Biological Data Science with R

Stephen D. Turner

2018-12-18

Table of contents

Preface

This book was written as a companion to a series of courses introducing the essentials of biological data science with R. While this book was written with the accompanying live instruction in mind, this book can be used as a self-contained self study guide for quickly learning the essentials need to get started with R. The BDSR book and accompanying course introduces methods, tools, and software for reproducibly managing, manipulating, analyzing, and visualizing large-scale biological data using the R statistical computing environment. This book also covers essential statistical analysis, and advanced topics including survival analysis, predictive modeling, forecasting, and text mining.

This is not a *"Tool X"* or *"Software Y"* book. I want you to take away from this book and accompanying course the ability to use an extremely powerful scientific computing environment (R) to do many of the things that you'll do *across study designs and disciplines* – managing, manipulating, visualizing, and analyzing large, sometimes high-dimensional data. Regardless of your specific discipline you'll need the same computational know-how and data literacy to do the same kinds of basic tasks in each. This book might show you how to use specific tools here and there (e.g., DESeq2 for RNA-seq analysis (Love, Huber, and Anders 2014), ggtree for drawing phylogenetic trees (Yu et al. 2017), etc.), but these are not important – you probably won't be using the same specific software or methods 10 years from now, but you'll still use the same underlying data and computational foundation. That is the point of this series – to arm you with a basic foundation, and more importantly, to enable you to figure out how to use *this tool* or *that tool* on your own, when you need to.

This is not a statistics book. There is a short lesson on essential statistics using R in Chapter 7 but this short chapter offers neither a comprehensive background on underlying theory nor in-depth coverage of implementation strategies using R. Some general knowledge of statistics and study design is helpful, but isn't required for going through this book or taking the accompanying course.

There are no prerequisites to this book or the accompanying course. However, each chapter involves lots of hands-on practice coding, and you'll need to download and install required softwar and download required data. See the setup instructions in Appendix A.

Acknowledgements

This book is partially adapted from material we developed for the University of Virginia BIMS8382 graduate course . The material for this course was adapted from and/or inspired by Jenny Bryan's STAT545 course at UBC (Bryan 2019), Software Carpentry (Wilson 2014) and Data Carpentry (Teal et al. 2015) courses, David Robinson's *Variance Explained* blog (Robinson 2015), the ggtree vignettes (Yu 2022) *Tidy Text Mining with R* (Silge and Robinson 2017), and likely many others.

Part I

Core lessons

1 Basics

This chapter introduces the R environment and some of the most basic functionality aspects of R that are used through the remainder of the book. This section assumes little to no experience with statistical computing with R. This chapter introduces the very basic functionality in R, including variables, functions, and importing/inspecting data frames (tibbles).

1.1 RStudio

Let's start by learning about RStudio. **R** is the underlying statistical computing environment. **RStudio** is a graphical integrated development environment (IDE) that makes using R much easier.

- **Options:** First, let's change a few options. We'll only have to do this once. Under *Tools… Global Options…*:
  - Under *General*: Uncheck "Restore most recently opened project at startup"
  - Under *General*: Uncheck "Restore .RData into workspace at startup"
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *R Markdown*: Uncheck "Show output inline for all R Markdown documents"
- Projects: first, start a new project in a new folder somewhere easy to remember. When we start reading in data it'll be important that the *code and the data are in the same place.* Creating a project creates an Rproj file that opens R running *in that folder.* This way, when you want to read in dataset *whatever.txt*, you just tell it the filename rather than a full path. This is critical for reproducibility, and we'll talk about that more later.
- Code that you type into the console is code that R executes. From here forward we will use the editor window to write a script that we can save to a file and run it again whenever we want to. We usually give it a `.R` extension, but it's just a plain text file. If you want to send commands from your editor to the console, use `CMD+Enter` (`Ctrl+Enter` on Windows).
- Anything after a `#` sign is a comment. Use them liberally to *comment your code.*

1.2 Basic operations

R can be used as a glorified calculator. Try typing this in directly into the console. Make sure you're typing into into the editor, not the console, and save your script. Use the run button, or press `CMD+Enter` (`Ctrl+Enter` on Windows).

```
2+2
5*4
2^3
```

R Knows order of operations and scientific notation.

```
2+3*4/(5+3)*15/2^2+3*4^2
5e4
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line. The keyboard shortcut for inserting the `<-` operator is `Alt-dash`.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```r
2.2 * weight_kg
```

We can also change a variable's value by assigning it a new one:

```r
weight_kg <- 57.5
2.2 * weight_kg
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```r
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```r
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```r
ls()
rm(weight_lb, weight_kg)
ls()
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

Exercise 1

What are the values after each statement in the following?

```r
mass <- 50              # mass?
age  <- 30              # age?
mass <- mass * 2        # mass?
age  <- age - 10        # age?
mass_index <- mass/age  # massIndex?
```

1.3 Functions

R has built-in functions.

```r
# Notice that this is a comment.
# Anything behind a # is "commented out" and is not run.
sqrt(144)
log(1000)
```

Get help by typing a question mark in front of the function's name, or `help(functionname)`:

```r
help(log)
?log
```

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```r
log(1000)
log(1000, base=10)
log(1000, 10)
sqrt(log(1000, base=10))
```

Exercise 2

See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be `2`.

1.4 Tibbles (data frames)

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. We are going to skip straight to the data structure you'll probably use most – the **tibble** (also known as the data frame). We use tibbles to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

We'll learn more about tibbles in Chapter 2.

2 Tibbles

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. Let's skip straight to the data structure you'll probably use most – the **data frame**. We use data frames to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

This lesson assumes a basic familiarity with R (see Chapter 1).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

2.1 Our data

The data we're going to look at is cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.

2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

2.2 Reading in data

2.2.1 dplyr and readr

There are some built-in functions for reading in data in text files. These functions are *read-dot-something* – for example, `read.csv()` reads in comma-delimited text data; `read.delim()` reads in tab-delimited text, etc. We're going to read in data a little bit differently here using the readr package. When you load the readr package, you'll have access to very similar looking functions, named *read-underscore-something* – e.g., `read_csv()`. You have to have the readr package installed to access these functions. Compared to the base functions, they're *much* faster, they're good at guessing the types of data in the columns, they don't do some of the other silly things that the base functions do. We're going to use another package later on called dplyr, and if you have the dplyr package loaded as well, and you read in the data with readr, the data will display nicely.

First let's load those packages.

```
library(readr)
library(dplyr)
```

If you see a warning that looks like this: `Error in library(packageName) : there is no package called 'packageName'`, then you don't have the package installed correctly. See the setup chapter (Appendix A).

2.2.2 `read_csv()`

Now, let's actually load the data. You can get help for the import function with `?read_csv`. When we load data we assign it to a variable just like any other, and we can choose a name for that data. Since we're going to be referring to this data a lot, let's give it a short easy name to type. I'm going to call it `ydat`. Once we've loaded it we can type the name of the object itself (`ydat`) to see it printed to the screen.

```
ydat <- read_csv(file="data/brauer2007_tidy.csv")
ydat
```

Take a look at that output. The nice thing about loading dplyr and reading in data with readr is that data frames are displayed in a much more friendly way. This dataset has nearly 200,000 rows and 7 columns. When you import data this way and try to display the object in the console, instead of trying to display all 200,000 rows, you'll only see about 10 by default. Also, if you have so many columns that the data would wrap off the edge of your screen, those columns will not be displayed, but you'll see at the bottom of the output which, if any, columns were hidden from view. If you want to see the whole dataset, there are two ways to do this. First, you can click on the name of the data.frame in the **Environment** panel in RStudio. Or you could use the `View()` function (*with a capital V*).

```
View(ydat)
```

2.3 Inspecting data.frame objects

2.3.1 Built-in functions

There are several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first few rows
  - `tail()`: shows the last few rows
- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `str()`: structure of the object and information about the class, length and content of each column
  - `summary()`: works differently depending on what kind of object you pass to it. Passing a data frame to the `summary()` function prints out useful summary statistics about numeric column (min, max, median, mean, etc.)

```r
head(ydat)
tail(ydat)
dim(ydat)
names(ydat)
str(ydat)
summary(ydat)
```

2.3.2 Other packages

The `glimpse()` function is available once you load the **dplyr** library, and it's like `str()` but its display is a little bit better.

```r
glimpse(ydat)
```

The **skimr** package has a nice function, skim, that provides summary statistics the user can skim quickly to understand your data. You can install it with `install.packages("skimr")` if you don't have it already.

```r
library(skimr)
skim(ydat)
```

2.4 Accessing variables & subsetting data frames

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out all the gene names in the data. Then let's calculate the average expression across all conditions, all genes (using the built-in `mean()` function).

```r
# display all gene symbols
ydat$symbol

#mean expression
mean(ydat$expression)
```

Now that's not too interesting. This is the average gene expression across all genes, across all conditions. The data is actually scaled/centered around zero:

We might be interested in the average expression of genes with a particular biological function, and how that changes over different growth rates restricted by particular nutrients. This is the kind of thing we're going to do in the next section.

Exercise 1

1. What's the standard deviation expression (hint: get help on the `sd` function with `?sd`).
2. What's the range of rate represented in the data? (hint: `range()`).

2.5 BONUS: Preview to advanced manipulation

What if we wanted show the mean expression, standard deviation, and correlation between growth rate and expression, separately for each limiting nutrient, separately for each gene, for all genes involved in the leucine biosynthesis pathway?

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient, symbol) |>
  summarize(mean=mean(expression), sd=sd(expression), r=cor(rate, expression))
```

Neat eh? We'll learn how to do that in the advanced manipulation with dplyr lesson.

3 Data Manipulation

Data analysis involves a large amount of janitor work – munging and cleaning data to facilitate downstream data analysis. This lesson demonstrates techniques for advanced data manipulation and analysis with the split-apply-combine strategy. We will use the dplyr package in R to effectively manipulate and conditionally compute summary statistics over subsets of a "big" dataset containing many observations.

This lesson assumes a basic familiarity with R (Chapter 1) and data frames (Chapter 2).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

3.1 Review

3.1.1 Our data

We're going to use the yeast gene expression dataset described on the data frames lesson in Chapter 2. This is a cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also

found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.
2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

3.1.2 Reading in data

We need to load both the dplyr and readr packages for efficiently reading in and displaying this data. We're also going to use many other functions from the dplyr package. Make sure you have these packages installed as described on the setup chapter (Appendix A).

```
# Load packages
library(readr)
library(dplyr)

# Read in data
ydat <- read_csv(file="data/brauer2007_tidy.csv")

# Display the data
ydat

# Optionally, bring up the data in a viewer window
# View(ydat)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                      mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport   mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*         t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
```

```
10 VPS5   YOR069W        Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

3.2 The dplyr package

The dplyr package is a relatively new R package that makes data manipulation fast and easy. It imports functionality from another package called magrittr that allows you to chain commands together into a pipeline that will completely change the way you write R code such that you're writing code the way you're thinking about the problem.

When you read in data with the readr package (`read_csv()`) and you had the dplyr package loaded already, the data frame takes on this "special" class of data frames called a `tbl` (pronounced "tibble"), which you can see with `class(ydat)`. If you have other "regular" data frames in your workspace, the `as_tibble()` function will convert it into the special dplyr `tbl` that displays nicely (e.g.: `iris <- as_tibble(iris)`). You don't have to turn all your data frame objects into tibbles, but it does make working with large datasets a bit easier.

You can read more about tibbles in Tibbles chapter in *R for Data Science* or in the tibbles vignette. They keep most of the features of data frames, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors). You can read more about the differences between data frames and tibbles in this section of the tibbles vignette, but the major convenience for us concerns **printing** (aka displaying) a tibble to the screen. When you print (i.e., display) a tibble, it only shows the first 10 rows and all the columns that fit on one screen. It also prints an abbreviated description of the column type. You can control the default appearance with options:

- `options(tibble.print_max = n, tibble.print_min = m)`: if there are more than n rows, print only the first m rows. Use `options(tibble.print_max = Inf)` to always show all rows.
- `options(tibble.width = Inf)` will always print all columns, regardless of the width of the screen.

3.3 dplyr verbs

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do. Here are some of the *single-table* verbs we'll be working with in this lesson (single-table meaning that they only work on a single table – contrast that to *two-table* verbs used for joining data together, which we'll cover in a later lesson).

1. `filter()`

2. `select()`
3. `mutate()`
4. `arrange()`
5. `summarize()`
6. `group_by()`

They all take a data frame or tibble as their input for the first argument, and they all return a data frame or tibble as output.

3.3.1 filter()

If you want to filter **rows** of the data where some condition is true, use the `filter()` function.

1. The first argument is the data frame you want to filter, e.g. `filter(mydata, ....`
2. The second argument is a condition you must satisfy, e.g. `filter(ydat, symbol == "LEU1")`. If you want to satisfy *all* of multiple conditions, you can use the "and" operator, `&`. The "or" operator `|` (the pipe character, usually shift-backslash) will return a subset that meet *any* of the conditions.

- `==`: Equal to
- `!=`: Not equal to
- `>`, `>=`: Greater than, greater than or equal to
- `<`, `<=`: Less than, less than or equal to

Let's try it out. For this to work you have to have already loaded the dplyr package. Let's take a look at LEU1, a gene involved in leucine synthesis.

```r
# First, make sure you've loaded the dplyr package
library(dplyr)

# Look at a single gene involved in leucine synthesis pathway
filter(ydat, symbol == "LEU1")

# Optionally, bring that result up in a View window
# View(filter(ydat, symbol == "LEU1"))

# Look at multiple genes
filter(ydat, symbol=="LEU1" | symbol=="ADH2")

# Look at LEU1 expression at a low growth rate due to nutrient depletion
# Notice how LEU1 is highly upregulated when leucine is depleted!
filter(ydat, symbol=="LEU1" & rate==.05)
```

```r
# But expression goes back down when the growth/nutrient restriction is relaxed
filter(ydat, symbol=="LEU1" & rate==.3)

# Show only stats for LEU1 and Leucine depletion.
# LEU1 expression starts off high and drops
filter(ydat, symbol=="LEU1" & nutrient=="Leucine")

# What about LEU1 expression with other nutrients being depleted?
filter(ydat, symbol=="LEU1" & nutrient=="Glucose")
```

Let's look at this graphically. Don't worry about what these commands are doing just yet - we'll cover that later on when we talk about ggplot2. Here's I'm taking the filtered dataset containing just expression estimates for LEU1 where I have 36 rows (one for each of 6 nutrients × 6 growth rates), and I'm *piping* that dataset to the plotting function, where I'm plotting rate on the x-axis, expression on the y-axis, mapping the value of nutrient to the color, and using a line plot to display the data.

```r
library(ggplot2)
filter(ydat, symbol=="LEU1") |>
  ggplot(aes(rate, expression, colour=nutrient)) + geom_line(lwd=1.5)
```

Look closely at that! LEU1 is *highly expressed* when starved of leucine because the cell has to synthesize its own! And as the amount of leucine in the environment (the growth *rate*) increases, the cell can worry less about synthesizing leucine, so LEU1 expression goes back down. Consequently the cell can devote more energy into other functions, and we see other genes' expression very slightly raising.

Exercise 1

1. Display the data where the gene ontology biological process (the `bp` variable) is "leucine biosynthesis" (case-sensitive) *and* the limiting nutrient was Leucine. (Answer should return a 24-by-7 data frame – 4 genes × 6 growth rates).
2. Gene/rate combinations had high expression (in the top 1% of expressed genes)? *Hint:* see `?quantile` and try `quantile(ydat$expression, probs=.99)` to see the expression value which is higher than 99% of all the data, then `filter()` based on that. Try wrapping your answer with a `View()` function so you can see the whole thing. What does it look like those genes are doing? Answer should return a 1971-by-7 data frame.

3.3.1.1 Aside: Writing Data to File

What we've done up to this point is read in data from a file (`read_csv(...)`), and assigning that to an object in our *workspace* (`ydat <- ...`). When we run operations like `filter()` on our data, consider two things:

1. The `ydat` object in our workspace is not being modified directly. That is, we can `filter(ydat, ...)`, and a result is returned to the screen, but `ydat` remains the same. This effect is similar to what we demonstrated in our first session.

```
# Assign the value '50' to the weight object.
weight <- 50

# Print out weight to the screen (50)
weight

# What's the value of weight plus 10?
weight + 10

# Weight is still 50
weight

# Weight is only modified if we *reassign* weight to the modified value
weight <- weight+10
# Weight is now 60
weight
```

2. More importantly, the *data file on disk* (`data/brauer2007_tidy.csv`) is *never* modified. No matter what we do to ydat, the file is never modified. If we want to *save* the result of an operation to a file on disk, we can assign the result of an operation to an object, and `write_csv` that object to disk. See the help for `?write_csv` (note, `write_csv()` with an underscore is part of the **readr** package – not to be confused with the built-in `write.csv()` function).

```
# What's the result of this filter operation?
filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Assign the result to a new object
leudat <- filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Write that out to disk
write_csv(leudat, "leucinedata.csv")
```

Note that this is different than saving your *entire workspace to an Rdata file*, which would contain all the objects we've created (weight, ydat, leudat, etc).

3.3.2 select()

The `filter()` function allows you to return only certain *rows* matching a condition. The `select()` function returns only certain *columns*. The first argument is the data, and subsequent arguments are the columns you want.

```
# Select just the symbol and systematic_name
select(ydat, symbol, systematic_name)

# Alternatively, just remove columns. Remove the bp and mf columns.
select(ydat, -bp, -mf)

# Notice that the original data doesn't change!
ydat
```

Notice above how the original data doesn't change. We're selecting out only certain columns of interest and throwing away columns we don't care about. If we wanted to *keep* this data, we would need to *reassign* the result of the `select()` operation to a new object. Let's make a new object called `nogo` that does not contain the GO annotations. Notice again how the original data is unchanged.

```
# create a new dataset without the go annotations.
nogo <- select(ydat, -bp, -mf)
nogo

# we could filter this new dataset
filter(nogo, symbol=="LEU1" & rate==.05)

# Notice how the original data is unchanged - still have all 7 columns
ydat
```

3.3.3 mutate()

The `mutate()` function adds new columns to the data. Remember, it doesn't actually modify the data frame you're operating on, and the result is transient unless you assign it to a new object or reassign it back to itself (generally, not always a good practice).

The expression level reported here is the log_2 of the sample signal divided by the signal in the reference channel, where the reference RNA for all samples was taken from the glucose-limited chemostat grown at a dilution rate of 0.25 h^{-1}. Let's mutate this data to add a new variable called "signal" that's the actual raw signal ratio instead of the log-transformed signal.

```r
mutate(nogo, signal=2^expression)
```

Mutate has a nice little feature too in that it's "lazy." You can mutate and add one variable, then continue mutating to add more variables based on that variable. Let's make another column that's the square root of the signal ratio.

```r
mutate(nogo, signal=2^expression, sigsr=sqrt(signal))
```

Again, don't worry about the code here to make the plot – we'll learn about this later. Why do you think we log-transform the data prior to analysis?

```r
library(tidyr)
mutate(nogo, signal=2^expression, sigsr=sqrt(signal)) |>
  gather(unit, value, expression:sigsr) |>
  ggplot(aes(value)) + geom_histogram(bins=100) + facet_wrap(~unit, scales="free")
```

3.3.4 arrange()

The `arrange()` function does what it sounds like. It takes a data frame or tbl and arranges (or sorts) by column(s) of interest. The first argument is the data, and subsequent arguments are columns to sort on. Use the `desc()` function to arrange by descending.

```r
# arrange by gene symbol
arrange(ydat, symbol)

# arrange by expression (default: increasing)
arrange(ydat, expression)

# arrange by decreasing expression
arrange(ydat, desc(expression))
```

Exercise 2

1. First, re-run the command you used above to filter the data for genes involved in the "leucine biosynthesis" biological process *and* where the limiting nutrient is Leucine.

2. Wrap this entire filtered result with a call to `arrange()` where you'll arrange the result of #1 by the gene symbol.
3. Wrap this entire result in a `View()` statement so you can see the entire result.

3.3.5 summarize()

The `summarize()` function summarizes multiple values to a single value. On its own the `summarize()` function doesn't seem to be all that useful. The dplyr package provides a few convenience functions called `n()` and `n_distinct()` that tell you the number of observations or the number of distinct values of a particular variable.

Notice that summarize takes a data frame and returns a data frame. In this case it's a 1x1 data frame with a single row and a single column. The name of the column, by default is whatever the expression was used to summarize the data. This usually isn't pretty, and if we wanted to work with this resulting data frame later on, we'd want to name that returned value something easier to deal with.

```
# Get the mean expression for all genes
summarize(ydat, mean(expression))

# Use a more friendly name, e.g., meanexp, or whatever you want to call it.
summarize(ydat, meanexp=mean(expression))

# Measure the correlation between rate and expression
summarize(ydat, r=cor(rate, expression))

# Get the number of observations
summarize(ydat, n())

# The number of distinct gene symbols in the data
summarize(ydat, n_distinct(symbol))
```

3.3.6 group_by()

We saw that `summarize()` isn't that useful on its own. Neither is `group_by()` All this does is takes an existing data frame and coverts it into a grouped data frame where operations are performed by group.

```
ydat
group_by(ydat, nutrient)
group_by(ydat, nutrient, rate)
```

The real power comes in where `group_by()` and `summarize()` are used together. First, write the `group_by()` statement. Then wrap the result of that with a call to `summarize()`.

```
# Get the mean expression for each gene
# group_by(ydat, symbol)
summarize(group_by(ydat, symbol), meanexp=mean(expression))

# Get the correlation between rate and expression for each nutrient
# group_by(ydat, nutrient)
summarize(group_by(ydat, nutrient), r=cor(rate, expression))
```

3.4 The pipe: |>

3.4.1 How |> works

This is where things get awesome. The dplyr package imports functionality from the magrittr package that lets you *pipe* the output of one function to the input of another, so you can avoid nesting functions. It looks like this: `|>`. You don't have to load the magrittr package to use it since dplyr imports its functionality when you load the dplyr package.

Here's the simplest way to use it. Remember the `tail()` function. It expects a data frame as input, and the next argument is the number of lines to print. These two commands are identical:

```
tail(ydat, 5)
ydat |> tail(5)
```

Let's use one of the dplyr verbs.

```
filter(ydat, nutrient=="Leucine")
ydat |> filter(nutrient=="Leucine")
```

3.4.2 Nesting versus |>

So what?

Now, think about this for a minute. What if we wanted to get the correlation between the growth rate and expression separately for each limiting nutrient only for genes in the leucine biosynthesis pathway, and return a sorted list of those correlation coeffients rounded to two digits? Mentally we would do something like this:

0. Take the `ydat` dataset
1. *then* `filter()` it for genes in the leucine biosynthesis pathway
2. *then* `group_by()` the limiting nutrient
3. *then* `summarize()` to get the correlation (`cor()`) between rate and expression
4. *then* `mutate()` to round the result of the above calculation to two significant digits
5. *then* `arrange()` by the rounded correlation coefficient above

But in code, it gets ugly. First, take the `ydat` dataset

```
ydat
```

then `filter()` it for genes in the leucine biosynthesis pathway

```
filter(ydat, bp=="leucine biosynthesis")
```

then `group_by()` the limiting nutrient

```
group_by(filter(ydat, bp=="leucine biosynthesis"), nutrient)
```

then `summarize()` to get the correlation (`cor()`) between rate and expression

```
summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient), r = cor(rate,
    expression))
```

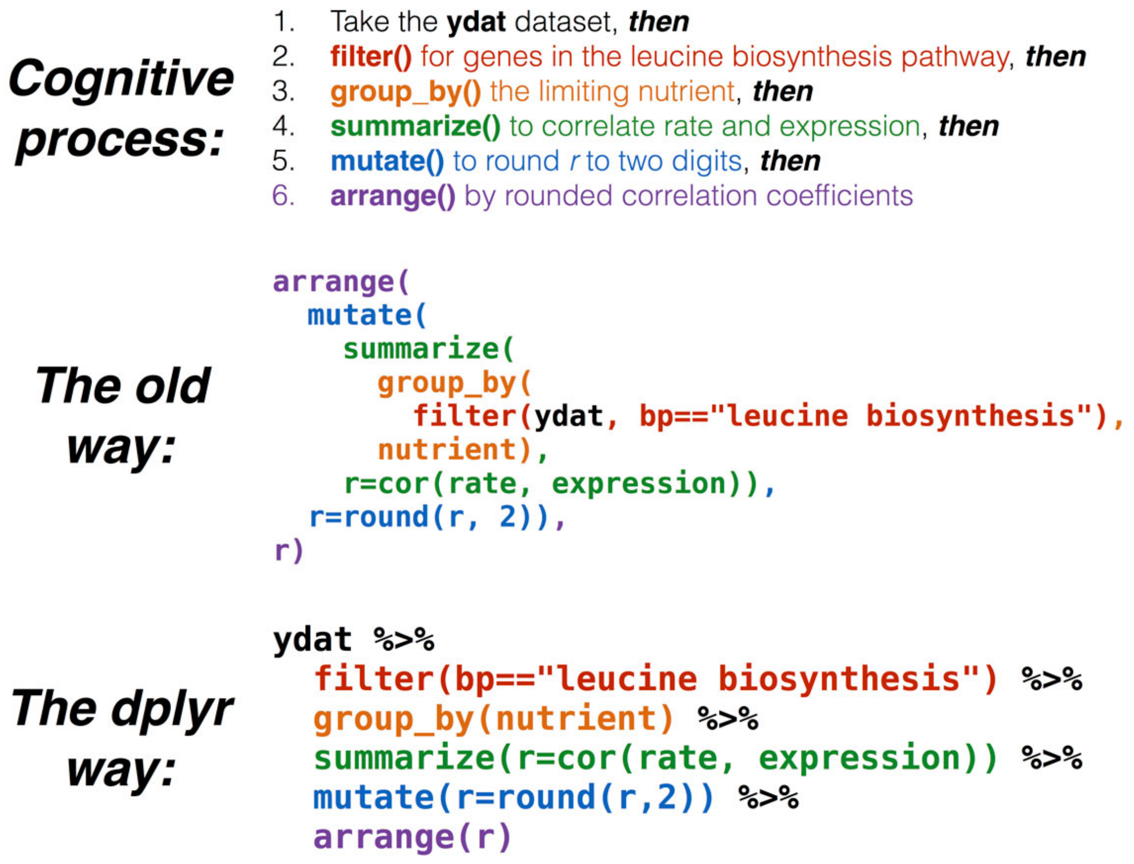
then `mutate()` to round the result of the above calculation to two significant digits

```
mutate(summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient),
    r = cor(rate, expression)), r = round(r, 2))
```

then `arrange()` by the rounded correlation coefficient above

```
arrange(
  mutate(
    summarize(
      group_by(
        filter(ydat, bp=="leucine biosynthesis"),
      nutrient),
    r=cor(rate, expression)),
  r=round(r, 2)),
r)
```

Now compare that with the mental process of what you're actually trying to accomplish. The way you would do this without pipes is completely inside-out and backwards from the way you express in words and in thought what you want to do. The pipe operator `|>` allows you to pass the output data frame from one function to the input data frame to another function.



Cognitive process:

1. Take the **ydat** dataset, ***then***
2. **filter()** for genes in the leucine biosynthesis pathway, ***then***
3. **group_by()** the limiting nutrient, ***then***
4. **summarize()** to correlate rate and expression, ***then***
5. **mutate()** to round *r* to two digits, ***then***
6. **arrange()** by rounded correlation coefficients

The old way:

```
arrange(
  mutate(
    summarize(
      group_by(
        filter(ydat, bp=="leucine biosynthesis"),
      nutrient),
    r=cor(rate, expression)),
  r=round(r, 2)),
r)
```

The dplyr way:

```
ydat %>%
  filter(bp=="leucine biosynthesis") %>%
  group_by(nutrient) %>%
  summarize(r=cor(rate, expression)) %>%
  mutate(r=round(r,2)) %>%
  arrange(r)
```

Figure 3.1: Nesting functions versus piping

This is how we would do that in code. It's as simple as replacing the word "then" in words to the symbol `|>` in code. (There's a keyboard shortcut that I'll use frequently to insert the `|>` sequence – you can see what it is by clicking the *Tools* menu in RStudio, then selecting *Keyboard Shortcut Help*. On Mac, it's CMD-SHIFT-M.)

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient) |>
  summarize(r=cor(rate, expression)) |>
  mutate(r=round(r,2)) |>
  arrange(r)
```

3.5 Exercises

Here's a warm-up round. Try the following.

Exercise 3

Show the limiting nutrient and expression values for the gene ADH2 when the growth rate is restricted to 0.05. *Hint:* 2 pipes: `filter` and `select`.

Exercise 4

What are the four most highly expressed genes when the growth rate is restricted to 0.05 by restricting glucose? Show only the symbol, expression value, and GO terms. *Hint:* 4 pipes: `filter`, `arrange`, `head`, and `select`.

Exercise 5

When the growth rate is restricted to 0.05, what is the average expression level across all genes in the "response to stress" biological process, separately for each limiting nutrient? What about genes in the "protein biosynthesis" biological process? *Hint:* 3 pipes: `filter`, `group_by`, `summarize`.

That was easy, right? How about some tougher ones.

Exercise 6

First, some review. How do we see the number of distinct values of a variable? Use `n_distinct()` within a `summarize()` call.

```
ydat |> summarize(n_distinct(mf))
```

Exercise 7

Which 10 biological process annotations have the most genes associated with them? What about molecular functions? *Hint:* 4 pipes: `group_by`, `summarize` with `n_distinct`, `arrange`, `head`.

Exercise 8

How many distinct genes are there where we know what process the gene is involved in but we don't know what it does? *Hint:* 3 pipes; `filter` where `bp!="biological process unknown" & mf=="molecular function unknown"`, and after `select`ing columns of interest, pipe the output to `distinct()`. The answer should be **`737`**, and here are a few:

Exercise 9

When the growth rate is restricted to 0.05 by limiting Glucose, which biological processes are the most upregulated? Show a sorted list with the most upregulated BPs on top, displaying the biological process and the average expression of all genes in that process rounded to two digits. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `mutate`, `arrange`.

Exercise 10

Group the data by limiting nutrient (primarily) then by biological process. Get the average expression for all genes annotated with each process, separately for each limiting nutrient, where the growth rate is restricted to 0.05. Arrange the result to show the most upregulated processes on top. The initial result will look like the result below. Pipe this output to a `View()` statement. What's going on? Why didn't the `arrange()` work? *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `View`.

Exercise 11

Let's try to further process that result to get only the top three most upregulated biolgocal processes for each limiting nutrient. Google search "dplyr first result within group." You'll need a `filter(row_number()......)` in there somewhere. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `filter(row_number()....` *Note:* dplyr's pipe syntax used to be `%.%` before it changed to `|>`. So when looking around, you might still see some people use the old syntax. Now if you try to use the old syntax, you'll get a deprecation warning.

Exercise 12

There's a slight problem with the examples above. We're getting the average expression of all the biological processes separately by each nutrient. But some of these biological processes only have a single gene in them! If we tried to do the same thing to get the correlation between rate and expression, the calculation would work, but we'd get a warning about a standard deviation being zero. The correlation coefficient value that results is `NA`, i.e., missing. While we're summarizing the correlation between rate and expression, let's also show the number of distinct genes within each grouping.

```
ydat |>
  group_by(nutrient, bp) |>
  summarize(r=cor(rate, expression), ngenes=n_distinct(symbol))
```

```
Warning: There was 1 warning in `summarize()`.
i In argument: `r = cor(rate, expression)`.
i In group 110: `nutrient = "Ammonia"` and `bp = "allantoate transport"`.
Caused by warning in `cor()`:
! the standard deviation is zero
```

Take the above code and continue to process the result to show only results where the process has at least 5 genes. Add a column corresponding to the absolute value of the correlation coefficient, and show for each nutrient the singular process with the highest correlation between rate and expression, regardless of direction. *Hint:* 4 more pipes: `filter`, `mutate`, `arrange`, and `filter` again with `row_number()==1`. Ignore the warning.

4 Data Visualization

Not much to see here...

5 Tidy EDA

Not much to see here...

6 R Markdown

Not much to see here...

Part II

Electives

7 Essential Statistics

Not much to see here...

8 Survival Analysis

Not much to see here...

9 Predictive Modeling

Not much to see here...

10 Probabilistic Forecasting

Not much to see here...

11 Text Mining

Not much to see here...

12 Phylogenetic Trees

Not much to see here...

13 RNA-seq

Not much to see here...

Summary

In summary, this book has no content whatsoever.

References

Bryan, Jennifer. 2019. "STAT 545: Data Wrangling, Exploration, and Analysis with r." https://stat545.com/.

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. "Moderated Estimation of Fold Change and Dispersion for RNA-seq Data with DESeq2." *Genome Biology* 15 (12): 1–21.

Robinson, David. 2015. "Variance Explained." http://varianceexplained.org/.

Silge, Julia, and David Robinson. 2017. *Text Mining with R: A Tidy Approach*. 1st edition. Beijing ; Boston: O'Reilly Media.

Teal, Tracy K., Karen A. Cranston, Hilmar Lapp, Ethan White, Greg Wilson, Karthik Ram, and Aleksandra Pawlik. 2015. "Data Carpentry: Workshops to Increase Data Literacy for Researchers."

Wilson, Greg. 2014. "Software Carpentry: Lessons Learned." *F1000Research* 3.

Yu, Guangchuang. 2022. "Ggtree: An r Package for Visualization of Tree and Annotation Data." http://bioconductor.org/packages/ggtree/.

Yu, Guangchuang, David K. Smith, Huachen Zhu, Yi Guan, and Tommy Tsan-Yuk Lam. 2017. "Ggtree: An R Package for Visualization and Annotation of Phylogenetic Trees with Their Covariates and Other Associated Data." *Methods in Ecology and Evolution* 8 (1): 28–36.

A Setup

A.1 Software

A.2 Data

1. **Option 1: Download all the data**. Download and extract **this zip file** (11.36 Mb) with all the data for the entire workshop. This may include additional datasets that we won't use here.
2. **Option 2: Download individual datasets as needed.**
   - Create a new folder somewhere on your computer that's easy to get to (e.g., your Desktop). Name it `bds`. Inside that folder, make a folder called `data`, all lowercase.
   - Download individual data files as needed, saving them to the new `bdsr/data` folder you just made. Click to download. If data displays in your browser, right-click and select *Save link as…* (or similar) to save to the desired location.

- data/airway_metadata.csv
- data/airway_scaledcounts.csv
- data/annotables_grch38.csv
- data/austen.csv
- data/brauer2007_messy.csv
- data/brauer2007_sysname2go.csv
- data/brauer2007_tidy.csv
- data/dmd.csv
- data/flu_genotype.csv
- data/gapminder.csv
- data/grads_dd.csv
- data/grads.csv
- data/h7n9_analysisready.csv
- data/h7n9.csv
- data/heartrate2dose.csv
- data/ilinet.csv
- data/movies_dd.csv
- data/movies_imdb.csv
- data/movies.csv

- data/nhanes_dd.csv
- data/nhanes.csv
- data/SRP026387_metadata.csv
- data/SRP026387_scaledcounts.csv
- data/stressEcho.csv

B Additional Resources

Not much to see here...