Data Wrangling Workshop

McGill initiative in Computational Medicine

Sean Nesdoly

2021-04-15

Contents

---

1. To Share—to copy, distribute and transmit the work;
2. To Remix—to adapt the work,

Under the following conditions:

- Attribution—You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike—If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.

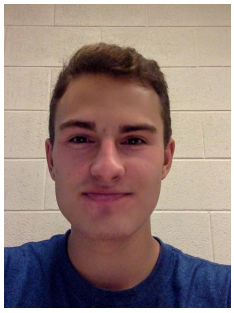
0 Introduction

This workshop will introduce the fundamental principles behind data wrangling as well as present some of the best practices for implementing these concepts in both the R and Julia programming languages. Specifically, it will cover: (1) exploration of raw data from the genomics domain; (2) identification of applicable data structures specific to the task at hand; (3) the manipulation or 'wrangling' of data into a desired structure using the R `tidyverse` packages; (4) cleaning the structured data, as needed; and, (5) transforming the 'wrangled' data into a format suitable for downstream analyses or archival storage, with reusability in mind. An alternative formulation of basic data wrangling techniques will also be presented in Julia, demonstrating that these concepts are generalizable between programming languages.

0.1 Main Objectives

1. To understand the fundamental principles behind data wrangling, from ingestion of raw data to its final form as a well-structured, annotated, and clean dataset.
2. To become proficient with data manipulation in **R** using the `tidyverse` packages.
3. Familiarize yourself with **Julia**, a general-purpose, high performance, open source programming language for computational medicine and beyond.

0.2 Instructor Biography



I'm Sean, a passionate programmer with a love for biology. I recently graduated from the Biological & Biomedical Engineering program at McGill University with an M.Eng. degree that focused on bioinformatic algorithm development; my undergraduate training was in Computer Science. I am originally from Calgary, Alberta. Outside of work, I enjoy hiking, cycling, hockey, and cooking new dishes (attempting to, at least). I am always excited to learn new things, and to teach!

1 Workshop Prerequisites & Setup

1.1 Obtain Workshop Materials

Clone or download the following `git` repository from GitHub:

https://github.com/SeanNesdoly/MiCM-Data-Wrangling-Workshop

Option 1: Download Repository

To **download** the repository, look for the green button labelled 'Code' and click on 'Download ZIP'. Once it downloads, extract the ZIP file to your desired location.

Option 2: Clone Repository

To **clone** the repository, execute the below code from a terminal. When `git` prompts you for your password, enter your GitHub personal access token (see here for details).

```
# Replace the file path below with your own location!
cd ~/path/to/working/directory/
git clone https://github.com/SeanNesdoly/MiCM-Data-Wrangling-Workshop.git
```

1.2 Overview of Workshop Materials

1. Presentation material.
2. Material within the GitHub repository, listed by folder:
    - `docs`: Workshop PDF handout, containing hands-on lecture material with inline code.
    - `src`: R and Julia source code.
    - `data`: Input datasets used by the code in `src`.
    - `out`: Output data; generated by executing the code in `src` on the input data in `data`.

1.3 Software Installation

Complete the following before attending the workshop:

1. Install R (version 3.5+; I'm using 4.0.4): https://utstat.toronto.edu/cran/
2. Install RStudio Desktop (the free version): https://rstudio.com/products/rstudio/download/
3. Install all R packages contained within the `tidyverse`; to do so, execute the following lines in R:

   ```
   install.packages("tidyverse")
   library(tidyverse)
   ```

4. Install Julia (version 1.6.0): https://julialang.org/downloads/

2 Basic Principles of Data Wrangling

2.1 Exploring Raw Data

- This step is critical prior to conducting downstream data wrangling & analysis. Spending time here can reduce future headaches.
- Given the structure of the raw data, define a schema for what you want the output to look like.
  - What figures do you want to create?
  - What datasets do you need to generate the final output?

Viewing Large Datasets

- `head` & `tail`
  - Output the first 100 lines: `head -n 100 in.txt`
- `more` & `less`
- `wc` – word, line, character, and byte count
  - Count the number of lines:

    ```
    wc -l file
    ```
  - Count the number of words:

    ```
    wc -w file
    ```
  - Count the number of characters (bytes):

    ```
    wc -c file
    ```

File Pattern Search Utility: `grep`

- Search for an exact string:

  ```
  grep search_string path/to/file
  ```
- Case-insensitive search:

  ```
  grep -i search_string path/to/file
  ```
- Search recursively (ignoring non-text files) in current directory for an exact string:

  ```
  grep -RI search_string .
  ```
- Print file name with the corresponding line number for each match:

  ```
  grep -Hn search_string path/to/file
  ```

Verification of Data Integrity

- Compute a cryptographic hash for your downloaded dataset to produce a 'message digest'; you can then compare this to the hash given by the provider of the data to ensure integrity.
- Secure Hash Algorithm 2 (SHA-2) family of cryptographic hash functions.
  - e.g. `sha256sum`, `sha512sum`
  - The number corresponds to how many bits are in the message digest.
- For example: the `COXPRESdb` gene coexpression database provides `md5` message digests to verify dataset integrity
  - Navigate to https://coxpresdb.jp/download/ and look for `md5`.

2.2 Guidelines for Creating Reproducible Data Wrangling Projects

Raw Datasets

- Raw (input) datasets should be **unmodified** throughout the entire analysis.
  - This ensures that others can reproduce your results given only the raw data.
  - However, in doing so, it is important to document the software environment, tools, and packages that you use in your analysis to guard against potential changes that break your toolset (e.g. updates to R or its packages).
  - If desired, you can remove **write** permissions for your raw datasets to prevent unwanted changes.

    `chmod a-w ~/path/to/file`

Project Template

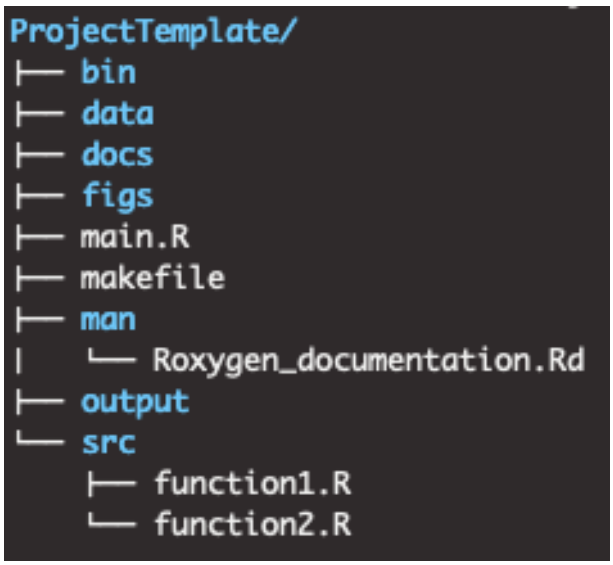


Figure 1: Template for reproducible data wrangling projects.

Appending Timestamps to Your Output

```
time <- function() {
    gsub('-', '', Sys.time()) %>% gsub('\\s+', '-', .) %>% gsub(':', '', .)
}

time() # append to output filenames, plots
```

3 Fundamentals of Data Wrangling in R using the `tidyverse`

3.1 The `tidyverse`: An 'Opinionated' Collection of R Packages

- Each package within the `tidyverse` shares common:
  - design
  - grammar (syntax)
  - data structures
- It is open source, which means you can view—and even contribute to—the code!
  - GitHub project page: https://github.com/tidyverse
  - Contributing: https://www.tidyverse.org/contribute/
- Developed by Hadley Wickham & co., with support from RStudio
- Heavily influenced by concepts taken from relational databases (e.g. Oracle, MySQL).

Reading & Writing Data

- `readr`
- `readxl`

Tidying Data

- `tibble`
- `tidyr`

General Programming

- `magrittr`
- `purrr`

Data Wrangling & Cleaning

- `dplyr`
- `lubridate`
- `stringr`
- `forcats`

Visualization

- `ggplot2`

3.2 The `tibble` Data Frame

- An alternative, enhanced version of the base R `data.frame` class.
- Selling point: `tibble`'s '...do less and complain more'

Table 1: A `tibble` is made up of variables (columns) and samples (rows).

	x_1	x_2	...	x_n
s_1				
s_2				
...				
s_m				

- A sample—one row of the `tibble` (think observations, cases)—is the set of values that the variables $\{x_1, x_2, ..., x_n\}$ take on for a given instance.
  - In mathematical terms, a sample is a 'tuple'.
- A set of n samples $\{s_1, s_2, ..., s_n\}$, where each sample s_i is an array/vector/tuple of the variables $\{x_1, x_2, ..., x_n\}$, makes up a `tibble`.
- Variables can take on the following types:
  - `int` = integers
  - `dbl` = doubles (double-precision floating point numbers)
  - `chr` = strings (character vectors)
  - `lgl` = logicals (true or false boolean values)
  - `fctr` = factors
    - These represent categorical variables by encoding each pre-defined string as an integer.
  - `date` = date

- A few advantages:
  1. Integrates seamlessly with the `tidyverse` packages as a type of 'tidy' data
  2. Keeps data in its raw format, rather than coercing data into a specific type or changing variable names without your knowledge.
     - The base R `data.frame` converts your input by default, which can be troublesome if you are not careful. A typical example is the conversion of strings to factors.
  3. Enhanced print method for large datasets.

- In RStudio, open up `src/FundamentalsOfTidyverse.R`

3.3 Reading & Writing Data with `readr`

- `readr::read_*()`
- `readr::write_*()`

Working with Excel files in the `readxl` package

- https://readxl.tidyverse.org/

3.4 Tidying Data with `tidyr`

Handling Missing (null) values

- `tidyr::drop_na()`
- `tidyr::fill()`
- `tidyr::replace_na()`

Pivots

- `tidyr::pivot_longer()`
- `tidyr::pivot_wider()`

3.5 `stringr`

- Regular expressions ('regex') for defining patterns to search for
  - Having a basic understanding of regex is incredibly powerful for data wrangling!

3.6 Data Wrangling with `dplyr`

`dplyr::mutate()`

- Creates new variables based on existing ones.
- `dplyr::transmutate()` mutates a variable and transforms the `tibble` by dropping existing ones.

```
new_tbl_df <- tbl_df %>% mutate(new_col = x)
```

`select()`

- Select variables (columns) by their name.

```
tbl_df %>% select(col1, col2)
```

`filter()`

- Pick out samples (rows) based on their values.

```
filter(df, col_name == "x")
```

`summarise()`

- Within a variable (column), collapse values into a single *summary* by performing an operation on them.

```
summarize(new_col_name = operator(col))
```

`arrange()`

- Reorder samples (rows).

```
tbl_df %>% arrange(desc(col))
```

3.7 Preparing Data for Downstream Analyses

4 Advanced Data Wrangling in R using the `tidyverse`

- Open `src/AdvancedTidyverse.R` in RStudio.

4.1 Piping with `magrittr`

```
f(x)
x %>% f
```

4.2 Functional Programming with purrr

`map()`

- `map(array, foo)`
- `map(array, function(x) # code here)`
- `map(array, ~#code here) # this is a lambda function`

4.3 Data Aggregation

Set Operations

- Union
- Intersection
- Difference

Joins

- `joined <- left_join(left_table, right_table, by=c("left_colname" = "right_colname"))`

Group Operators

- `groupby()`
- `summarise()`

5 Introduction to Julia

- A cross between C, R, Python, and MATLAB.
- Extremely configurable as a high-level language.
  - Has the option to directly interface with other languages (C/C++, Fortran, R, Python, Java). Can also be called from R or Python.
- Designed with speed & efficiency in mind.
  - Compiled code directly interfaces with the 'bare-metal' hardware in a highly optimized manner due to the design of its type system.
  - Performance rivals that of C.
- The ability to add explicit type annotations into code improves human readability and catches errors upfront, both of which are issues in R.
  - For example, 'function foo(num::Int)::String' defines a function 'foo' that takes an integer 'num' and returns a string.

5.1 Possible use case, coming from a background in R or Python

1. Ingest data with Julia.
2. Wrangle in the R `tidyverse`, or in Julia itself (`DataFrames.jl`).
3. Perform resource-intensive operations in Julia.
4. Visualize data with R's `ggplot2`, or directly with packages in Julia (`Gadfly.jl`, `Plots.jl`).

5.2 The Julia Read-Eval-Print Loop (REPL)

Table 2: Julia REPL keybindings

Key	Description
`?`	help
`^C`	interrupt/cancel a command
`]`	package manager ('Pkg mode')
`;`	shell mode*
`^R`	reverse search through history of commands
`^S`	forward search through history of commands

* To use shell mode in Windows, type `powershell` or `cmd` while in shell mode to access shell commands.

5.3 Adding Packages

1. Enter 'Pkg mode' by entering `]` into the Julia REPL. This will change the prompt from `julia>` to `(@v1.6) pkg>`.
2. Add in the following packages by entering the below command into the REPL:
   `add DataFrames, StatsBase, Statistics`
3. For a complete list of commands, enter `help`.
4. To view the current status of packages that are added to your session, enter `status`.

5.4 Hands-on Code Examples

- See `src/IntroToJulia.jl`.

5.5 Julia vs. R vs. Python

Reproducibility

- Well-designed package manager.
- Effortlessly recreate package environments, with the ability to *pin* packages in a given state.
- Pre-built binaries available for numerous platforms. It just works.

Performance

- As R is an interpreted language, control flow—like that of `if` statements and loops (`for`, `while`)—is slow in comparison to compiled languages (C, Java).
- To speed up operations in R:
  - Pre-allocate memory for data structures before filling them up in a loop.
  - Make use of **vectorized** operations that—under the hood—are implemented in C:

```
a <- 1:5 # what if 5 were 1e9? Try it!
b <- 1:5

# Non-vectorized operation (slow)
result1 <- numeric(length = length(a))
for (i in seq_along(a)) {
    result1[i] <- a[i] + b[i]
}
result1 # print output

# Vectorized operation (fast)
result2 <- a + b
result2 # print output
```

  - Make use of the `apply` family of functions: `lapply()`, `sapply()`, `vapply()`

```
df <- tibble(x=1:10, y=100:90)
apply(X=df, MARGIN=2, FUN=mean) # MARGIN=1 operates on rows; 2 operates on col
```

Object-Oriented Programming vs. Multiple Dispatch

- Methods are owned by functions and not by objects
  - *Here, a method is an instance of a function.*

```
Square.getArea(); # typical OOP paradigm
```

```
function area(x) = 2
function area(x::Square) = 2 * 2
```

Julia is a Fast, Dynamically-Typed Language

- A **statically** typed language—like that of C or Java—requires the type of a variable to be known at compile-time.
  - Offers large increases in *performance*, as the compiled code can be heavily optimized when types are explicitly known prior to run-time.
- A **dynamically** typed language—like that of R—checks for types at run-time and are often interpreted languages (no compilation required).
  - Benefit: much more forgiving for programmers in terms of frequency of errors encountered, with the cost being a reduction in performance.

- Julia is dynamically typed, but incredibly fast because of how its type system was designed to be stable. This allows for its compiled code to be heavily optimized.
  - Benchmarks for Julia: https://julialang.org/benchmarks/
- Take a look at the Julia codebase: https://github.com/JuliaLang/julia
  - Generic functions make extending its codebase a simple task, yet highly effective.

5.6 IJulia: Running Julia in a Jupyter Environment

- https://github.com/JuliaLang/IJulia.jl
- In the package mode (enter `]`), enter the following:
  ```
  add IJulia
  ```
- Then, exit the package mode. Initiate the package for use:
  ```
  using IJulia
  notebook()
  ```

6 Practical Assignment

6.1 Wrangle Gene Location Data in R

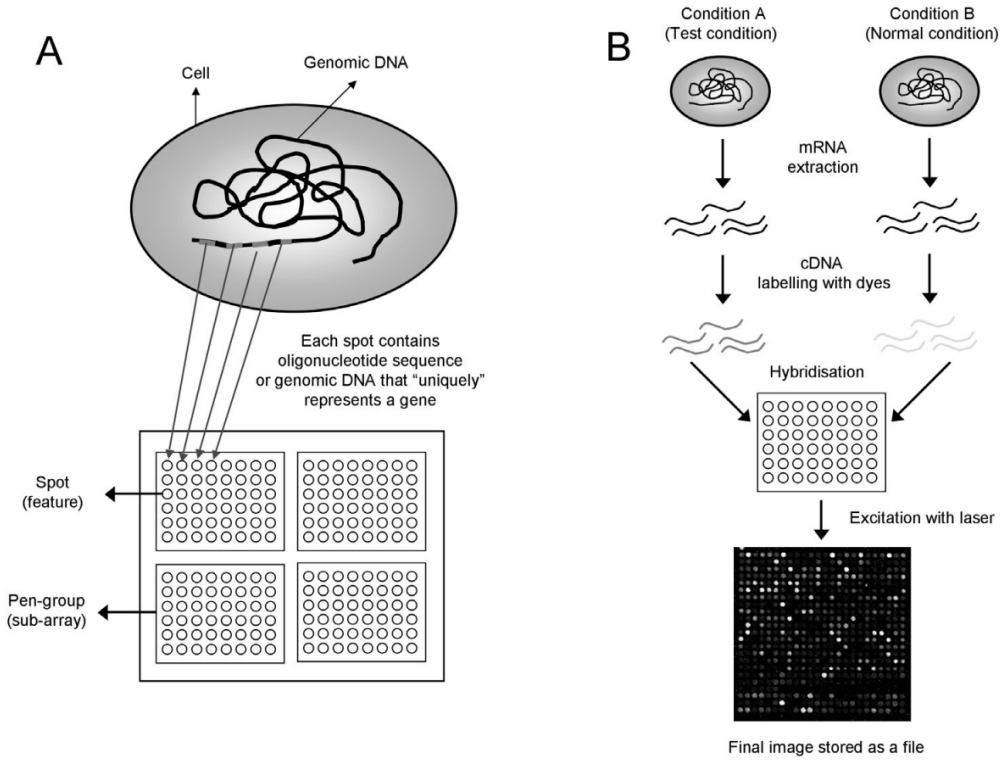


Figure 2: Overview of microarray gene expression experiments[1].

- Obtain chromosome 9 gene locations (gene name, chr, txStart, txEnd).
- UCSC file contains gene locations (transcription start and transcription end sites)

7 Resources

7.1 R

- R for Data Science: https://r4ds.had.co.nz/index.html
  - Specifically: Tidy data and Relational data
- Advanced R, by Hadley Wickham: http://adv-r.had.co.nz/Introduction.html
- Literate programming in R using `knitr`:
  - https://github.com/GreenwoodLab/knitr-tutorial

7.2 Common Errors

XQuartz on macOS

Some R packages require XQuartz to be installed. If you run into this error, download & install a stable version of XQuartz and restart your computer. Alternatively, if you use Homebrew as your package manager, you can run the following from a terminal:

```
brew install --cask xquartz
```

Bibliography

1. M. M. Babu. An introduction to microarray data analysis. Computational Genomics, 2004. URL https://www.mrc-lmb.cam.ac.uk/genomes/madanm/microarray/chapter-final.pdf.