks = '10 years', date_labels = '%Y')
```

Pivoting or reshaping datasets

Pivoting Data in R

- **Pivoting** is the process of reshaping data, commonly used for summarizing and reorganizing datasets.
- In R, pivoting is done using the `tidyr` package, part of the tidyverse.
- There are two primary types of pivoting:
  - **Pivot Longer**: Converts wide data into long format.
  - **Pivot Wider**: Converts long data into wide format.

Example

- Consider the following dataset:

lugar	year_2020	year_2021	year_2022	year_2023
Cuenca	72	37	82	83
Cumbayá	6	4	5	1
Guayaquil	73	46	107	44
Loja	8	4	8	4
Quito	386	199	252	148

- Is it tidy? How can we put it in a better format for statistical analysis?

Tidyr

- `tidyr` is a part of the `tidyverse` collection of packages.
- It helps you create tidy data:
  - Each variable in its own column.
  - Each observation in its own row.
  - Each value in its own cell.
- **Key tasks**:
  - Pivoting data (longer and wider).
  - Handling missing values.
  - Separating and uniting columns.
- `tidyr` is essential for preparing data for analysis in a clean, organized manner.

Essentials of `tidyr`

1. **`pivot_longer()`**:
   - Converts wide data into long format by gathering columns.
   - Useful when each column is a separate variable.
2. **`pivot_wider()`**:
   - Spreads long data into wide format, converting key-value pairs into columns.
3. **`separate()`**:
   - Splits one column into multiple columns.
   - Useful when a single column contains multiple variables.
4. **`unite()`**:
   - Combines multiple columns into a single column.
5. **`fill()`**:
   - Fills missing values with the last known value.

Pivot Longer

- **Pivot longer** is used when you want to collapse multiple columns into two key columns:
  - One column for the variable name.
  - One column for the value.
- **Example**:

```
used_cars_long <-
  used_cars_wide |>
  pivot_longer(year_2020:year_2023, names_to = "year", values_to

used_cars_long
```

```
# A tibble: 20 x 3
   lugar     year       cars
   <chr>     <chr>     <int>
 1 Cuenca    year_2020    72
 2 Cuenca    year_2021    37
 3 Cuenca    year_2022    82
```

Pivot Wider

- **Pivot wider** is the opposite of pivot longer. It spreads key-value pairs across multiple columns.
- **Example**:

```
used_cars_long |>
  pivot_wider(values_from = cars, names_from = year)
```

- This transforms long data into a wide format, often used to make datasets easier to analyze.

Customizing Pivot Operations

- You can **drop missing values** while pivoting by using the `values_drop_na` argument.
  - This avoids including rows with `NA` values in the new column.
- Can also include or remove prefixes with `names_prefix`
- You can **pivot multiple variables** at once by specifying them in the `names_to` argument, and including a vector of columns (this is for `pivot_longer()`)

Alternatives to `tidyr`

- **`reshape2`**:
  - One of the original packages for reshaping data.
  - Functions like `melt()` and `dcast()` are similar to `pivot_longer()` and `pivot_wider()`.
  - However, `reshape2` is more manual and less flexible than `tidyr`.
- **`data.table`**:
  - A high-performance alternative for working with large datasets.
  - Provides `melt()` and `dcast()` functions for reshaping data.
  - Extremely fast and memory-efficient, especially with large datasets.
- **Base R**:
  - You can reshape data using `reshape()` in base R.
  - Though less intuitive, it's a viable alternative for basic reshaping tasks.

separate_rows()

- **separate_rows()** splits a column where multiple values are stored in a single cell.
- It turns the one row into multiple rows, one for each value.
- **Example**: Splitting a column where values are separated by commas:

```r
data <- tibble(
  id = c(1, 2),
  tags = c("A,B,C", "D,E")
)

data_separated <- separate_rows(data, tags, sep = ",")
```

- Useful when working with delimited lists stored within a single column.

complete()

- `complete()` ensures that all combinations of variables are present in your data.
- It's useful when some combinations are missing but should be included.
  - "Implicit NAs" to "Explicit NAs"
- **Example**: Completing all combinations of `year` and `product`:

```r
sales_data <- tibble(
  year = c(2020, 2021),
  product = c("A", "B"),
  sales = c(100, 150)
)

complete_data <- complete(sales_data, year, product)
```

- **Result**: Original data: | year | product | sales | |——|———|——-| | 2020 | A | 100 | | 2021 | B | 150 |

unite()

- **unite()** is used to combine multiple columns into a single column.
- It concatenates the values from the columns and separates them with a specified delimiter.
- **Syntax**:

```r
unite(data, new_column_name, col1, col2, ..., sep = "_")
```

- It's particularly useful when you need to combine categorical variables into one column.

Example: Using `unite()`

- Suppose we have a dataset where `first_name` and `last_name` are in separate columns, and we want to combine them into a single `full_name` column.
- **Original Data**:

```
people <- tibble(
  first_name = c("John", "Jane"),
  last_name = c("Doe", "Smith")
)

people
```

```
# A tibble: 2 x 2
  first_name last_name
  <chr>      <chr>
1 John       Doe
2 Jane       Smith
```

Example: Using `unite()`

- **Using** `unite()`:

```r
people_united <- unite(people,
                       full_name,
                       first_name,
                       last_name,
                       sep = " ")
```

`separate()` Function Overview

- **`separate()`** is used to split a single column into multiple columns based on a delimiter or pattern.
- It helps when one column contains multiple variables that should be spread across multiple columns.
- **Syntax**:

```
separate(data, col, into, sep = " ", remove = TRUE)
```

Example of separate()

- Suppose we have a dataset where a date_time column contains both the date and time, and we want to split it into separate date and time columns.
- **Original Data**:

```
date_time_data <- tibble(
  date_time = c("2023-10-20 12:30", "2024-11-15 08:45")
)

date_time_data
```

```
# A tibble: 2 x 1
  date_time
  <chr>
1 2023-10-20 12:30
2 2024-11-15 08:45
```

Splitting on Multiple Delimiters

- **`separate()`** can handle more complex delimiters, such as splitting on characters like commas, slashes, or hyphens.
- **Example**: Splitting a column that contains names with both first and last names separated by commas:

```
name_data <- tibble(
  full_name = c("John,Doe", "Jane,Smith")
)

name_separated <- separate(name_data, full_name, into = c("first
```

Handling Missing Values with separate()

- If a row doesn't have enough values to split into all columns, separate() will fill the missing cells with NA.
- **Example**: Splitting with missing values:

```
incomplete_data <- tibble(
  full_name = c("John,Doe", "Jane")
)

name_separated <-
  separate(incomplete_data, full_name,
           into = c("first_name", "last_name"), sep = ",")
```