Tidy data, the tidyverse and tidying data

Data Science option of BIO00058M Data Analysis.

Emma Rand
University of York, UK

Outline

The aim of this section is to introduce you to the tidyverse (Wickham Averick, et al., 2019) including the pipe, `%>%` and some commonly applied data tidying operations.

It includes a case study which demonstrates how to problem solve your way to a solution using a one-step-at-a-time approach.

Outline

You should be able to code along with the examples. When you see the film clapper it is ..

🎬 .. an instruction to do something!!

🎬 Start a new project containing directories called scripts, data-raw, and data-processed. Use a new script for example.

What is tidy data?

What is tidy data?

Tidy data adhere to a consistent structure which makes it easier to manipulate, model and visualize them. The structure is defined by:

1. Each variable has its own column.
2. Each observation has its own row.
3. Each value has its own cell.

Closely allied to the relational algebra of relational databases (Codd, 1990).
Underlies the enforced rectangular formatting in SPSS, STATA and R's dataframe.
The term 'tidy data' was popularised by Wickham (2014).

Note: There may be more than one potential tidy structure.

Tidy format

Suppose we had just 3 individuals in each of two populations:

NOT TIDY!

A	B
12.4	12.6
11.2	11.3
11.6	12.1

TIDY!

population	distance
A	12.4
A	11.2
A	11.6
B	12.6
B	11.3
B	12.1

Tidyverse

Tidyverse

The tidyverse (Wickham, 2017) is both a paradigm for coding in R and a metapackage (a collection of packages).

Contributors describe it as "an opinionated" collection of R packages designed for data science.

The key feature is that `tidyverse` packages share an underlying design philosophy, grammar, and data structures.

This means they work well together and learning new tidyverse packages is quick. This consistency is intended to make data work more efficient.

Tidyverse

`tidyverse` packages have a reputation for making code which is easy to read and write for humans, and for the connection of tools together into reproducible workflows.

The R Views article by Joe Rickert gives a good overview.

Tidyverse

The are many other extremely useful packages that are not part of the tidyverse but which also use a common design across packages.

An example is the Bioconductor Project.

However, Bioconductor packages that take a tidyverse approach are beginning to accumulate. For example:

- `tidybulk` (Mangiola, 2020). See also https://stemangiola.github.io/rpharma2020_tidytranscriptomics/articles/ti
- `biobroom` (Bass Robinson, et al., 2020)

Tidyverse

You should already have `tidyverse` installed. Packages need installing only once (unless you wish to update them) but must be loaded every session.

🎬 Load the core tidyverse packages with:

```
library(tidyverse)
```

Core packages are: `ggplot2`, `dplyr`, `tidyr`, `readr`, `purr`, `tibble`, `stringr` and `forcats`

Actually, `ggplot2` predates the tidyverse which is why it uses `+` to link functions together. It does use the same sort of grammar and works very well within the tidyverse.

Tidyverse

The tidyverse also includes many other packages with more specialised usage. They are not loaded automatically with `library(tidyverse)` and need their own `library()` calls.

Examples are `readxl`, `haven`, `rvest` and `lubridate.`

The pipe %>%

The pipe %>%

The pipe `%>%` operator from the `magrittr` package (Bache and Wickham, 2014b) is loaded with core tidyverse.

It is the feature that allows us to connect tools together in a readable way.

It can improve code readability by:

- structuring sequences of data operations left-to-right or top-to-bottom (as opposed to from the inside and out),
- minimizing the need for intermediates,
- making it easy to add steps anywhere in the sequence of operations.

The pipe %>%

The pipe means that instead of using:

```
function(object)
```

We can use

```
object %>% function()
```

This is useful when you have multiple functions to apply.

The pipe %>%

Instead of:

```
function2(function1(object))
```

We can use

```
object %>% function1()  %>% function2()
```

The pipe %>%

As a simple example, suppose we want to apply a log-squareroot transformation[1] to some proportion data.

🎬 Generate a random sample of ten proportions to work with:

```
nums <- sample((1:100)/100, size = 10, replace = FALSE)
```

Two ways we *could* apply the log-squareroot transformation are to:

1. nest the squareroot and log functions
2. create intermediate variables

1. a transformation commonly applied to proportion data to make it less platykurtic $x^t = log(\sqrt{x})$

The pipe %>%

🎬 Nest the `sqrt()` and `log()` functions:

```
tnums <- log(sqrt(nums))
```

Code must read from inside to outside.

Increasingly difficult to read as number of functions increases.

Also makes simple debugging harder.

The pipe %>%

🎬 Create intermediate variables:

```
sqrtnums <- sqrt(nums)
tnums <- log(sqrtnums)
```

Easier to read than nesting.

But you have extra variables you don't need and which become increasingly difficult to name appropriately creating code and workspace clutter.

The pipe %>%

Using the pipe avoids these by taking the output of one operation as the input of the next.

The pipe has long been used by Unix operating systems (where the pipe operator is `|`).

The R pipe operator is `%>%`

🎬 The keyboard short cut is ctrl-shift-M.

The pipe %>%

🎬 Use pipes to code the functions in sequence:

```
tnums <- nums %>%
  sqrt() %>%
  log()
```

This can be read as: take `nums` *and then* squareroot it *and then* log it.

The pipe %>%

More explicitly, this is:

```
tnums <- nums %>%
  sqrt(.) %>%
  log(.)
```

Where `.` stands for the object being passed in.

In most cases, you don't need to include the `.`.

Occasionally you do have to, for example when arguments are optional or there is ambiguity over which argument is meant.

Data tidying tasks

Data tidying tasks

Tidying data includes reshaping it in to 'tidy' format but also other tasks such as:

- renaming variables for consistency
- recoding variables
- cleaning content for consistency with respect to valid values, missing values and NA

👀 Key point!

- Keep the raw data exactly as it came to you and do not alter/edit.
- Script and document all tidying tasks.

Converting "wide" to "long"

Converting "wide" to "long"

Data commonly need to be reshaped from a format with more than one observation per row.

The data given in biomass.txt are taken from an experiment in which the insect pest biomass (g) was measured on plots sprayed with water (control) or one of five different insecticides.

Also in the data file are variables indicating the replicate number and the identity of the tray in which the plant was grown.

Converting "wide" to "long"

🎬 Save a copy of this file to your `data-raw` folder and read it in.

```
file <- "../data-raw/biomass.txt"
biomass <- read_table2(file)
```

`read_table2()` is a function from the tidyverse's `readr` package. It works a lot like base R's `read.table()` but with some useful defaults (e.g., is header assumed) and extra functionality (e.g.,it's faster).

tidyverse import functions treat strings as character variables by default, not factors. Until R 4.x, `read.table()` turned strings into factors. This used to be an important reason for me to use `read_table2()`.

Whilst analysis and visualisation often require factor variables, any processing of strings is made much easier if they are characters.

Converting "wide" to "long"

🎬 View the dataframe.

```
## # A tibble: 10 x 7
##    WaterControl     A     B     C     D     E rep_tray
##           <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
##  1         350. 159.  150.   80.0  267.  350. 1_x
##  2         324. 146.  154.  266.   110.  320. 2_x
##  3         359. 116.   69.5 161.   221.  359. 3_x
##  4         255. 135.  151.  161.   160.  255. 4_y
##  5         208. 137.  213.   51.2  198.  208. 5_y
##  6         326.  81.8 144.  184.   270.  326. 6_y
##  7         295. 116.  150.  176.   224.  295. 7_z
##  8         285. 114.  134.  136.   141.  285. 8_z
##  9         383. 155.  153.  159.   290.  383. 9_z
```

A tibble is the tidyverse's dataframe. In fact, tibbles are dataframes as well as tibbles.

Converting "wide" to "long"

These data are in "wide" format and can be converting to "long" format using the `tidyr` package function `pivot_longer()`.

`pivot_longer()` collects the values from specified columns (`cols`) into a single column (`values_to`) and creates a column to indicate the group (`names_to`).

We want to gather the first 6 columns but the `rep_tray` column contents should be repeated.

Converting "wide" to "long"

🎬 Gather all the columns of biomass except `rep_tray`:

```
biomass2 <- biomass %>%
  pivot_longer(names_to = "spray",
               values_to = "mass",
               cols = -rep_tray)
```

The values will be in a column called `mass`, the treatments in a column called `spray`.

Converting "wide" to "long"

🎬 View the dataframe.

```
## # A tibble: 60 x 3
##    rep_tray spray         mass
##    <chr>    <chr>        <dbl>
##  1 1_x      WaterControl 350.
##  2 1_x      A            159.
##  3 1_x      B            150.
##  4 1_x      C             80.0
##  5 1_x      D            267.
##  6 1_x      E            350.
##  7 2_x      WaterControl 324.
##  8 2_x      A            146.
##  9 2_x      B            154.
```

Converting "wide" to "long"

Extra exercise: Write this dataframe to file to your `processed_data` folder. This was covered in the Project Organisation workshop

See the result: biomass2-base.txt

Extra exercise: Can you find a tidyverse function for writing dataframes.

See the result: biomass2-tidyverse.txt

Splitting column contents

Splitting column contents

We sometimes have single columns which contain more than one type of encoded information.

For example, a column might contain a full species name and you might want to separate the genus and species names to perform a by-genus analysis.

Another example arises when you read in multiple files with names that encode a date, experiment or treatment. Typically, we add the file name to a column in the dataframe before combining the dataframes for analysis and then split the name into columns for the date, experiment of treatment.

Splitting column contents

For the `biomass2` data we might wish to separate the replicate number from the tray identity and put them in two separate columns.

We can do this with a 'regular expression' or regex. A regex defines a pattern for matching text.

It's a big topic and there are many tutorials. I remember a few bits and google "how to match ... regex".

A quick reference

Splitting column contents

The `extract()` function from the `tidyr` helps us achieve this.

We give:

- the names of the new columns we want to create
- the patterns matching the part of the `rep_tray` value we want to go in each column.

Splitting column contents

🎬 Extract parts of `rep_tray` and put in new columns `replicate_number`, `tray_id`:

```
biomass3 <- biomass2 %>%
  extract(rep_tray,
          c("replicate_number", "tray_id"),
          "([0-9]{1,2})\\_([a-z])")
```

- The patterns to save into columns are inside `( )`.
- The pattern going into `replicate_number`, `[0-9]{1,2}`, means 1 or 2 numbers.
- The pattern going into `tray_id`, `[a-z]` means one lowercase letter.
- the bit between the two `( )`, `\_` is a pattern that matches what is in `rep_tray` but is not to be saved.

Case study

Case study

Overivew

You will now work through an example of some real data from The Genever Group. The arrangement and format of these data are typical of many protein and gene expression datasets so the processing is representative of that needed in a variety of situations.

The data are mass spectrometry data of the soluble protein fraction from five immortalised mesenchymal stromal cell (MSC) lines.

The data are normalised protein abundances. Each row is a protein.

Case study

🎬 Save a copy of Y101_Y102_Y201_Y202_Y101-5.csv file to your `data-raw` folder.

🎬 You may wish to view it in excel while reading the information on the next page.

Data description

The cells lines are Y101, Y102, Y201, Y202 and Y101.5 and there are three replicates for each cell line arranged in columns. Also in the file are columns for:

- the protein accession
- the number of peptides used to identify the protein
- the number of unique peptides used to identify the protein
- a measure of confidence in that identification
- the maximum fold change between the mean abundances of two cell lines (i.e., highest mean / lowest mean)
- a p value for a comparison test between the highest mean and lowest mean
- a q value (corrected p value)
- a measure of the power of that test
- the cell line with the highest mean
- the cell line with the lowest mean
- the protein mass
- whether at least two peptides were detected for a protein.

Case study

Data Import

Column names are spread over three rows but are primarily in the third row.

We can read in from the third row by skipping the first two. We can also use the `clean()` function from the `janitor` package to improve the column names.

Case study

Data Import

🎬 Read in the file using `read_csv()` from the `tidyverse`'s `readr` package.

```
# define file name
filesol <- "../data-raw/Y101_Y102_Y201_Y202_Y101-5.csv"

# skip first two lines
sol <- read_csv(filesol, skip = 2) %>%
  janitor::clean_names()
```

Case study

Data Import

👀 the `::` notation gives you access to a package's functions without first using the `library()` command.

This is useful when you want to use a single function from a package, or you need to specify which package when a function name is used in two loaded packages.

Filtering rows

This dataset includes bovine serum proteins from the medium on which the cells were grown which need to be filtered out.

We also filter out proteins for which fewer than 2 peptides were detected since we can not be confident about their identity. This is common practice for such proteomic data.

Case study

Filtering rows

`dplyr` (yep, a tidyverse package) has the `filter()` function. It is easier to use than selection using `datframe[rows, cols]`

🎬 Keep rows of human proteins identified by more than one peptide:

```
sol <- sol %>%
  filter(str_detect(description, "OS=Homo sapiens")) %>%
  filter(x1pep == "x")
```

Case study

👀 `str_detect(string, pattern)` returns a logical vector according to whether 'pattern' is found in 'string'.

👀 Notice that we have applied `filter()` twice using the pipe.

Processing cells contents

It would be useful to extract the genename from the description and put it in a column.

One entry from the description column looks like this:

```
sol$description[1]
```

```
## [1] "Neuroblast differentiation-associated protein AHNAK OS=Homo sapiens GN=AHNAK PE=
```

The genename is after `GN=`. We need to extract the part of the string with the genename and put it in a new column.

Case study

Processing cells contents

A way to problem-solve your way through this is work with *one* value carrying out one operation at a time until you've worked out what to do before implementing on an entire column.

The pipe makes it especially easy to break problems down like this.

Processing cells contents

One step at a time on one value

🎬 Extract the first value of the description to work with:

```
one_description <- sol$description[1]
```

🎬 Extract the part of the string after `GN=` using a regex:

```
str_extract(one_description,"GN=[^\\s]+")
```

```
## [1] "GN=AHNAK"
```

Explanation on next slide.

Case study

Processing cells contents

One step at a time on one value

- `[ ]` means some characters
- `^` means 'not' when inside `[ ]`
- `\s` means white space
- the `\` before is an escape character to indicate that the next character, `\` should not be taken literally (because it's part of `\s`)
- `+` means one or more

So `GN=[^\\s]+` means `GN=` followed by one or more characters that are not whitespace. This means the pattern stops matching at the first white space after "GN=".

Processing cells contents

One step at a time on one value

We're close. Now we will drop the `GN=` part by replacing it with nothing:

🎬 Add replacing `GN=` with an empty string, `""`, to the pipeline:

```
str_extract(one_description, "GN=[^\\s]+") %>%
  str_replace("GN=", "")
```

```
## [1] "AHNAK"
```

Processing cells contents

Creating a new column

Now we know how to get the result for one value, we need to apply the same process to the whole column

`mutate()` is the `dplyr` function that adds new variables and preserves existing ones. It takes `name` = `value` pairs of expressions where:

- `name` is the name for the new variable and
- `value` is the value it takes. This is usually an expression.

Processing cells contents

Creating a new column

🎬 Add a variable `genename` which contains the processed string from the `description` variable:

```
sol <- sol %>%
  mutate(genename =  str_extract(description,"GN=[^\\s]+")
           str_replace("GN=", ""))
```

Case study

You will continue to work on this case study in the workshop.

Summary

- `tidyverse` is a collection of packages with a common design; many excellent packages are not part of it
- `library(tidyverse)` loads the core packages; other `tidyverse` packages need their own `library()` call
- the pipe `%>%` is key to connecting `tidyverse` tools together to create highly readable code
- reshaping data from wide to long format is common: `pivot_longer()`
- splitting cell contents is common: `extract()`
- regular expressions allow you to match text patterns; expect to have to google and do a lot of trial and error
- `clean_names()` is well useful!
- `::` gives you access to a package's functions without using `library()`
- use particular rows `filter()`

Reading

Strongly recommended

- article by Joe Rickert gives a good overview.
- Tidy Data (Wickham, 2014) sections 1 and 2

Further

- Welcome to the Tidyverse (Wickham Averick, et al., 2019)
- Tidy Data (Wickham, 2014) sections 3 - 6
- R for Data Science (Wickham and Grolemund, 2017) Chapter 12.1, to 12.3 and Chapter 18

References

Bache, S. M. and H. Wickham (2014b). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. URL: https://CRAN.R-project.org/package=magrittr.

Bass, A. J, D. G. Robinson, et al. (2020). *biobroom: Turn Bioconductor objects into tidy data frames*. R package version 1.20.0. URL: https://github.com/StoreyLab/biobroom.

Codd, E. F. (1990). *The Relational Model for Database Management: Version 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co, Inc. URL: https://dl.acm.org/doi/pdf/10.5555/77708.

Mangiola, S. (2020). *tidybulk: Brings transcriptomics to the tidyverse*. R package version 1.0.2. URL: https://bioconductor.org/packages/release/bioc/html/tidybulk.html.

References

Wickham, H. (2014). "Tidy Data". In: *Journal of Statistical Software, Articles* 59.10, pp. 1-23. URL: https://vita.had.co.nz/papers/tidy-data.pdf.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1. URL: https://CRAN.R-project.org/package=tidyverse.

Wickham, H, M. Averick, et al. (2019). "Welcome to the Tidyverse". In: *JOSS* 4.43, p. 1686. URL: https://joss.theoj.org/papers/10.21105/joss.01686.

Wickham, H. and G. Grolemund (2017). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st. O'Reilly Media, Inc. ISBN: 1491910399, 9781491910399. URL: https://r4ds.had.co.nz/.

Slides made with with xaringan (Xie, 2019) and xaringanExtra (Aden-Buie, 2020)

Emma Rand

emma.rand@york.ac.uk

Twitter: @er13_r

GitHub: 3mmaRand

blog: https://buzzrbeeline.blog/