Advanced Computing — Data wrangling and plotting*

Stefano Allesina (original author) *University of Chicago*
John Novembre *University of Chicago*

Data wrangling

As biologists living in the XXI century, we are often faced with tons of data, possibly replicated over several organisms, treatments, or locations. We would like to streamline and automate our analysis as much as possible, writing scripts that are easy to read, fast to run, and easy to debug. Base `R` can get the job done, but often the code contains complicated operations (think of the cases in which you used `lapply` only because of its speed), and a lot of `$` signs and brackets.

To start, we need to import `tidyverse`:

```
library(tidyverse)
```

`tidyverse` is a fantastic bundle of packages: a collection of `R` packages designed to manipulate large data frames in a simple and straightforward way. These tools are also much faster than the corresponding base `R` commands, and allow you to write compact code by concatenating commands to build "pipelines". Moreover, all of the packages in the bundle share the same philosophy, and are seamlessly integrated. By default, calling `library(tidyverse)` loads the packages `readr`, `tidyr` and `dplyr` (to read, organize and manipulate data), `ggplot2` (data plotting), `stringr` (string manipulation) and a few others; many others ancillary packages that are part of the `tidyverse` can be loaded if needed.

Then, we need a dataset to play with. We take a dataset containing all the papers published by UofC researchers in *Nature* or *Science* between 1999 and July 2019:

```
pubs <- read.csv("../data/UC_Nat_Sci_1999-2019.csv")
```

A new data type, `tibble`

The data are stored in a `data.frame`:

```
is.data.frame(pubs)
```

`tidyverse` ships with a new data type, called a `tibble`. It also comes with its improved function to read data:

```
pubs <- read_csv("../data/UC_Nat_Sci_1999-2019.csv")
pubs
```

*This document is included as part of the workshop packet for the BSD qBio Bootcamp, University of Chicago, 2022.
Current version: August 16, 2024.

which automatically reads the data as a `tibble`. The nice feature of `tibble` objects is that they will print only what fits on the screen, and also give you useful information on the size of the data, as well as the type of data in each column. Other than that, a `tibble` object behaves very much like a `data.frame`. If you want to transform the `tibble` back into a `data.frame`, use the function `as.data.frame(my_tibble)`; the function `as_tibble(my_data_frame)` transforms a `data.frame` into a tibble.

We can take a look at the data using one of several functions:

- `head(pubs)` shows the first few rows
- `tail(pubs)` shows the last few rows
- `glimpse(pubs)` a summary of the data (similar to `str` in base `R`)
- `View(pubs)` open data in spreadsheet-like window

Selecting rows and columns

There are many ways to subset the data, either by row (subsetting the *observations*), or by column (subsetting the *variables*). For example, let's select only articles published after 2009:

```
filter(pubs, Year > 2009)
```

You can see that 515 of the 953 documents were published in the last 10 years. We have used the command `filter(tbl, conditions)` to select certain observations. We can combine several conditions, by listing them side by side, possibly using logical operators.

Exercise: what does this do?

```
 filter(pubs, Year == 2008, `Source title` == "Nature", `Cited by` > 100)
```

Note that the "back ticks" can be used to type column names that contain spaces and non-standard characters. This is nice, because otherwise the name of the column would need to be altered (as done automatically by `read.csv`, sometimes creating column names that are difficult to interpret or type).

We can also select particular variables using the function `select(tbl, cols to select)`. For example, select only `Authors` and `Title`:

```
select(pubs, Authors, Title)
```

How many years are represented in the data set? We can use the function `distinct(tbl)` to retain only the rows that differ from each other:

```
distinct(select(pubs, Year))
```

Where we first extracted only the column `Year`, and then retained only distinct values.

Other ways to subset observations:

- `sample_n(tbl, howmany, replace = TRUE)` sample `howmany` rows at random with replacement

- `sample_frac(tbl, proportion, replace = FALSE)` sample a certain proportion (e.g. `0.2` for 20%) of rows at random without replacement
- `slice(tbl, 50:100)` extract the rows between `50` and `100`
- `top_n(tbl, 10, Year)` extract the first `10` rows, once ordered by `Year`

More ways to select columns:

- `select(pubs, contains("Cited"))` select all columns containing the word `Cited`
- `select(pubs, -Authors, -Year)` exclude the columns `Authors` and `Year`
- `select(pubs, matches("astring|anotherstring"))` select all columns whose names match a regular expression.

Creating pipelines using %>%

We've been calling nested functions, such as `distinct(select(pubs, ...))`. If you have to add another layer or two, the code would become unreadable. `dplyr` allows you to "un-nest" these functions and create a "pipeline", in which you concatenate commands separated by the special operator %>%. For example:

```
pubs %>% # take a data table
  select(Year) %>% # select a columns
  distinct() # remove duplicates
```

does exactly the same as the command we've run above, but is much more readable. By concatenating many commands, you can create incredibly complex pipelines while retaining readability.

Producing summaries

Sometimes we need to calculate statistics on certain columns. For example, calculate the average number of citations. We can do this using `summarise`:

```
pubs %>% summarise(avg = mean(`Cited by`))
```

which returns a `tibble` object with just the average number of citations. You can combine multiple statistics (use `first`, `last`, `min`, `max`, `n` [count the number of rows], `n_distinct` [count the number of distinct rows], `mean`, `median`, `var`, `sd`, etc.):

```
pubs %>% summarise(avg = mean(`Cited by`),
                   sd = sd(`Cited by`),
                   median = median(`Cited by`))
```

Summaries by group

One of the most useful features of `dplyr` is the ability to produce statistics for the data once subsetted by *groups*. For example, we would like to compute the average number of citations by journal and year:

```
pubs %>%
  group_by(`Source title`, Year) %>%
  summarise(avg = mean(`Cited by`))
```

> **Exercise:** count the number of articles by UofC researchers in *Nature* and *Science* by `Source title` and `Year`.

Ordering the data

To order the data according to one or more variables, use `arrange()`:

```
pubs %>% select(Title, `Cited by`) %>% arrange(`Cited by`)
pubs %>% select(Title, `Cited by`) %>% arrange(desc(`Cited by`))
```

Renaming columns

To rename one or more columns, use `rename()`:

```
pubs %>% rename(Cites = `Cited by`)
```

If you want to retain the new name(s), simply overwrite the object:

```
pubs <- pubs %>% rename(Cites = `Cited by`, Journal = `Source title`)
```

Adding new variables using `mutate`

If you want to add one or more new columns, use the function `mutate`. For example, suppose we want to count the number of authors for each document. Authors are separated by commas (with small errors, but let's disregard that), and therefore a strategy would be to first count the number of commas, and then add 1:

```
pubs <- pubs %>% mutate(Num_authors = str_count(Authors, ",") + 1)
```

use the function `transmute()` to create a new column and drop the original columns. You can also use `mutate` and `transmute` on grouped data.

When writing code, it is good practice to separate the operations by line:

```
# A more complex example: for each paper,
# compute the percentile rank of citations
# compared to other papers of the same year
pubs %>%
  group_by(Year) %>% # group papers according to year
  mutate(pr = percent_rank(Cites)) %>% # compute % rank by Citations
  ungroup() %>% # remove group information
  arrange(Year, desc(pr), Authors) %>% # order by Year then % rank (decreasing)
  head(20) # display first 20 rows
```

in this way, you can easily comment out a part of the pipeline (or add another piece in the middle).

Data plotting

The most salient feature of scientific graphs should be clarity. Each figure should make crystal-clear a) what is being plotted; b) what are the axes; c) what do colors, shapes, and sizes represent; d) the message the figure wants to convey. Each figure is accompanied by a (sometimes long) caption, where the details can be explained further, but the main message should be clear from glancing at the figure (often, figures are the first thing editors and referees look at).

Many scientific publications contain very poor graphics: labels are missing, scales are unintelligible, there is no explanation of some graphical elements. Moreover, some color graphs are impossible to understand if printed in black and white, or difficult to discern for color-blind people (8% of men, 0.5% of women).

Given the effort that you put in your science, you want to ensure that it is well presented and accessible. The investment to master some plotting software will be rewarded by pleasing graphics that convey a clear message.

In this section, we introduce `ggplot2`, a plotting package for `R` This package was developed by Hadley Wickham who contributed many important packages to `R` (including `dplyr`), and who is the force behing `tidyverse`. Unlike many other plotting systems, `ggplot2` is deeply rooted in a "philosophical" vision. The goal is to conceive a grammar for all graphical representation of data. Leland Wilkinson and collaborators proposed The Grammar of Graphics. It follows the idea of a well-formed sentence that is composed of a subject, a predicate, and an object. The Grammar of Graphics likewise aims at describing a well-formed graph by a grammar that captures a very wide range of statistical and scientific graphics. This might be more clear with an example – Take a simple two-dimensional scatterplot. How can we describe it? We have:

- **Data** The data we want to plot.
- **Mapping** What part of the data is associated with a particular visual feature? For example: Which column is associated with the x-axis? Which with the y-axis? Which column corresponds to the shape or the color of the points? In `ggplot2` lingo, these are called *aesthetic mappings* (`aes`).
- **Geometry** Do we want to draw points? Lines? In `ggplot2` we speak of *geometries* (`geom`).
- **Scale** Do we want the sizes and shapes of the points to scale according to some value? Linearly? Logarithmically? Which palette of colors do we want to use?
- **Coordinate** We need to choose a coordinate system (e.g., Cartesian, polar).
- **Faceting** Do we want to produce different panels, partitioning the data according to one (or more) of the variables?

This basic grammar can be extended by adding statistical transformations of the data (e.g., regression, smoothing), multiple layers, adjustment of position (e.g., stack bars instead of plotting them side-by-side), annotations, and so on.

Exactly like in the grammar of a natural language, we can easily change the meaning of a "sentence" by adding or removing parts. Also, it is very easy to completely change the type of geometry if we are moving from say a histogram to a boxplot or a violin plot, as these types of plots are meant to describe one-dimensional distributions. Similarly, we can go from points to lines, chang-

ing one "word" in our code. Finally, the look and feel of the graphs is controlled by a theming system, separating the content from the presentation.

Basic `ggplot2`

`ggplot2` ships with a simplified graphing function, called `qplot`. In this introduction we are not going to use it, and we concentrate instead on the function `ggplot`, which gives you complete control over your plotting. First, we need to load the package (note that `ggplot2` is automatically loaded by `tidyverse`). While we are at it, let's also load a package extending its theming system:

```
library(ggplot2)
library(ggthemes)
```

A particularity of `ggplot2` is that it accepts exclusively data organized in tables (a `data.frame` or a `tibble` object). Thus, all of your data needs to be converted into a table format for plotting.

For our first plot, we're going to produce a barplot showing the number of papers in `Science` and `Nature` by UofC researcher for each `Year`. To start:

```
ggplot(data = pubs)
```

As you can see, nothing is drawn: we need to specify what we would like to associate to the *x* axis (i.e., we want to set the *aesthetic mappings*):

```
ggplot(data = pubs) + aes(x = Year)
```

Note that we concatenate pieces of our "sentence" using the + sign! We've got the axes, but still no graph... we need to specify a geometry. Let's use barplot:

```
ggplot(data = pubs) + aes(x = Year) + geom_bar()
```

As you can see, we wrote a well-formed sentence, composed of **data** + **mapping** + **geometry**, and this has produced a well-formed plot. We can add other mappings, for example, showing the journal in which the paper was published:

```
ggplot(data = pubs) + aes(x = Year, fill = Journal) + geom_bar()
```

Scatterplots

Using `ggplot2`, one can produce very many types of graphs. The package works very well for 2D graphs (or 3D rendered in two dimensions), while it lack capabilities to draw proper 3D graphs, or networks.

The main feature of `ggplot2` is that you can tinker with your graph fairly easily, and with a common grammar. You don't have to settle on a certain presentation of the data until you're ready, and it is very easy to switch from one type of graph to another.

For example, let's plot the number of citations in the `y` axis, the year in the `x` axis. We want a scatterplot, which is produced by the geometry `geom_point`:

```
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = Cites) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

This does not look very good, because some papers have a much larger number of citations than other. We can attempt plotting the `log(Cites + 1)` instead (the `+1` is added because some papers might have 0 citations):

```
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = log(Cites + 1)) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

Much nicer! Now we can add a smoother by typing:

```
pl + geom_smooth() # spline by default
pl + geom_smooth(method = "lm", se = FALSE) # linear model, no standard errors
```

> **Exercise:** repeat the plot of the citations, but showing a different `colour` for each journal; add a smoother for each journal separately. Do papers receive more citations when they're published in *Nature* or *Science*?

Histograms, density and boxplots

What is the distribution of citations?

```
ggplot(data = pubs) + aes(x = Cites) + geom_histogram()
```

You can see that there are some papers with many more citations than others. Try log-transforming the data:

```
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_histogram()
```

Now we observe an histogram much closer to a Normal distribution, meaning that the number of citations is approximately log-normally distributed. You can switch to a density plot quite easily (just change the geometry!):

```
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_density()
```

Similarly, we can produce boxplots, for example showing the number of citations for papers in *Nature* and *Science* (log transformed):

```
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_boxplot()
```

It is very easy to change geometry, for example switching to a violin plot:

```
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_violin()
```

Exercise:

- Produce a boxplot showing the number of authors (in log) per year (use `factor(Year)` for the x axis). Is science becoming more collaborative?
- Now produce a scatterplot showing the same trend, and add a smoothing function.

Scales

We can use scales to determine how the aesthetic mappings are displayed. For example, we could set the x axis to be in logarithmic scale, or we can choose how the colors, shapes and sizes are used. `ggplot2` uses two types of scales: `continuous` scales are used for continuos variables (e.g., real numbers); `discrete` scales for variables that can only take a certain number of values (e.g., treatments, labels, factors, etc.).

For example, let's plot a histogram showing the number of authors per paper:

```
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() # no transformation
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "log") # natural log
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "log10") # base 10 log
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "sqrt", name = "Number of authors")
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() + scale_x_log10() # shorthand
```

We can use different color scales. For example:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = Num_authors, y = Cites, colour = factor(Year)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
pl + scale_colour_brewer()
pl + scale_colour_brewer(palette = "Spectral")
pl + scale_colour_brewer(palette = "Set1")
pl + scale_colour_brewer("year of publication", palette = "Paired")
```

Or use the number of authors a continuous variable:

```
pl <- ggplot(data = pubs) +
  aes(x = Year, y = log(Cites + 1), colour = log(Num_authors)) +
  geom_point()
pl + scale_colour_gradient()
pl + scale_colour_gradient(low = "red", high = "green")
pl + scale_colour_gradientn(colours = c("blue", "white", "red"))
```

Similarly, you can use scales to modify the display of the shapes of the points (`scale_shape_contintuous`, `scale_shape_discrete`), their size (`scale_size_continuous`, `scale_size_discrete`), etc. To set values manually (useful typically for discrete scales of colors or shapes), use `scale_colour_manual`, `scale_shape_manual` etc.

Themes

Themes allow you to manipulate the look and feel of a graph with just one command. The package `ggthemes` extends the themes collection of `ggplot2` considerably. For example:

```
library(ggthemes)
pl + theme_bw() # white background
pl + theme_economist() # like in the magazine "The Economist"
pl + theme_wsj() # like "The Wall Street Journal"
```

Faceting

In many cases, we would like to produce a multi-panel graph, in which each panel shows the data for a certain combination of parameters. In `ggplot` this is called *faceting*: the command `facet_grid` is used when you want to produce a grid of panels, in which all the panels in the same row (column) have axis-ranges in common; `facet_wrap` is used when the different panels do not have axis-ranges in common.

For example:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = log10(Cites + 1)) +
  geom_histogram()
show(pl)
pl + facet_grid(~Year) # in the same row
pl + facet_grid(Year~.) # col
pl + facet_grid(Journal ~ Year) # two facet variables
pl + facet_wrap(Journal ~ Year, scales = "free") # just wrap around
```

Setting features

Often, you want to simply set a feature (e.g., the color of the points, or their shape), rather than using it to display information (i.e., mapping some aestethic). In such cases, simply declare the feature outside the `aes`:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = log10(Num_authors))
pl + geom_histogram()
pl + geom_histogram(colour = "red", fill = "lightblue")
```

Saving graphs

You can either save graphs as done normally in R:

```
# save to pdf format
pdf("my_output.pdf", width = 6, height = 4)
print(my_plot)
dev.off()
# save to svg format
svg("my_output.svg", width = 6, height = 4)
print(my_plot)
dev.off()
```

or use the function ggsave

```
# save current graph
ggsave("my_output.pdf")
# save a graph stored in ggplot object
ggsave(plot = my_plot, filename = "my_output.svg")
```

Multiple layers

Finally, you can overlay different data sets, using different geometries. For example, suppose that we have two data sets: one for papers with few authors (say <10) and one for large collaborations:

```
small_collab <- pubs %>% filter(Num_authors < 10)
large_collab <- pubs %>% filter(Num_authors >= 10)
```

We can overlay different geometries for the same data set:

```
ggplot(data = small_collab) +
  aes(x = factor(Num_authors), y = log(Cites + 1)) +
  geom_boxplot(fill = "lightblue") +
  geom_violin(fill = "NA") +
  geom_point(alpha = 0.25) # alpha stands for transparency
```

Or combine different data sets (with the same aes!):

```
ggplot(data = small_collab) +
  aes(x = Year) +
  geom_bar(fill = "red", alpha = 0.5) +
  geom_bar(data = large_collab, fill = "blue", alpha = 0.5)
```

Tidying up data

The best way to organize data for plotting and computing is the *tidy form*, meaning that a) each variable has its own column, and b) each observation has its own row. When data are not in tidy form, you can use the package `tidyr` to reshape them.

For example, suppose we want to produce a table in which for each journal and year, we report the average number of authors. First, we need to compute the values:

```
avg_authors <- pubs %>%
  group_by(Journal, Year) %>%
  summarise(avg_au = mean(Num_authors))
```

This table is in tidy format (also called "narrow" format); we want to create columns for each journal, and report the average in the corresponding cell. To do so, we "spread" the journals into columns:

```
avg_authors <- avg_authors %>% spread(Journal, avg_au)
```

Note that this is not in tidy form, as two observations are in the same row (also called "messy" or "wide" format). While this is not ideal for computing, it is great for human consumption, as we can easily compare the two numbers in the same row.

If we want to go back to tidy form, we can "gather" the column names, and return to tidy:

```
# gather(where to store col names,
#        where to store values,
#        which columns to gather)
avg_authors %>% gather(Journal, Average_num_authors, 2:3)
# alternatively, if it's cleaner
avg_authors %>% gather(Journal, Average_num_authors, -Year)
```

Joining tables

If you have multiple data frames or `tibble` objects with shared columns, it is easy to join them (as in a database). To showcase this, we are going to extract papers by very prolific authors. First, we want to compute how many papers are in the data for each "author" (actually, last-name initial combinations, which might represent different authors with common names...). First, we need a data set in which the authors have been separated:

```
by_author <- pubs %>%
  select(Authors, Title) %>%
  separate_rows(sep = ", ", Authors) %>%
  rename(Focal_author = Authors)
```

Now we can count the number of appearances of each name:

```
by_author <- by_author %>%
  group_by(Focal_author) %>%
  mutate(Tot = n())
```

Where we have created a new column (`Tot`) by calling `mutate` on grouped data. Who are the authors most represented in the data?

```
tot_author <- by_author %>%
  select(Focal_author, Tot) %>%
  distinct() %>%
  arrange(desc(Tot))
```

You can see that common Chinese name combinations are in the top few rows (meaning that probably we conflated several authors...). Let's plot an histogram:

```
tot_author %>% ggplot() + aes(x = Tot) + geom_histogram() + scale_y_log10()
```

As you can see, the vast majority of authors appears only once, and very few, appear 15 or more times. We want to extract the papers of the most prolific authors from the data that we have stored in `pubs`. For example, we want to consider authors that are represented 10 or more times in these papers. To do so, we first extract the prolific authors:

```
prolific <- by_author %>% filter(Tot >= 10)
```

and now we can join `pubs` and `prolific`. By calling `inner_join`, only rows that are present in both tables will be retained; because the two tables share a column (`Title`), `dplyr` can proceed automatically:

```
pubs %>% inner_join(prolific)
```

We can use this table to compute the number of citations received by each prolific author:

```
pubs %>% inner_join(prolific) %>% ggplot() +
  aes(x = Focal_author, y = Cites) +
  geom_col() + # similar to bar plot
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) # rotate labels
```

Besides `inner_join(x, y)`, you can use:

- `left_join(x, y)`: return all rows from `x`, and all columns from `x` and `y` (those with no match will show `NA`);
- `right_join(x. y)`: return all rows from `y`, and all columns from `x` and `y`;
- `full_join(x, y)`: return all rows and all columns from both `x` and `y`. Where there are not matching values, returns `NA` for the one missing;
- `anti_join(x, y)`: return all rows from `x` where there are not matching values in `y`.

Exercise in groups

Chicago's Divvy bike-share system (the light blue bikes you will see around town) collects data on all its rides and shares them publically, after anonymizing the rider info (https://www.divvybikes.com/system-data). The file `data/202207-divvy-tripdata.csv` contains a list of all the Divvy bike rides taken in Chicago in July 2022. Form small groups and work on the following exercises. Hint: use the package `lubridate` to work with days, dates, time:

- **Ride map** write a function that takes as input a calendar date (YYYY-MM-DD), and draws a map of all the starting points of rides. Mark a point for each occurrence using the starting latitude and longitude `start_lat` and `start_lng`. Set the `alpha` to something like 0.1 to show brighter colors in areas with many occurrences. Use color to indicate member type or ride type (Optional: Add a feature that draws a line from the starting location to end location)
- **Daily usage profile for the month**: Make a bar plot of the number of rides per day across the 31 days of the month. Produce a facet graph that stratifies the results by member type (the `member_casual` field). Also use the `fill` to denote the `rideable_type`.
- **Busiest hour and day of the week** on which day of the week do most rides to start? On which hour of the day do most rides start? (Extract day of the week from `started_at` using the `wday` function of `lubridate`) Make a plot of the number of rides per hour of the day, faceted on the day of the week.
- **Ride length classes** add a new column to the dataset specifying whether the ride is considered short (<5 minutes), medium (5-30 minutes), or long (>30 minutes) (Hint: again use the package `lubridate` to work with days, dates, time and extract duration from in the `started_at` and `ended_at` fields)
- **Number of rides of different lengths by day of the week** plot the number of rides against the day of the week, faceting by ride length class.
- **Rides by neighborhood** write a function that takes as input a given day, and produces a histogram of the number of rides per neighborhood on that day (i.e. x-axis is number of rides, y-axis is number of neighborhoods). A table relating `starting_station_id` to neighborhood (where neighborhood is assigned using the Google Maps API) is found in the file `data/202207-divvy-station_id_neighborhoods.csv`. You will need to `join` the tables before plotting. What is the mean number of rides per neighborhood? What is the max number? What are the top 5 "neighborhoods" for Divvy rides.
- **Interactions**: Create a table with the starting station, ending station, and the number of rides between each station. What is a feature of the most frequent trips that you observe? (Advanced: Create a graph showing edges connecting stations where the edge color represents the number of trips taken)
- **Miscellaneous**: if you'd like to play more, the `spDistsN1` function of the sp library returns distances as a function of latitude/longitude pairs.