THE UNIVERSITY OF
CHICAGO BIOLOGICAL SCIENCES

BSD qBio^6 Boot Camp

September 13–18, 2020

Group logos by Professor Stephanie Palmer

BSD qBio^6 @ MBL

General Schedule

Sunday, September 13

3:00-4:00	Check-in with TAs
4:00-4:20	Welcome to qBio6 - Novembre and Prince
4:30-5:15	Team building activities
5:30-6:30	Group compacts

Monday, September 14

8:30-10:00	Basic comp. I / Advanced comp I
10:00-10:30	Coffee break
10:30-12:00	Basic comp. I / Advanced comp I
12:00-1:30	MBL Orientation + Lunch
1:30-3:00	Basic comp. II / Advanced comp II
3:00-3:30	Coffee break
3:30-5:00	Basic comp. II / Advanced comp II
5:00-5:30	Welcome to UChicago Biosciences - Prince
8:00-9:00	Happy Hour

Tuesday, September 15

8:30-10:00	Data visualization (*C.jacchus D.rerio P.polytes T.thermophila*)
	Reproducibility of data analysis (*A.thaliana C.elegans T.roseae X.laevis*)
	Statistics for a data rich world (*D.melanogaster M.musculus O.bimaculoides S.aegyptiacus*)
10:00-10:30	Coffee break
10:30-12:00	Data visualization (*C.jacchus D.rerio P.polytes T.thermophila*)
	Reproducibility of data analysis (*A.thaliana C.elegans T.roseae X.laevis*)
	Statistics for a data rich world (*D.melanogaster M.musculus O.bimaculoides S.aegyptiacus*)
12:00-1:30	Lunch
1:30-3:00	Data visualization (*A.thaliana M.musculus O.bimaculoides X.laevis*)
	Reproducibility of data analysis (*D.melanogaster D.rerio S.aegyptiacus T.thermophila*)
	Statistics for a data rich world (*C.elegans C.jacchus P.polytes T.roseae*)
3:00-3:30	Coffee break
3:30-5:00	Data visualization (*A.thaliana M.musculus O.bimaculoides X.laevis*)
	Reproducibility of data analysis (*D.melanogaster D.rerio S.aegyptiacus T.thermophila*)
	Statistics for a data rich world (*C.elegans C.jacchus P.polytes T.roseae*)
7:15-8:00	Science Faculty Star

Wednesday, September 16

8:30-10:00	Data visualization (*C.elegans D.melanogaster S.aegyptiacus T.roseae*)
	Reproducibility of data analysis (*C.jacchus M.musculus O.bimaculoides P.polytes*)
	Statistics for a data rich world (*A.thaliana D.rerio T.thermophila X.laevis*)
10:00-10:30	Coffee break
10:30-12:00	Data visualization (*C.elegans D.melanogaster S.aegyptiacus T.roseae*)
	Reproducibility of data analysis (*C.jacchus M.musculus O.bimaculoides P.polytes*)
	Statistics for a data rich world (*A.thaliana D.rerio T.thermophila X.laevis*)

7:15-8:00	Wednesday Zoom Happy Hour

Thursday, September 17

8:30-10:00	Workshop Khan (*A.thaliana C.elegans D.rerio*)
	Workshop M. Chen (*O.bimaculoides P.polytes S.aegyptiacus*)
	Workshop Novembre (*C.jacchus D.melanogaster M.musculus*)
10:00-10:30	Coffee break
10:30-12:00	Workshop Khan (*A.thaliana C.elegans D.rerio*)
	Workshop M. Chen (*O.bimaculoides P.polytes S.aegyptiacus*)
	Workshop Novembre (*C.jacchus D.melanogaster M.musculus*)
12:00-1:30	Lunch
1:30-3:00	Workshop Khan (*D.melanogaster O.bimaculoides T.thermophila*)
	Workshop M. Chen (*C.elegans C.jacchus X.laevis*)
	Workshop Novembre (*A.thaliana P.polytes T.roseae*)
3:00-3:30	Coffee break
3:30-5:00	Workshop Khan (*D.melanogaster O.bimaculoides T.thermophila*)
	Workshop M. Chen (*C.elegans C.jacchus X.laevis*)
	Workshop Novembre (*A.thaliana P.polytes T.roseae*)
7:15-9:15	Professional Development Night with CAs

Friday, September 18

8:30-10:00	Workshop Khan (*C.jacchus S.aegyptiacus T.roseae*)
	Workshop M. Chen (*A.thaliana M.musculus T.thermophila*)
	Workshop Novembre (*D.rerio O.bimaculoides X.laevis*)
10:00-10:30	Coffee break
10:30-12:00	Workshop Khan (*C.jacchus S.aegyptiacus T.roseae*)
	Workshop M. Chen (*A.thaliana M.musculus T.thermophila*)
	Workshop Novembre (*D.rerio O.bimaculoides X.laevis*)
12:00-1:30	Lunch
1:30-3:00	Workshop Khan (*M.musculus P.polytes X.laevis*)
	Workshop M. Chen (*D.melanogaster D.rerio T.roseae*)
	Workshop Novembre (*C.elegans S.aegyptiacus T.thermophila*)
3:00-3:30	Coffee break
3:30-5:00	Workshop Khan (*M.musculus P.polytes X.laevis*)
	Workshop M. Chen (*D.melanogaster D.rerio T.roseae*)
	Workshop Novembre (*C.elegans S.aegyptiacus T.thermophila*)
8:00-8:20	Wrap up Session
8:20-9:00	Finishing celebration

Tutorials

Basic computing I
Basic computing II
Advanced computing I
Advanced computing II
Data visualization
Reproducibility of data analysis
Statistics for a data-rich world

Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.

help.search('weighted mean')
Search the help files for a word or phrase.

help(package = 'dplyr')
Find help for a package.

More about an object

str(iris)
Get a summary of an object's structure.

class(iris)
Find the class an object belongs to.

Using Packages

install.packages('dplyr')
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	`2 4 6`	Join elements into a vector
`2:6`	`2 3 4 5 6`	An integer sequence
`seq(2, 3, by=0.5)`	`2.0 2.5 3.0`	A complex sequence
`rep(1:2, times=3)`	`1 2 1 2 1 2`	Repeat a vector
`rep(1:2, each=3)`	`1 1 1 2 2 2`	Repeat elements of a vector

Vector Functions

sort(x)
Return x sorted.

table(x)
See counts of values.

rev(x)
Return x reversed.

unique(x)
See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 5.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){

   Do something

}
```

Example

```
for (i in 1:4){

   j <- i + 10

   print(j)

}
```

While Loop

```
while (condition){

   Do something

}
```

Example

```
while (i < 5){

   print(i)

   i <- i + 1

}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){

   Do something

   return(new_variable)
}
```

Example

```
square <- function(x){

   squared <- x*x

   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

Learn more at web page or vignette • package version • Updated: 3/15

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.

`as.factor`	`'1', '0', '1',`
	`levels: '1', '0'`
		Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.

`m[2, ]` - Select a row

`m[ , 1]` - Select a column

`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

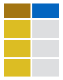
Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

`df$x`


`df[[2]]`

Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]`

`df[2, ]`

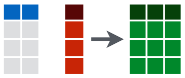
`df[2, 2]`

`nrow(df)`
Number of rows.

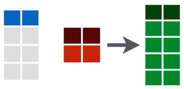
`ncol(df)`
Number of columns.

`dim(df)`
Number of columns and rows.

cbind - Bind columns.



rbind - Bind rows.



Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

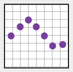
`aov`
Analysis of variance.

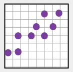
Distributions

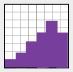
	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.


`plot(x)`
Values of x in order.


`plot(x, y)`
Values of x against y.


`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

Basic Computing 1 — Introduction to R*

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aim of this workshop is to introduce R and RStudio, and show how it can be used to analyze data in an automated, replicable way. We will illustrate the notion of assignment and present the main data structures available in R. We will learn how to read, inspect, manipulate, analyze and write data, and how to execute simple programs. This workshop is intended for biologists with little to no background in programming.

About this tutorial

We will work through the tutorial examples all together as a class, as well as in smaller groups. Feel to free to try the examples on your own, before or after the in-class tutorial. To run the examples before the data have been collected in class, you may use the 2019 data, which are stored in file `laptop_2019.csv`.

Setup

To complete the tutorial, you will need to install R and RStudio. To install R, follow instructions at `cran.rstudio.com`. Then install Rstudio following the instructions at `bit.ly/2JARd4v`.

Before starting up RStudio, it is helpful to quit applications that are not needed, and other "clutter", to reduce distractions.

Launch RStudio. It is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu. Also, make sure your R working directory is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this.

Motivation

When it comes to analyzing data, there are two competing paradigms. One could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; or one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is preferred because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

*This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2020. **Current version**: August 17, 2020; **Corresponding author**: pcarbo@uchicago.edu. Thanks to John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a labmate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce R, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

What is R?

R is a statistical software that is completely programmable. This means that one can write a program ("script") containing a series of commands for the analysis of data, and execute them automatically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is free software: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. See www.r-pkg.org for a listing of official packages (which have been vetted by R core developers); many more are available on Bioconductor, GitHub and other websites.

The main hurdle new users face when approaching R is that it is based on a command-line interface; when you launch R or RStudio, you open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example, this calculates the sum $1 + 2 + \cdots + 99$ and prints the result:

```
sum(1:99)
```

A little history

R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

A "data-centric" view of programming

For those of you who already have experience with a programming language—interactive (e.g., MATLAB, Python) or otherwise (e.g., C++, Java)—you will find that there is much to learn and discover in R. One distinguishing feature of R is that it promotes a *data-centric* view of programming practice; that is, it is not the proedures (actions) that are the primary focus, but instead the *data structures* (objects). This is not accidental—analysis of data is central to R. Appreciating this emphasis on data can help you navigate R, particularly if you are more familiar with procedural programming langauges such as C++.

RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. One advantage of RStudio is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

The main RStudio interface is split up into "panels". The most important panels are:

1. **Console:** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
2. **Source:** In this panel, you can write a program and save it to a file. The code and also be run from this panel, but the actual results show up in the Console.
3. **Environment:** This panel lists all the variables you created (more on this later).
4. **History:** This gives you the history of the commands you typed.
5. **Plots:** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (e.g., type `help(p.adjust)` in the Console).

What is an R "program"?

An R program is simply a list of commands, which are executed one after the other (for this reason, it is sometimes called a "script"). The commands are written in a text file (with extension .R). When R executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time R encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. The advantage of a program, or "script", is that it allows for automating the computations; that is, it eliminates the manual copying and pasting. Moreover, after a while you will have accumulated a lot of code, at which point you can reuse much of your code for new projects.

Tutorial outline

In this tutorial, we will learn about the basic elements of R through analysis of a small data set. This will involve:

1. Collecting the data.
2. Deciding on some analysis goals.
3. Importing the data into R.
4. Inspecting the data.
5. Manipulating the data.
6. Automating the analysis.

In-class activity: Collaborative data collection

We begin this tutorial by collecting the data we will analyze. Specifically, we will collect data about the laptops (or desktops) of incoming BSD graduate students. We will use these data to learn about R, and how it can be used to analyze a data set; we will write some R code that can be used to answer basic questions such as, what is the most common operating system among the population (where "population" is all the current incoming BSD grad students)?

We will collect the data in Etherpad. Specifically, we will collect four data points: first name, operating system, amount of memory (in GB), and number of processors ("cpus"). We will enter these data into a table with four columns. We will use commas to separate the columns in the table. The first few lines of the table will look something like this:

```
name,os,mem,cpus
rishi,mac,8,4
anna,windows,4,1
peter,mac,16,4
```

If you are unsure how to enter some of this information, ask your teammates or peers for help.

Once we have collected the data from the entire class, we will create a new file in RStudio, copy & paste the data (including table header) into the new file, and save the file as laptops.csv.

In-class activity: Formulate analysis aims

Before writing any R code to explore the data, it is helpful to have a focus to the exploration. Suppose you are in charge of designing the QBio programamaing tutorials for next year's incoming BSD graduate cohort. What information about the students laptops would help in designing the course? Here are some basic questions we might want to answer:

- What are the different operating systems used?
- Which operating system is used most?
- Are all the laptops multicore (more than one processor)?

Write down a few more guiding questions in the space below. Try to state your questions using plain (non-technical) language, avoiding statistical terms such as "mean", "correlation" and "significant".

Import the data into R

You should now have a CSV file on your computer, `laptops.csv`, containing the data we collected. (CSV is an abbreviation of "comma-separated values".) The standard `R` function for importing data from a CSV file is `read.csv`:

```
read.csv("laptops.csv", comment = "#", stringsAsFactors = FALSE)
```

In time, we will understand what this code is doing. For now, we will jump ahead.

This command will only work if your `R` working directory is the same as the directory containing `laptops.csv`. You can check your working directory with `getwd()` and `list.files()`. If you are in the wrong directory, use **Session > Set Working Directory** from the `RStudio` menu bar to move to the right directory.

This line of code, on its own, is a dead-end; all it does is print the result to the screen. What is missing from this expression is *assignment of the output to an object.* A common and frustrating mistake—even experienced programmers make this mistake—is forgetting to assign the output.

```
laptops <- read.csv("laptops.csv", comment = "#", stringsAsFactors = FALSE)
```

A good name for the new object helps remind you of its contents (and it is particularly important when you have created many objects).

Assuming you started with a clean workspace, your workspace, or "environment", should now contain a single object, `laptops`:

```
ls()
```

This object stores the output of our `read.csv` call. Invoking its name will print its contents:

```
laptops
```

Or you can explicitly type

```
print(laptops)
```

which does the exact same thing.

What *kind* of object is it?

```
class(laptops)
```

It is a "data frame". This is `R`'s term for "table" or "spreadsheet". (You will find that `R` has a habit of renaming things you are already familiar with.)

If you are used to the "point-and-click" way of doing things in software such as Excel, it may seem burdensome to have arrived at this point where we have written a bunch of code, and all we have done is printed the contents of the table to the screen. But now that we have created a data frame, laptops, there are many powerful things we can do it. Also bear in mind that the advantages of the programmatic approach to data analysis are less obvious when working with small data sets, but the techniques we use here also work with large data sets that are not so easily analyzed by point-and-click (e.g., tables containing millions of rows).

Inspect the data

R has many commands that are easy to use and quickly give you insight into the data. Let's try some of the more commonly used commands for inspecting and summarizing the contents of a data frame. You might want to add a note next to each line of code to remind yourself what it does:

```
nrow(laptops)
ncol(laptops)
head(laptops)
tail(laptops)
str(laptops)
summary(laptops)
```

Some of these commands, like head, str and summary, are "generic" functions, meaning that they work for many object types, from basic types to complex data structures. Generic functions such as these are routinely used, and they will likely become part of your go-to data analysis toolkit.

Data subviews

This data set is small enough that you can easily inspect all of it by eye. However, when you are working with a much larger data set you need a strategy to inspect manageable subsets of the data.

Each of the examples below print a subset of the data. As before, add a note next to each line of code to remind yourself what it does.

```
laptops[ ,2]
laptops[[2]]
laptops[ ,"os"]
laptops["os"]
laptops$os
laptops[4, ]
laptops[4, 2]
laptops[4, "os"]
laptops$os[4]
laptops[4, ]$os
```

Here are a few slightly more complex examples:

```
laptops[1:4, ]
laptops[ ,2:3]
laptops[ ,c("os", "mem")]
laptops[c("os", "mem")]
```

Once you are comfortable with the basic elements of selecting subsets, you can combine these elements in an endless variety of ways, e.g.,

```
laptops[c(1:3, 5:7), c("name", "os")]
```

Creating new data sets from subviews

The result of almost any calculation in R can be saved to an object. This includes selecting subsets. For example,

```
x <- laptops[1:10, ]
print(x)
class(x)
```

creates a new table (a data frame) from the first 10 rows of full `laptops`. There should now be two data frames in your environment:

```
ls()
```

Objects can also be *overwritten*:

```
x <- laptops[1:5, ]
```

The 10-row table you created is now gone.

Observe that both assignment (creating new objects) and overwriting both use `<-`. So be careful—*there is no undo command in R!*

This ability to have multiple data sets floating around in your environment underscores the importance of naming your objects well, and keep track of what is in your environment; names such as “x” or “temp” or “dat”, while commonly used, should only be used for temporary or “throw-away” calculations. It isn’t unusual to have half a dozen different copies of a data set over the course of an analysis. This is where the `rm` function also comes in handy for cleaning up your workspace. Another important practice is to document your code in case you do accidentally overwrite something important!

Conditional subviews

One powerful way to inspect subsets is by condition. For example, to view all the laptops with exactly two processors, do

```
subset(laptops, cpus == 2)
```

There are at least a couple other ways to achieve the same thing:

```
laptops[laptops$cpus == 2, ]
```

and

```
rows <- which(laptops$cpus == 2)
laptops[rows,]
```

The last approach is interesting because it involves creating a new *numeric* object containing the numbers of the rows we are interested in.

We will learn more about logical expressions below.

Question: Which way is best?

Exercise: How would you select the subset of laptops that have Windows?

A side note: "<-" versus "="

Let's take a moment to discuss a common point of confusion. In R, the equality symbol (=) can also be used for creating and overwriting objects. Many people prefer to use =, but we recommend against using it because:

1. The = is easily confused with ==. (How are = and == different?)
2. The = is also used for named arguments to functions (see the read.csv call above for an example). This is not assignment—no objects are created or overwritten. (Some people do incorrectly all this "assignment"—feel free to correct them.)

Deconstructing the "laptops" data frame

We will continue to explore the laptops data frame. A data frame is an example of a *composite* data structure—it is made of simpler data objects. These simpler ("atomic") data objects are R's "building blocks" for all other data structures. In a data frame, the *columns* are the atomic data objects; in the next sections, we will take a close look at the columns of the laptops data frame.

Text data

Let's begin with the "os" column:

```
x <- laptops$os
print(x)
```

In R, the character type is used to store text data:

```
class(x)
```

You can access individual elements by their index: the first element is indexed at 1, the second at 2, *etc.*

```
x[1]
x[2]
```

R has many built-in functions for operating on text data. Here are some examples (add notes explaining what these lines of code do):

```
nchar(x)
toupper(x)
sort(x)
unique(x)
table(x)
paste(x[1], x[2], x[3])
```

If you want to learn more about a particular function, R has extensive documentation that can accessed directly from the Console (no need to Google it). For example, to learn more about the sort function, type

```
help(sort)
```

From this help page, you will learn, among other things, that you can control the order of the sorting using the decreasing option, and by default entries are sorted in increasing order.

Just as data types can be combined to form more complex data structures, operations can also be combined (provided the combination makes sense, of course!). For example,

```
unique(toupper(x))
```

is equivalent to

```
y <- toupper(x)
unique(y)
```

Numeric data

The "mem" column is an example of numeric data:

```
x <- laptops$mem
class(x)
```

Specifically, it is an integer type since all the numbers are whole numbers. There is another data type, numeric, used to store real numbers (or rather their approximations, as computers have limited memory and thus cannot store numbers like π, or even 0.2), e.g.,

```
y <- x / 100
class(y)
```

Very small or very large numbers can be represented in `R` (any idea what the ^ operator is doing?):

```
x^10
```

Like the `character` data type, a numeric object is a vector, and individual elements can be accessed their index,

```
x[1]
x[2]
```

Also, like the `character` data type, `R` has many built-in functions for working with numbers, such as

```
abs(x)
sqrt(x)
cos(x)
log(x)
```

Notice that, for example, `log(x)` computes the logarithm for *each data point in* `x`, and the result is *a vector of the same length as* `x`. This is idea of automatically applying an operation to each entry of a vector is sometimes called "vectorization". In `R`, vectorization is quite natural because vectors are one of the elemental data structures. Vectorization allows complex operations to be accomplished very simply; for example, if `x` contains 10 million numbers, `y <- sqrt(x)` will compute the square root of these 10 million numbers, and store the result in vector `y`.

Some functions are more versatile, and can work with different data types:

```
sort(laptops$mem)
sort(laptops$os)
table(laptops$mem)
table(laptops$os)
```

Given that `R` was born for statistics, there are many statistical operations—from basic to sophisticated—that you can perform on numeric data (add notes next to these lines explaining what they do):

```
min(x)
max(x)
range(x)
sum(x)
median(x)
quantile(x, 0.5)
mean(x)
summary(x)
```

Finally, standard mathematical operations such as + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation) can be applied to numbers. A less familiar operator is the modulo (%%), which gives the remainder from integer division:

```
x %% 3
```

Exercise: Combine some of the functions and/or operators mentioned above to compute the standard deviation of `x`. Compare your answer against R's built-in solution, `sd(x)`. *Hint:* Use the definition of variance. Write your answer in the box below.

Logical data

The `laptops` data set does not contain logical data. But we can generate logical data from one of the columns:

```
x <- laptops$os == "mac"
print(x)
class(x)
```

The logical data type takes only two values, `TRUE` and `FALSE`.

Exercises:

1. Besides equality, other comparison operators include > (greater than), < (less than), `!=` (differs) and >= and <=. Write an expression using one or more of these operators to create some new logical data.

2. You can also formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write an expression using these operators, as well as the ones above, to create some new logical data.

Manipulating data

Up until now, we have treated the laptops data frame as if it were a static object. But like almost any other object in `R`, a data frame can be modified and overwritten. This is very powerful but also obviously dangerous! Here we will illustrate a few of the many kinds of data manipulations we can make, building on the elements we learned above. But before doing so, it is good practice to create a copy of the data frame in case we make a mistake along the way:

```
laptops2 <- laptops
```

Note that `laptops` and `laptops2` are two *entirely independent* copies of the data. Although they are identical at first, as soon you a make change to `laptop2`, you will have two different data sets. Again, when you have multiple versions of a data set present in your environment, it is your responsibility to keep your environment organized.

You can overwrite individual entries of the data frame, or several entries at once:

```
laptops2[c(1, 2), "mem"] <- 2
```

Using a conditional subview, we can replace all instances of "mac" values in the "os" column with "macOS":

```
rows <- which(laptops2$os == "mac")
laptops2[rows, "os"] <- "macOS"
```

If you don't like the column names, you can also change them, too:

```
colnames(laptops2) <- c("name", "OS", "GB", "CPUs")
```

You can reorder the rows or columns:

```
rows     <- order(laptops2$name)
laptops2 <- laptops2[rows, ]
laptops2 <- laptops2[c("name", "CPUs", "GB", "OS")]
```

Or you can even create new columns:

```
n <- nrow(laptops2)
laptops2$id <- seq(1, n)
laptops2$MB <- 1000 * laptops2$GB
```

Building a data frame

Above, we deconstructed the `laptops` data frame, and we found that the constituent parts are the table columns (data vectors). We also go in the reverse direction—we can build a data frame by joining together data vectors. Here's a small illustration of this:

```
x <- laptops$os
y <- laptops$cpu
```

Given text data `x` and numeric data `y`, construct a new data frame:

```
dat <- data.frame(os = x,cpu = y)
```

Factors

So far, we have gotten acquainted with three basic data types: `character`, `numeric` and `logical`. Here we introduce a fourth: the `factor`. Factors are vectors, like the other atomic data types we have seen. What is unusual about factors is that there is no equivalent of factors in other popular programming languages, at least not as a native data type.

None of the columns in our `laptops` table are a factor, but we can easily create a factor from one of the columns using a function called "factor". Let's try this with the "mem" column:

```
x <- laptops$mem
y <- factor(x)
class(x)
class(y)
```

Let's now compare the contents of `x` and `y`:

```
print(x)
print(y)
```

Although the contents of `x` and `y` look very similar, the result of `summary` is very different:

```
summary(x)
summary(y)
```

Let's try the same with the "os" column:

```
x <- laptops$os
y <- factor(x)
class(x)
class(y)
summary(x)
summary(y)
```

Question: Based on the summaries of the two factors, what do you think a factor is?

More questions: Which representation to do you prefer for "mem", numeric or factor? What about the "os" column? Are other columns good candidates for being factors?

These questions—should I convert by data to a factor or not—touch on the broader question of *data representation* or *encoding*. Choosing the right representation of your data can be a critical element to your analysis. For example, in quantitative genetics the genotypes of a diploid organism (e.g., AA, AG, GG) are conventionally encoded as 0, 1 and 2. This numerical encoding has many advantages; for example, the allele frequency is easily calculated as `mean(x)/2`. Judicious use of factors can also help you perform complex calculations very simply. (You may find that factors are particularly useful in the programming challenge!)

If you prefer the "os" column to be a factor, you can change it inside the data frame by doing

```
laptops$os <- factor(laptops$os)
```

Save your work

Since are nearing the end of the tutorial, so this is a good point to save our work. To save your results, go to **Session > Save Workspace As** in `RStudio`, or run

```
save.image("basic_computing_1.RData")
```

Later, to restore your environment, select **Session > Load Workspace** in RStudio, or run

```
load("basic_computing_1.RData")
```

Question: What is the difference between **File > Save As** and **Session > Save Workspace As**?

Group activity: Analysis of laptops data

In this activity, your team will pick two or three of the guiding questions that you and others have formulated, and use R to answer these questions.

Before writing any code, discuss with your team what steps you will take to answer a question. Points of discussion may include: (1) What column(s) will you need? (2) What calculations or operations will you need to perform? (3) Will you need to perform your calculations on all rows, or a subset of the rows?

For each question, record your answer, as well as the code you wrote to generate the answer.

Use this code to create a *script*—a text file containing the code necessary to run the complete data analysis, starting with the `read.csv` call, and ending with the calculations that produce the answers.

Once you have written your script, verify that it runs, and save the file somewhere on your computer. Give your script a memorable name such as `analyze_laptops_data.R`. Also, please add a few comments (these are lines starting with "#") explaining in plain language what the code does. We will use Etherpad to share our scripts.

Programming Challenge

Instructions

Work with your team to solve the following exercise. When you have found the solution, go to the `jnovembre.github.io/BSD-QBio6` and follow the link "Submit solution to challenge 1" to submit your answer.

At the end of the boot camp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or the UofC-hosted Google Drive.) The aim is not just to complete the challenges, but also to do collaboratively; all team members should be included, and should have the opportunity to contribute and learn from each other.

Nobel nominations

The file `nobel_nominations.csv` contains data on Nobel Prize nominations from 1901 to 1964. There are three columns (the file has no "header"): (1) the field (e.g., "Phy" for physics), (2) the year of the nomination, and (3) the id and name of the nominee.

1. Take Chemistry (`Che`). Who received the most nominations?
2. Find all researchers who received nominations in more than one field.
3. Take Physics (`Phy`). Which year had the largest number of nominees?
4. For each field, what is the average number of nominees per year? Calculate the number of nominees in each field and year, and take the average across all years.

Hints

- You will need to subset the data. To make your calculations more clear, it may be helpful to give names to the columns. For example, suppose you imported the data into a data frame called `nobel`. Then `colnames(nobel) <- c("field", "year", "nominee")` should do the trick.
- The simplest way to obtain a count from a vector is to use the `table` function. For example, the command `sort(table(x))` produces a table of the occurrences in `x`, in which the counts are sorted from smallest to largest.
- You can also use the same function, `table`, to build a table using more than one vector. For example, suppose `x` and `y` are vectors of the same length. Then `table(x,y)` will build a table with counts for each unique pair of occurrences in `x` and `y`.
- Some other functions you may find useful for the challenge: `colMeans`, `factor`, `head`, `length`, `max`, `read.csv`, `subset`, `tail`, `tapply`, `unique`, `which` and `which.max`.
- Save your solution code for each exercise in a file.

Additional topics

Creating data

Above, we imported data into R. In addition to viewing and manipulating existing data, R also has many facilities for creating data structures, either from scratch, or from existing data. (We already saw some examples of this.)

The most basic tool is the c function, short for “combine”. It can combine multiple objects or values:

```
x      <- c(2, 3, 5, 27, 31, 13, 17, 19)
sex    <- c("M", "M", "F", "M", "F")           # Sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # Weight (in mg)
```

You can generate sequences of numbers using seq. For example, this generates all odd numbers from 1 to 100:

```
x <- seq(from = 1, to = 100, by = 2)
```

For simpler number sequences, use the colon operator:

```
x <- 1:10
```

To repeat a value (or values) several times, use rep:

```
x <- rep("treated", 5)  # Treatment status
x <- rep(c(1, 2, 3), 4)
```

Finally, there are many functions for generating random numbers. For example,

```
x <- runif(100)
```

Question: What is the result of running runif(100)?

Matrices

A matrix is like a data frame—it also has rows and columns. A key difference is that all the columns must be of the same type. Here’s an example of a 2 x 2 matrix:

```
A <- matrix(c(1, 2, 3, 4), 2, 2)  # Inputs are values, nrows, ncols.
```

In the case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, *etc*).

```
A %*% A        # Matrix product
solve(A)       # Matrix inverse
t(A)           # Transpose
A %*% solve(A) # This should return the identity matrix.
```

Determine the dimensions of a matrix:

```
nrow(A)
ncol(A)
dim(A)
```

Use indices to access a particular row or column of a matrix:

```
Z <- matrix(1:9, 3, 3)
Z[1, ]              # First row.
Z[ ,2]              # Second column.
Z[1:2, 2:3]         # Submatrix with coefficients in rows 1 & 2, and columns 2 & 3.
Z[c(1, 3), c(1, 3)] # Indexing non-adjacent rows and columns.
```

Some operations apply to all elements of the matrix:

```
sum(Z)
mean(Z)
```

Question: When is it better to store data in a matrix instead of a data frame?

Arrays

If you need tables with more than two dimensions, use arrays:

```
A <- array(1:24, c(4, 3, 2))
```

A matrix is a special case of an array with two dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing to be careful about: `R` drops dimensions that are not needed. So, if you access a "slice" of a 3-dimensional array, you should obtain a matrix:

```
A[ ,2, ]
dim(A[ ,2, ])
class(A[ ,2, ])
```

Lists

You have already worked with a list object (perhaps without realizing it). The laptops data frame is also a list:

```
is.list(laptops)
```

A data frame is a special kind of list—it is a list in which every list item is a vector of the same length. Lists also allow vectors of different lengths, for example, here we add an additional item ("date") that has a length of 1:

```
mylist <- as.list(laptops)
mylist$last_modified <- "September 9, 2019"
```

We could not do this in a data frame.

A list is a very general and very widely used `R` data structure for storing complex data.

Missing data

Finally, `R` allows most types of data—text, numeric, factors, *etc*—to take on a special value, `NA`. This is short for "not available" or "not assigned", and is commonly called "missing data". One special feature of `R` is that it often gracefully handles data sets containing missing data.

Exercise: Set a few entries in the `laptops` data frame to `NA`, then run `summary(laptops)`. How are the missing data reported in the summary?

Optional group activity: Analyzing genetic data

In this activity, you will apply the tools we have developed so far to look at an example data set from human genetics: genotype data on chromosome 6 collected from European individuals. (Data adapted from the Human Genome Diversity Project by John Novembre.)

```
chr6 <- read.table("H938_Euro_chr6.geno", header = TRUE)
```

Setting `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
nrow(chr6)
ncol(chr6)
```

It contains 7 columns and over 40,000 rows!

The table reports the number of homozygotes (`nA1A1`, `nA2A2`) and heterozygotes (`nA1A2`), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals. The other columns are:

- `CHR`: The chromosome (in this case, 6).

- `SNP`: The identifier of the Single Nucleotide Polymorphism.
- `A1`: One of the observed alleles.
- `A2`: The other allele.

Write `R` code to answer the following questions:

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. *Hint:* Recall that you can access the columns by index (e.g., `chr6[ ,5]`), or by name (e.g., `chr6$nA1A1`, or `chr6[ ,"nA1A1"]`).
2. Use the `rowSums` function to obtain the same answer.
3. How many SNPs have no heterozygotes (*i.e.,* no "A1A2")?
4. How many SNPs have less than 1% heterozygotes?

Basic Computing 2 — Packages, functions and better code[*]

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aims of this workshop are to: (1) learn how to install, load and use the many of the freely available R packages; (2) illustrate how to write user-defined functions, and how to organize and improve your code; use basic plotting functions; and (3) introduce the package knitr for writing beautiful reports. *This workshop is intended for biologists with basic knowledge of R.*

Setup

To follow the examples below, you will first need to install the **knitr** and **readr** packages. You will also use the **MASS** package for statistics (this package is already included with R).

Introductory activity: Exploring simple steps for creating better code

Much of Basic Computing 2—as well as some of the other qBio tutorials—is about the *practice* of coding in R. Using R effectively for your research projects builds on Good Practice; without it, your analyses will quickly become unmanageable, you will become discouraged, and your progress will be slow.

It may seem premature to talk about practice when we have only just started learning about R (and particularly so if you are new to programming!). But you will find that it isn't long into a project—sometimes with as little as 40–60 lines of code—when you need strategies to help you break out of a jam.

Many researchers, of course, manage to push through without adopting Good Practices. This is because they are creative and are able to perservere. But perserverance and creativity are finite resources that we would rather you use elsewhere!

Incorporating Good Practices into your work is not something that will happen overnight; it will take time and experience to develop your practice. Our aim here, in this short activity, and in other parts of Basic Computing 2, is to give you a head start—in particular, to show you that *simple steps* can go a long way to improving your coding practice.

We begin this short activity with a *script* implementing the initial steps in an exploratory analysis of tornado data from the National Weather Service. This script is tornadoes.R. For convenience, I've added it here:

```
# This short script produces a plot showing the number of tornado
# events recorded by the NOAA's National Weather Service on each day
# of 2011.
```

[*] This document is included as part of the Basic Computing 2—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2020. **Current version**: August 17, 2020; **Corresponding author**: pcarbo@uchicago.edu. Thanks to John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

```r
# Load the storm event data downloaded from
# www.ncdc.noaa.gov/stormevents/ftp.jsp
library(readr)
storms <- read_csv("storms2011.csv.gz", guess_max = 5000)
class(storms) <- "data.frame"

# Add columns for date and "day of year" (number from 1 to 365).
storms <- transform(storms, date = as.Date(paste(2011, BEGIN_YEARMONTH - 201100,
                                                 BEGIN_DAY, sep = "-")))
storms <- transform(storms, dayofyear = as.numeric(format(date, "%j")))

# Focus on storm events classified as tornadoes.
tornadoes <- subset(storms, EVENT_TYPE == "Tornado")

# Plot the number of tornado events per day.
breaks     <- seq(0, 365, 3)
first_days <- c(1,32,60,91,121,152,182,213,244,274,305,335)
months     <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug",
                "Sep","Oct","Nov","Dec")
counts     <- as.numeric(table(cut(tornadoes$dayofyear, breaks)))
plot(breaks[-1], counts, type = "l", xaxt = "n")
axis(1, cex.axis = 0.7, at = first_days, labels = months)
```

If you run this code—say, by copying and pasting the code into the Console, or by running `source("tornadoes.R")`—you should see a plot showing the number of tornado events that were recorded on each day of 2011. You should also see a large "spike" in tornadoes at the end of April.

This code is quite a bit more complicated than any of the code we've seen before in Basic Computing 1. We will not spend much time trying to understand it. Before moving on, however, this is a good opportunity to point out some of the Good Practices that are implemented in this script:

1. It is self-contained—that is, it is a complete script that includes steps such as loading the necessary packages.
2. The steps are written in a logical order, starting with loading the data, and ending with a result.
3. The variable names are helpful.
4. Short, high-level comments explain what the code does.

My hope is that all these elements become part of your coding practice over time.

This script only implements the very first step in the analysis. There is still much we would like to learn, including the tornadoes' severity, and their geographic distribution. For example, are these patterns repeated in the state of New York? Using `subset`, and copying some of the code above, we can investigate this question:

```r
temp   <- subset(tornadoes,STATE == "NEW YORK")
counts <- as.numeric(table(cut(temp$dayofyear, breaks)))
plot(breaks[-1], counts, type = "l", xaxt = "n")
axis(1, cex.axis = 0.7, at = first_days, labels = months)
```

This exploration will quickly become tedious if we do it in this way: fiddling with the code, and re-running it to produce a new result. You will also find that it makes difficult to keep track of the code that produced the most interesting insights.

Fortunately, there is a better way: we can *package* this code into a function. Let's begin with the part of the code that does the calculation of the daily counts (to be precise, it counts the number of events in 3-day intervals):

```
count_events <- function (dat) {
  x <- as.numeric(table(cut(dat$dayofyear, breaks)))
  return(x)
}
```

On its own, this code doesn't do anything except create the new function, `count_events`. You need to invoke it to do something useful:

```
dat    <- subset(tornadoes, STATE == "ILLINOIS")
counts <- count_events(dat)
print(counts)
```

We can now use this function anywhere we choose. For example, suppose we wanted to see whether these same patterns hold up in the more severe tornadoes (as measured by the "Fujita scale"). The `count_events` function allows us to implement this multi-step calculation in a one-liner:

```
counts <- count_events(subset(tornadoes, TOR_F_SCALE == "EF2"))
print(counts)
```

Before turning to the plots, let's take a moment to reflect on the immediate benefits of having created the `count_events` function:

1. It simplifies any parts of our analysis that make use of the this calculation.
2. It reduces the potential to make mistakes because we have wrapped these tricky calculations inside a function with a memorable name;
3. Avoids repetition of complicated code.
4. Packaging this code in a function suggests its *generality* (nothing in this code is specific to tornado events), and thereby promotes code reuse.

To further streamline our analysis, let's wrap the complicated plotting code inside a function:

```
plot_events <- function (dat) {
  x <- count_events(dat)
  plot(breaks[-1], x, type = "l", xaxt = "n")
  axis(1, cex.axis = 0.7, at = first_days, labels = months)
}
```

Let's try out this new function:

```
plot_events(tornadoes)
plot_events(subset(tornadoes, TOR_F_SCALE == "EF2"))
plot_events(subset(tornadoes, STATE == "ALABAMA"))
```

Note that this function does not output anything; the important action is to draw something to the screen.

Observe that we are making good use of our `count_events` function. Splitting the analysis into two functions seems logical; we have separated the calculation of the counts from plotting of the counts.

Again, this function is general—there is nothing about this code that is specific to tornadoes. So we may be able to reuse this function for similar analyses (e.g., studying annual flooding patterns).

Incorporating the above improvements (and a couple other small improvements), our new analysis script looks like this (see also `tornadoes_better.R`):

```
# This short script produces a plot showing the number of tornado
# events recorded by the NOAA's National Weather Service on each day
# of 2011.

# Load the storm event data downloaded from
# www.ncdc.noaa.gov/stormevents/ftp.jsp
library(readr)
storms <- read_csv("storms2011.csv.gz", guess_max = 5000)
class(storms) <- "data.frame"

# Add columns for date and "day of year" (number from 1 to 365).
storms <- transform(storms, date = as.Date(paste(2011, BEGIN_YEARMONTH - 201100,
                                                  BEGIN_DAY, sep = "-")))
storms <- transform(storms, dayofyear = as.numeric(format(date, "%j")))

# Focus on storm events classified as tornadoes.
tornadoes <- subset(storms, EVENT_TYPE == "Tornado")

# Returns a vector of length 365 giving the number of storm events per day.
count_events <- function (dat) {
  x <- as.numeric(table(cut(dat$dayofyear, breaks)))
  return(x)
}

# Plots the number of events per day.
plot_events <- function (dat, title) {
  x <- count_events(dat)
  plot(breaks[-1], x, type = "l", xaxt = "n", main = title)
  axis(1, cex.axis = 0.7, at = first_days, labels = months)
}
```

```r
# Plot the total number of tornado events per day.
breaks     <- seq(0, 365, 3)
first_days <- c(1,32,60,91,121,152,182,213,244,274,305,335)
months     <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug",
                "Sep","Oct","Nov","Dec")
layout(matrix(1:2))
plot_events(tornadoes, "Tornadoes in 2011")

# Plot more severe tornadoes only.
plot_events(subset(tornadoes, TOR_F_SCALE == "EF2"), "Severe tornadoes in 2011")
```

As we continue to develop our analysis, functions can also reduce effort of *revising code*; for example, to change the colour of the lines in all the plots, only one small change needs to be made to the code inside `plot_events`.

Designing functions to improve your code is very much an "art" that you will learn as you gain experience. But don't get hung up on finding the best way to organize your code into functions—it starts with recognizing the most obvious opportunities, and over time you will refine your practice.

Group activity: Compare 2011 and 2019 tornado patterns

Create a copy of `tornadoes_better.R` and call it `tornadoes_generic.R`. Modify the code in `tornadoes_generic.R` so that it can be used to analyze either the 2011 or 2019 data. The first few lines of your script should look like this:

```r
year <- 2011
library(readr)
storms <- read_csv(paste0("storms",year,".csv.gz"),guess_max = 5000)
```

Hint: `paste` may be useful.

Run `tornadoes_generic.R` on the 2011 data, then re-run on the 2019 data by changing the first line to `year <- 2019`. What is the most striking difference in the 2011 and 2019 tornado patterns?

A brief tour of packages, functions, loops, and other topics

Packages

`R` is the most popular statistical computing software among biologists. One reason for its popularity is the availability of many packages for tackling specialized research problems. These packages are often written by biologists for biologists. You can contribute a package, too! The `RStudio` website (`goo.gl/harVqF`) provides guidance on how to start developing `R` packages. See also Hadley Wickham's free online book (`r-pkgs.had.co.nz`).

You can often find highly specialized packages to address your research questions. Here are some suggestions for finding an appropriate package. The Comprehensive R Archive Network (CRAN) offers several ways to find specific packages for your task. You can browse the full list of CRAN packages (goo.gl/7oVyKC). Or you can go to the CRAN Task Views (goo.gl/0WdIcu) and browse a compilation of packages related to a topic or discipline.

From within R or RStudio, you can also call the function RSiteSearch("keyword"), which submits a search query to the website search.r-project.org. The website rseek.org casts an even wider net, as it not only includes package names and their documentation, but also blogs and mailing lists related to R. If your research interests are to high-throughput genomic data or other topics in bioinformatics or computational biology, you should also search the packages provided by Bioconductor (goo.gl/7dwQlq).

Installing a package

Suppose you want to install the rsvd package. The **rsvd** package provides functions to quickly perform singular value decompositions (SVD) and principal components analysis (PCA) on large data sets. To install the package, run:

```
install.packages("rsvd")
```

Or, in RStudio, select the **Packages** panel, and click **Install**.

Loading packages

Once it is successfully installed, to load the **rsvd** package into your R environment, run:

```
library(rsvd)
```

Once you have loaded the package, you can use, for example, the rpca function for running PCA. To access the documentation that explains what rpca does, and how to use it, type

```
help(rpca)
```

Now suppose you would like to access the "bacteria" data set, which reports the incidence of *H. influenzae* in Australian children. The data set is included with **MASS** package. If you try to access these data before loading the package, you will get an error:

```
data(bacteria)
```

First, you need to load the package:

```
library(MASS)
```

Now the data set is available, and you can load it:

```
data(bacteria)
head(bacteria)
```

Random numbers

Your will sometimes need to generate random numbers. (They are actually "pseudorandom" numbers because they are not perfectly random.) In fact, random numbers are needed in the "case study" below.

R has many functions to sample random numbers from different statistical distributions. For example, use runif to create a vector containing 10 random numbers:

```
runif(10)
```

Question: What kind of random numbers are generated by runif? How could you check this?

To sample from a set of values, use sample:

```
v <- c("a", "b", "c", "d")
sample(v, 2)                  # Sample without replacement.
sample(v, 6, replace = TRUE)  # Sample with replacement.
sample(v)                     # Shuffle the elements.
```

The normal distribution is one of the most commonly used distributions, so naturally there is a function in R for simulating from the normal distribution:

```
rnorm(3)                   # Three draws from the standard normal.
rnorm(3, mean = 5, sd = 4) # Change the mean and standard deviation.
```

Exercise: The normal distribution has a familiar shape. Use rnorm to generate a large number of values from the standard normal, then use hist to draw a histogram of these values (you can adjust the number of bins in the histogram with the n argument). Is the histogram "bell shaped"? Use mean, median and sd to verify that the random numbers recover the expected properties of the normal distribution. Here is some code to get you started:

```
set.seed(1)
x <- rnorm(10000)
hist(x, n = 64)
```

Why is set.seed useful? What happens if we remove the call to set.seed?

Writing functions

It is good practice to subdivide your analysis into functions, and then write a short "master" program that calls the functions and performs the analysis. In this way, the code will be more legible, easier to debug, and you will be able to recycle the functions for your other projects.

In R, every function has this form:

```r
my_function_name <- function (arg1, arg2, arg3) {
  #
  # Body of the function.
  #
  return(return_value) # Not required, but most functions output something.
}
```

Here is a very simple example:

```r
sum_two_numbers <- function (a, b) {
  s <- a + b
  return(s)
}
sum_two_numbers(5, 7.2)
```

In R, a function can return only one object. If you need to return multiple values, organize them into a vector, matrix or list, and return that; e.g.,

```r
sum_and_prod <- function (a, b) {
  s <- a + b
  p <- a * b
  return(c(s,p))
}
sum_and_prod(5, 7.2)
```

Here is a more interesting function. It accepts two arguments, `x` and `s`, and returns the density of the normal distribution with zero mean and standard deviation `s` at `x`. This is the mathematical formula for the normal density with mean zero and standard deviation `s`:

$$\frac{e^{-(x/s)^2/2}}{\sigma\sqrt{2\pi}}$$

Let's call this function `normpdf`:

```r
normpdf <- function (x, s) {
  y <- exp(-(x/s)^2/2)/(sqrt(2*pi)*s)
  return(y)
}
```

Exercise: Check that this function gives the correct answers by comparing to the built-in function `dnorm`.

When developing and testing your function, remember this rule: *Whatever happens in the function stays in the function.* Inside `normpdf`, a new object, `y`, is created. But you will not see `y` in your environment (unless you happen to already have an object named `y`).

Vectorization

R has a feature called *vectorization*—many operations in R, including most of the basic mathematical operations, are automatically applied to all values in a vector. (We briefly learned about vectorization in Basic Computing 1.) Let's check whether R automatically vectorizes the normpdf function by running this code:

```
n <- 1000
x <- seq(-3, 3, length.out = n)
y <- normpdf(x, 1)
print(y)
plot(x, y, type = "l")
```

Activity (optional): Which operations in normpdf were applied 1,000 times, and which were applied just once? It may not be immediately obvious just by looking at the code, so to investigate this question, try running parts of the code in the console, e.g., exp(-(x/s)^2/2), and see what happens.

Vectorization is very powerful, but it may take time to get comfortable with it, and know when it will work.

Conditional branching

When we want a block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met) {
  # Execute this block of code.
} else {
  # Execute this other block of code.
}
```

For example, try running these lines of code (you might want to try running them a few times):

```
x <- rnorm(1)
if (x < 0) {
  msg <- paste(x, "is less than zero")
} else if (x > 0) {
  msg <- paste(x, "is greater than zero")
} else {
  msg <- paste(x, "is equal to zero")
}
print(msg)
```

We have created a conditional branching point, so that the value of msg changes depending on whether x is less than zero, greater than zero, or equal to zero.

Activity: Improved normal probability density function

The probability density function of the normal distribution is not defined if the standard devation is less than zero. When the standard deviation is exactly zero, the density is a "spike" at zero; it is `Inf` exactly at $x = 0$, and zero everywhere else. The pseudocode for this improved normal probability density function might look something like this:

```
normpdf(x, s)
  if s < 0
    return NaN
  else if s = 0
    if x = 0
      return Inf
    else
      return 0
  else
    evaluate normal pdf with s.d. s at x
```

Using this pseudocode as a guide, write an improved `normpdf` function:

Activity: The quadratic formula

There is a famous formula used to solve for x in the quadratic equation $ax^2 + bx + c = 0$. It is called the *quadratic formula.* Write down the quadratic formula here:

Write a function, `solvequad`, that takes three numbers as input (a, b and c) and returns the solution(s) x that are real (*i.e.*, not complex). **Hint:** Recall that a quadratic equation may have more than one real solution—or it may have none! You will need to use `if` and `else` to handle all possible cases.

Before writing any R code, first describe your solvequad function without worrying about R syntax—that is, using pseudocode:

Guided by your pseudocode, write the R code for your solvequad function:

After creating solvequad, check that it does the right thing by running these tests:

```r
solvequad(4, 4, 1)    # Should return -1/2.
solvequad(1, -1, -2)  # Should return 2 and -1.
solvequad(1, 1, 1)    # Should return no solutions.
```

Run a few more tests using www.wolframalpha.com.

Looping

Another way to change the flow of your program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

R provides you with two ways to loop over code blocks: the for loop and the while loop. Let's start with the for loop, which is used to iterate over a vector or list; for each value of the vector (or list), a series of commands will be run, as shown by the following example:

```
v <- 1:10
for (i in v) {
  a <- i ^ 2
  print(a)
}
```

In the code above, the variable i takes the value of each element of v in sequence. Inside the block within the for loop, you can use the variable i to perform operations.

The anatomy of the for statement:

```
for (variable in list_or_vector) {
  execute these commands
} # Automatically moves to the next value.
```

You should use a for loop when you know that you want to perform the analysis over a given set of values (e.g., files of a directory, rows in your data frames, sequences of a fasta file, *etc*).

The while loop is used when the commands need to be repeated while a certain condition is true, as shown by the following example:

```
i <- 1
while (i <= 10) {
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

The script gives exactly the same result as the for loop above. A key difference is that you need to include a step to update the value of i, (using i <- i + 1), whereas in the for loop it is done for you automatically. The anatomy of the while statement:

```
while (condition is met) {
  execute these commands
} # Beware of infinite loops... remember to update the condition!
```

You can break a loop using break. For example:

```
i <- 1
while (TRUE) {
  if (i > 10) {
    break
  }
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

Question: Above, we ran three different loops, we found that each of them accomplished the same thing. Is one approach better? Why?

Programming challenge

Instructions

You will work with your group to solve the exercises below. When you have found the solutions, go to `https://jnovembre.github.io/BSD-QBio6` and follow the link "Submit solution to challenge 2" to submit your answer. At the end of the bootcamp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Google Flu Trends

Google Flu started strong, with a paper in *Nature* (Ginsberg *et al*, 2009, `doi:10.1038/nature07634`) showing that, using data on Web search engine queries, one could predict the number of physician visits for influenza-like symptoms. Over time, the quality of predictions degraded considerably, requiring many adjustments to the model. Now defunct, Google Flu Trends has been proposed as a poster child of "Big Data hubris" (Lanzer *et al*, *Science*, 2014, `doi:10.1126/science.1248506`). In the folder containing the Basic Computing 2 tutorial materials, you will find the data used by Preis and Moat in their 2014 paper (`doi:10.1098/rsos.140095`) to show that, after accounting for some additional historical data, Google Flu Trends are correlated with outpatient visits due to influenza-like illnesses.

1. Read the data using function `read.csv`, and plot the number of weekly outpatient visits versus the Google Flu Trends estimates.
2. Calculate the (Pearson's) correlation using the `cor` function.
3. The data span 2010–2013. In August 2013, Google Flu changed their algorithm. Did this lead to improvements? Compare the data from August and September 2013 with the same months in 2010, 2011 and 2012. For each, calculate the correlation, and see whether the correlation is higher for 2013.

Hint: You will need to extract the year from a string for each row. This can be done using `substr(gf$WeekCommencing, 1, 4)`, in which `gf` is the data frame containing the Google Flu data.

Case study: Do shorter titles lead to more citations? (optional)

To keep learning about `R`, we study the following question:

Is the length of a paper title related to the number of citations?

This is what Letchford *et al* claimed (`doi:10.1098/rsos.150266`). In 2015, they analyzed 140,000 papers, and they found that *shorter titles* were associated with a larger number of citations.

In the folder containing the Basic Computing 2 tutorial materials, you will find data on scientific articles published between 2004 and 2013 in three top disciplinary journals, *Nature Neuroscience*, *Nature Genetics* and *Ecology Letters*. These data are conatined in three CSV files. We are going to use these data to explore this question.

Load data

Start by reading in the data:

```
data.file <- file.path("citations", "nature_neuroscience.csv")
papers    <- read.csv(data.file, stringsAsFactors = FALSE)
```

Next, take a peek at the data. How large is it?

```
nrow(papers)
ncol(papers)
```

Let's see the first few rows:

```
head(papers)
```

The goal is to test whether papers with longer titles accrue fewer (or perhaps more?) citations than those with shorter titles. The first step is to add another column to the data containing the length of the title for each paper:

```
papers$TitleLength <- nchar(papers$Title)
```

Basic statistics

In the original paper, Letchford *et al* used rank-correlation: rank all the papers according to their title length and the number of citations. If Kendall's τ ("rank correlation") is positive, then longer titles are associated with *more* citations; if τ is negative, longer titles are associated with *fewer* citations. In `R`, you can compute rank correlation using `cor`:

```
k <- cor(papers$TitleLength, papers$Cited.by, method = "kendall")
```

To perform a significance test for correlation, use `cor.test`:

```
k.test <- cor.test(papers$TitleLength, papers$Cited.by, method = "kendall")
```

Does the output of `cor.test` show that the correlation between the ranks is positive or negative? Is this positive or negative correlation significant? You should find that the correlation is opposite of the one reported by Letchford *et al*—longer titles are associated with *more* citations!

Now we are going to examine the data in a different way to test whether this result is robust.

Basic plotting

To plot title length vs. number of citations, we need to learn about plotting in `R`. To produce a simple scatterplot using the base plotting functions, simply run:

```
plot(papers$TitleLength, papers$Cited.by)
```

The problem with this very simple plot is that it is hard to detect any trend—a few papers have many more citations than the rest, obscuring the data at the bottom of the plot. This suggests that plotting the data on the *logarithmic scale* is a better approach:

```
plot(papers$TitleLength, log10(papers$Cited.by))
```

Question: This is a better plot, but there is one problem with it. What is the problem, and what fix would you suggest? Write your code to fix the problem:

Again, it is hard to see any trend in here. Maybe we should plot the best fitting line and overlay it on top of the graph. To do so, we first need to learn about linear regressions in `R`.

Linear regression

`R` was born for statistics — the fact that it is very easy to fit a linear regression in is not surprising! To build a linear model comparing columns `x` and `y` in data frame, `dat`, use `lm`, the "Swiss army knife" for linear regression in `R`:

```
model <- lm(y ~ x, dat)   # y = a + b*x + error
```

Let's perform a linear regression of the number of citations (on the log-scale) vs. title length. To do so, first create a new column in the data frame containing the counts on the log-scale:

```
papers$LogCits <- log10(papers$Cited.by + 1)
```

Now you can perform a linear regression:

```
model_citations <- lm(LogCits ~ TitleLength, papers)
model_citations           # Gives best-fit line.
summary(model_citations)  # Gives more info.
```

And we can easily add this best-fit line to our plot:

```
plot(papers$TitleLength, papers$LogCits)
abline(model_citations, col = "red", lty = "dotted")
```

Once we add the best-fit line, the positive trend is more clear.

One thing to consider is that this data set spans a decade. Naturally, older papers have had more time to accrue citations. In our models, we should control for this effect. But first, we should explore whether this is an important factor to consider.

First, let's plot the distribution of the number of citations for all the papers:

```
hist(papers$LogCits)
```

You can control the number of histogram bins with the `n` argument:

```
hist(papers$LogCits, n = 50)
```

Alternatively, estimate the density using `density`, then plot it:

```
plot(density(papers$LogCits))
```

Next, compare the distributions for papers published in 2004, 2009 and 2013:

```
plot(density(papers$LogCits[papers$Year == 2004]), col = "black")
lines(density(papers$LogCits[papers$Year == 2009]), col = "blue")
lines(density(papers$LogCits[papers$Year == 2013]), col = "red")
```

More recent papers should have fewer citations. Does your plot support this hypothesis? You can account for this in your regression model by incorporating the year of publication into the linear regression:

```
model_citations_better <- lm(LogCits ~ TitleLength + Year, papers)
summary(model_citations_better)
```

Does the regression coefficient (slope) for year confirm that older papers have more citations?

Our new analysis is better than before, but might be even better to have a separate "baseline" for each year. This can be done by converting the "Year" column to a factor:

```
papers$Year            <- factor(papers$Year)
model_citations_better <- lm(LogCits ~ Year + TitleLength, papers)
summary(model_citations_better)
```

This model has a different baseline for each year, and then title length influences this baseline. In this new model, are longer titles still associated with more citations?

Computing *p*-values using randomization

Kendall's τ takes as input two rankings, x and y, both of the same length, n. It calculates the number of "concordant pairs" (if $x_i > x_j$, then $y_i > y_j$) and the number of "discordant pairs". The final value is

$$\tau = \frac{n_{\text{concordant}} - n_{\text{discordant}}}{\frac{n(n-1)}{2}}$$

If x and y are completely independent, we would expect τ to have a distribution centered at zero. The variance of the "null" distribution of τ depends on the data. It is typically approximated as a normal distribution. If you want to have a stronger result that does not rely on a normality assumption, you can use randomizations to calculate a p-value. Simply, compute τ for the actual data, as well as for many "fake" datasets obtained by randomizing the data. Your p-value is then the proportion of τ values for the randomized sets that exceed the τ value for the actual data.

Here, we will try to implement this randomization to calculate a p-value for papers published in 2006, and then we will compare against the p-value obtained from running `cor.test`. To do this, we will use a for-loop for the randomization.

First, subset the data:

```
dat <- papers[papers$Year == 2006, ]
```

Compute τ from these data:

```
k <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Now calculate τ in "fake" data sets by randomly scrambling the citation counts. Begin by doing this for one fake data set:

```
shuffled_citation_counts <- sample(dat$Cited.by)
k.fake <- cor(dat$TitleLength, shuffled_citation_counts, method = "kendall")
```

Is the value of τ closer to zero in this "shuffled" data set?

To get an accurate p-value, we should compute τ for a large number of shuffled data sets. Let's try 1,000 of them. This and similar randomization techniques are known as "bootstrapping".

```
nr     <- 1000         # Number of fake data sets.
k.fake <- rep(0, nr)   # Storage all the "fake" taus.
```

Since this computation involves lots of repetition, a for-loop makes a lot of sense here:

```
for (i in 1:nr){
  shuffled_citation_counts <- sample(dat$Cited.by)
  k.fake[i] <- cor(dat$TitleLength, shuffled_citation_counts,
                   method = "kendall")
}
```

After running this loop, you should have 1,000 correlations calculated from 1,000 fake data sets. You have just generated a "null" distribution for τ. What does this null distribution look like? Try plotting it:

```
hist(k.fake, n = 50)
```

What proportion of the fake data sets have a correlation that exceeds the correlation in the actual data? This is the p-value.

```
pvalue <- mean(k.fake >= k)
```

How does your new p-value compare to the p-value computed by `cor.test`? Is it smaller or larger?

```
cor.test(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Question: Did you get the same result as the instructor, or your neighbours? If not, why? How could you ensure that your result is more similar, or the same?

Whenever possible, use randomizations rather than relying on classical tests. They are more difficult to implement, and more computationally expensive, but they allow you to avoid making assumptions about your data.

Repeating the analysis for each year

Up until this point, we have only analyzed the citation data for 2006. Does the result we obtained for 2006 hold up in other years? Let's explore this question—we will use a for-loop to repeat the analysis for 2004 to 2013. Let's be smart about designing our code and use a *function* to decompose the problem into parts. The code for the final analysis will look like this:

```
years <- 2004:2013
for (i in years){
  dat <- papers[papers$Year == i, ]
  out <- analyze_citations(dat)
  cat("year:", i, "tau:", out$k, "pvalue:", out$pvalue, "\n")
}
```

The missing piece is the code implementing function `analyze_citations`. You can re-use your code above to write this function.

```
analyze_citations <- function (dat) {
  nr     <- 1000
  k      <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
  k.fake <- rep(0, nr)
  for (i in 1:nr) {
    k.fake[i] <- cor(dat$TitleLength, sample(dat$Cited.by), method = "kendall")
  }
  return(list(k = k, pvalue = mean(k.fake >= k)))
}
```

Activity: Organizing and running your code

Now we would like to be able to automate the analysis, such that we can repeat it for each journal. This is a good place to pause and introduce how to go about writing programs that are well-organized, easy to write, easy to debug, and easier to reuse.

1. Take the problem, and divide it into smaller tasks (these are the functions).
2. Write the code for each task (function) separately, and make sure it does what it is supposed to do.
3. Document the code so that you can later understand what you did, how you did it, and why.
4. Combine the functions into a master program.

For example, let's say we want to write a program that takes as input the name of files containing citation data. The program should first fit a linear regression model,

```
log(citations + 1) ~ as.factor(Year) + TitleLength
```

then output the coefficient associated with `TitleLength`, and its *p*-value.

We could split the program into the following tasks:

1. A function to load and prepare the data for a linear regression analysis.
2. A function to run the linear regression analysis.
3. A master code that puts it all together.

Let's begin with the master code—the bulk of the code is a for-loop that repeats the regression analysis for each journal:

```
files <- list.files("citations", full.names = TRUE)
for (i in files) {
  cat("Analyzing data from",i,"\n")
  papers <- load_citation_data(i)
  out    <- fit_citation_model(papers)
  cat("coefficient:", out$estimate, "p-value:", out$pvalue, "\n")
}
```

This code doesn't work yet because you haven't written the functions that are called inside the loop. (What error message to you get when you try to run the code?)

The `load_citation_data` function reads in the data from the CSV file, then prepares the data for the linear regression analysis:

```
load_citation_data <- function (filename) {
  dat <- read.csv(filename, stringsAsFactors = FALSE)
  dat$TitleLength <- nchar(dat$Title)
  dat$LogCits     <- log10(dat$Cited.by + 1)
  dat$Year        <- as.factor(dat$Year)
  return(dat)
}
```

Before continuing, check that it works by running it on one of the CSV files:

```
papers <- load_citation_data("citations/nature_neuroscience.csv")
```

The `fit_citation_model` function fits a linear regression model to the input data, then extracts the quantities from the regression analysis we are most interested in (the "best-fit" slope and the *p*-value corresponding to "TitleLength").

```
fit_citation_model <- function (papers) {
  model <- lm(LogCits ~ Year + TitleLength, papers)
  terms <- summary(model)$coefficients
  return(list(estimate = terms["TitleLength", "Estimate"],
              pvalue   = terms["TitleLength", "Pr(>|t|)"]))
}
```

Check that this function runs, and does what it is supposed to do:

```
out <- fit_citation_model(papers)
```

Now that you have defined the necessary functions, try running the master code above.

Question: Suppose you download a fourth CSV file containing data on papers from the *American Journal of Human Genetics*. Would any changes need to be made to your R code above to run it on the four citation data sets?

Creating computational notebooks using R Markdown (optional)

> *Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.*
>
> Donald E. Knuth, *Literate Programming*, 1984

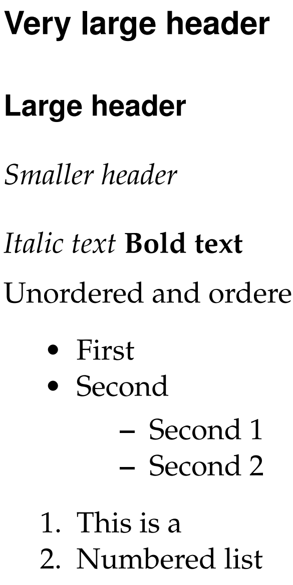
When doing experiments, it is important to develop the habit of writing down everything you do in a laboratory notebook. That way, when writing your manuscript, responding to queries or discussing progress with your advisor, you can go back to your notes to find exactly what you did, how you did it, and possibly *why* you did it. The same should be true for computational work.

`RStudio` makes it very easy to build a computational laboratory notebook. First, create a new `R Markdown` file. (Choose **File > New File > R Markdown** from the RStudio menu bar.)

An R Markdown file is simply a text file. But it is interpreted in a special way that allows the text to be transformed it into a webpage (`.html`) or PDF file. You can use special syntax to render the text in different ways. Here are a few examples of R Markdown syntax:

```
*Italic text* **Bold text**

# Very large header

## Large header

### Smaller header

Unordered and ordered lists:

+ First
+ Second
    + Second 1
    + Second 2

1. This is a
2. Numbered list
```

When rendered as a PDF, the above R Markdown looks like this:



Very large header

Large header

Smaller header

Italic text **Bold text**

Unordered and ordered lists:

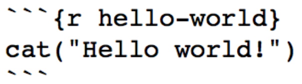
- First
- Second
    – Second 1
    – Second 2

1. This is a
2. Numbered list

You can also insert inline code by enclosing it in backticks.

The most important feature of R Markdown is that you can include blocks of code, and they will be interpreted and executed by `R`. You can therefore combine effectively the code itself with the description of what you are doing.

For example, including a code chunk in your R Markdown file,



~~~
```{r hello-world}
cat("Hello world!")
```
~~~

will render a document containing both the results and the code that run to generate those results:

```r
cat("Hello world!")
```

If you don't want to run the `R` code, but just display it, use `{r hello-world, eval = FALSE}`; if you want to show the output but not the code, use `{r hello-world, echo = FALSE}`.

You can include plots, tables, and even mathematical equations using LaTeX. In summary, when exploring your data, or describing the methods for your paper, give R Markdown a try!

You can find inspiration in this Boot Camp; the materials for Basic and Advanced Computing were written in R Markdown.

Advanced Computing – Data wrangling and plotting

Stefano Allesina *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`:

```r
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:

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

A new data type, `tibble`

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

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

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

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

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 researcher 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, changing 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:

```r
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:

```r
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*):

```r
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:

```r
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:

```r
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:

```r
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):

```r
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:

```r
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?

```r
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:

```r
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!):

```r
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):

```r
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:

```r
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:

```r
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:

```r
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`.

Exercises in groups

The file `data/Chicago_Crimes_May2016.csv` contains a list of all the crimes reported in Chicago in May 2016. Form small groups and work on the following exercises:

- **Crime map** write a function that takes as input a crime's `Primary Type` (e.g., `ASSAULT`), and draws a map of all the occurrences. Mark a point for each occurrence using `Latitude` and `Longitude`. Set the `alpha` to something like 0.1 to show brighter colors in areas with many occurrences.
- **Crimes by community** write a function that takes as input a crime's `Primary Type`, and produces a barplot showing the number of crimes per `Community area`. The names of the community areas are found in the file `data/Chicago_Crimes_CommunityAreas.csv`. You will need to `join` the tables before plotting.
- **Violent crimes** add a new column to the dataset specifying whether the crime is considered violent (e.g., `HOMICIDE`, `ASSAULT`, `KIDNAPPING`, `BATTERY`, `CRIM SEXUAL ASSAULT`, etc.)
- **Crimes in time** plot the number of violent crimes against time, faceting by community areas. (Hint: use the package `lubridate` to work with days, dates, time)
- **Dangerous day** which day of the week is the most dangerous? (Extract day of the week from `Date`)
- **Dangerous time** which time of the day is the most dangerous (Extract the hour of the day from `Date`).
- **Correlation between crimes** [difficult] which crimes tend to have the same pattern? Divide the crimes by day and type, and plot the correlation between crimes using `geom_tile` and colouring the cells according to the correlation (see `cor` for a function that computes the correlation between different columns).

Data Jujutsu I – The name game[*]

Stefano Allesina & Graham Smith *University of Chicago*

Description of the data

The Social Security Administration releases every year data on first names as reported on Social Security cards. Basically, all names that were given to 5+ people in a given year are reported (details here). The data goes back to 1880, and the latest release covers all years until 2018. The data are organized by year, and contained in a zip file that you can access at `https://www.ssa.gov/oact/babynames/names.zip` (about 7Mb). For each year, a file (e.g. `yob2012.txt`) contains information on all the names, organized in three columns (**no header**):

```
Sophia,F,22313
Emma,F,20945
Isabella,F,19099
Olivia,F,17316
Ava,F,15533
...
```

The first column is the name (with spaces and hypens removed: `Mary-Jane` → `Maryjane`), the second column the sex assigned at birth (`F` or `M`—will this change in the near future?) and the third column the number of babies for a given assigned sex that were given that name.

The challenge

1. Read the data Write code to download, parse and organize the data, building a tibble with columns `year`, `name`, `sex`, `prop`, where `prop` is the proportion of babies with a given name for a year and sex combination.

```
source("solution_name_game.R") # this is the code **you** have to write
all_names # store all information here
```

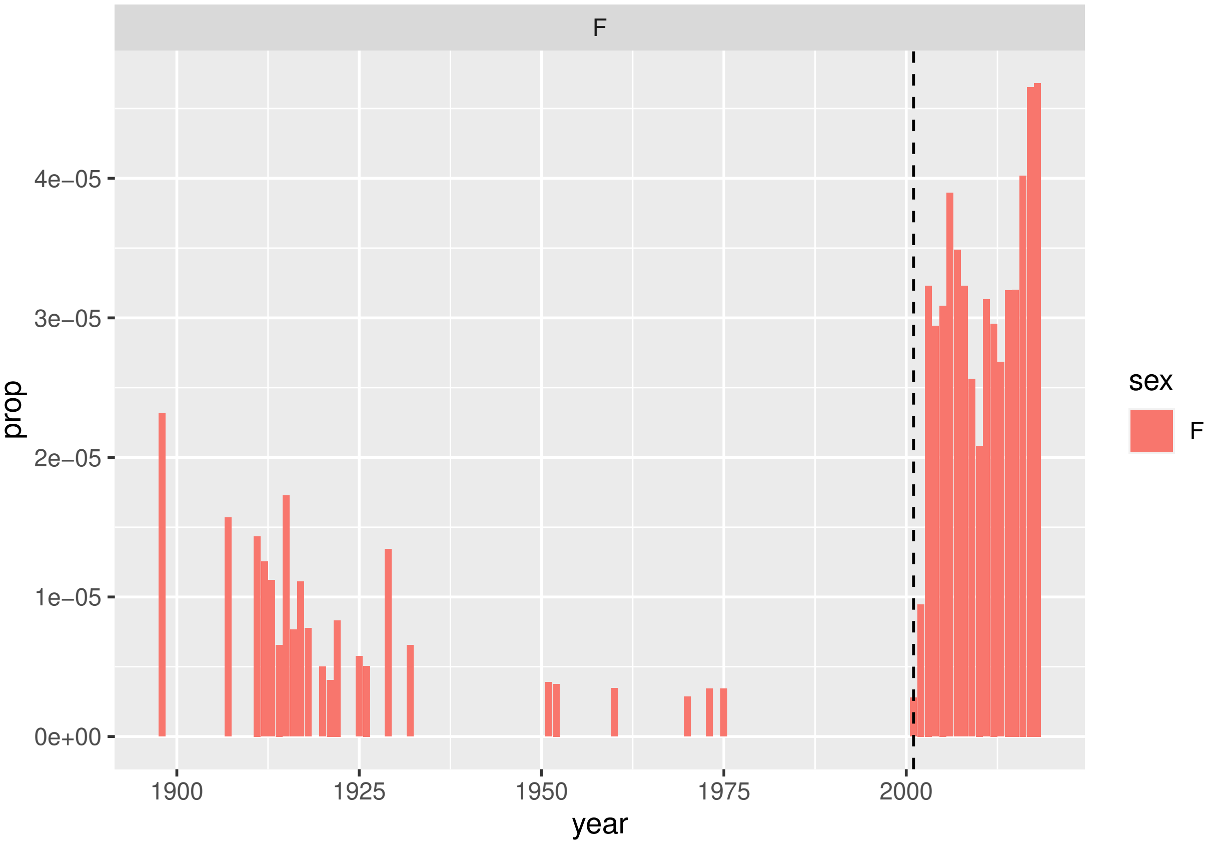
```
## # A tibble: 1,957,046 x 4
##      year name      sex     prop
##     <dbl> <chr>     <chr>  <dbl>
##   1  1880 Mary      F     0.0776
##   2  1880 Anna      F     0.0286
##   3  1880 Emma      F     0.0220
##   4  1880 Elizabeth F     0.0213
##   5  1880 Minnie    F     0.0192
##   6  1880 Margaret  F     0.0173
##   7  1880 Ida       F     0.0162
```

[*] This document is included as part of the Advanced Computing I packet for the U Chicago BSD qBio6 boot camp 2020. **Current version**: August 11, 2020; **Corresponding author**: sallesina@uchicago.edu.

```
##  8  1880 Alice     F     0.0155
##  9  1880 Bertha    F     0.0145
## 10  1880 Sarah     F     0.0142
## # ... with 1,957,036 more rows
```

2. Plot the data Write a function that plots the frequency in time, and accepts an extra parameter to highlight a particular year (a vertical dashed line marks the highlighted year). For example:

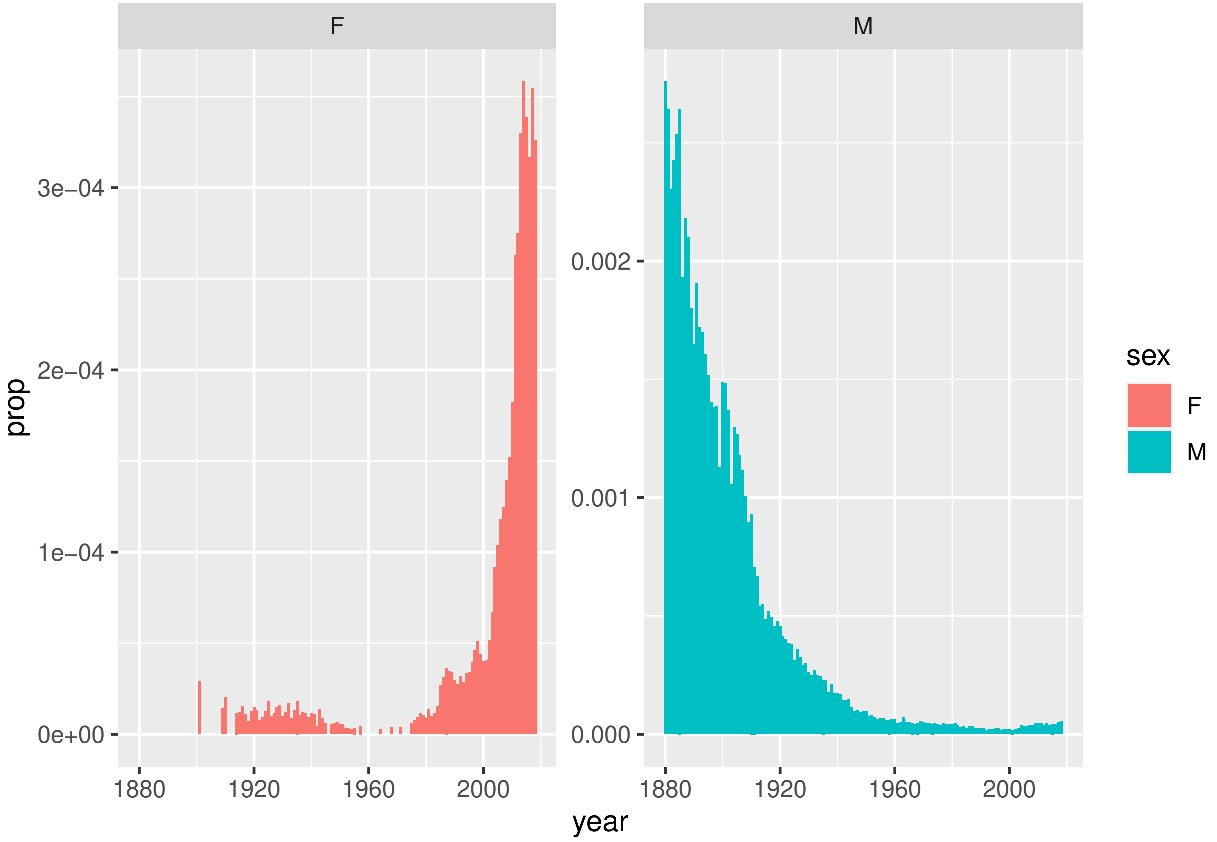
```
plot_name_in_time(data = all_names, my_name = "Hermione", highlight = 2001)
```



3. Dramatic changes in name frequencies Try to find name/year combinations where the name frequency changes significantly due to people/event in popular culture (e.g., "Harry Potter and the Sorcerer's stone" movie came out in 2001; other examples are Ariel/1989, Alanis/1995, Beyonce/1998, Neo/1999, Osama/2001, Barack/2008, Ivanka/2016, etc.). Be prepared to share the most interesting combinations with the class.

4. Changes in association between name and sex [Optional] Certain names went from being given predominantly to boys to more frequently to girls, or vice versa. Write code to discover the most impressive transitions. For example, "Charley" was predominantly assigned to boys back in the day, while today is mostly assigned to girls:

```
plot_name_in_time(data = all_names, my_name = "Charley", highlight = NULL)
```



Hints & Nifty tricks

- If you don't want to store the downloaded zip file, use a temporary file (it will be deleted by `R` automatically once you call `unlink()`):

```
temp <- tempfile()
download.file("https://www.ssa.gov/oact/babynames/names.zip",temp)
```

- Similarly, you don't need to extract all the files and store them on your disk: you can use unz to extract a file from a zip file, and use a connection to read it directly from the memory. For example:

```
con <- unz(temp, "yob2012.txt")
dt <- read.table(con, header = FALSE, sep = ",", stringsAsFactors = FALSE)
```

- Don't read the names as `factors` (the default in `read.table`) as binding together long lists of factors takes much time.
- Save your elaborated data, to save the time it takes to download and parse the data when you re-run your code. For extra brownie points, write your code such that it checks if the data have been parsed already before starting the download. You can use `file.exists()` to check whether a file is present or not.

Data Jujutsu II – PhD Trends*

Stefano Allesina & Graham Smith *University of Chicago*

Description of the data

Every year, the National Science Foundation sponsors a very large survey (with almost complete sampling) of the PhD graduates, the *Survey of Earned Doctorates* (SED). They publish statistics on the number of PhDs awarded per year, and report PhD completion by gender, field, ethnic background, etc. In particular, table 16 reports the number of PhDs awarded in total (and divided by sex) for each field of study. We are going to attempt reading the table directly from the `xlsx` files that are published by NSF.

The challenge

1. Read the data The file `urls_and_skip_NSF_SED.csv` reports the location (`url`) of the excel files for the years 2013-1018, as well as the number of lines to skip (`skip`) and the number of lines to read (`read`) for best results. Read the documentation of `read_xlsx` from the library `readxl` to see how to read the file while skipping a few lines and capping the total number of lines to be read.

```
library(tidyverse)
library(readxl)
read_csv("urls_and_skip_NSF_SED.csv")
```

```
## # A tibble: 6 x 4
##    year url                                                       skip  read
##   <dbl> <chr>                                                    <dbl> <dbl>
## 1  2018 https://ncses.nsf.gov/pubs/nsf20301/assets/data-tables/tabl~     3   274
## 2  2017 https://ncses.nsf.gov/pubs/nsf19301/assets/data/tables/sed1~     3   271
## 3  2016 https://nsf.gov/statistics/2018/nsf18304/data/tab16.xlsx         1   270
## 4  2015 https://nsf.gov/statistics/2017/nsf17306/data/tab16.xlsx         1   264
## 5  2014 https://nsf.gov/statistics/2016/nsf16300/data/tab16.xlsx         1   293
## 6  2013 https://nsf.gov/statistics/sed/2013/data/tab16.xlsx              1   284
```

Read all the files, building the tibble `tb_sed` retaining only the field and the total for each year:

```
source("solution_PhD_trends.R") # this is the code you have to write!
tb_sed
```

```
## # A tibble: 1,583 x 3
##    field                                                              total  year
##    <chr>                                                              <dbl> <dbl>
##  1 All fields                                                         55195  2018
```

[*] This document is included as part of the Advanced Computing packet for the U Chicago BSD qBio6 boot camp 2020. **Current version**: August 11, 2020; **Corresponding author**: sallesina@uchicago.edu.

```
##  2 Life sciences                                                          12780  2018
##  3 Agricultural sciences and natural resources                             1445  2018
##  4 Agricultural sciences                                                    875  2018
##  5 Agricultural economics                                                   108  2018
##  6 Agronomy, horticulture science, plant breeding, plant pathology,~        349  2018
##  7 Animal nutrition, poultry science                                         68  2018
##  8 Animal sciences, other                                                   121  2018
##  9 Food science, food technology-other                                      163  2018
## 10 Soil chemistry and microbiology, soil sciences-other                      66  2018
## # ... with 1,573 more rows
```

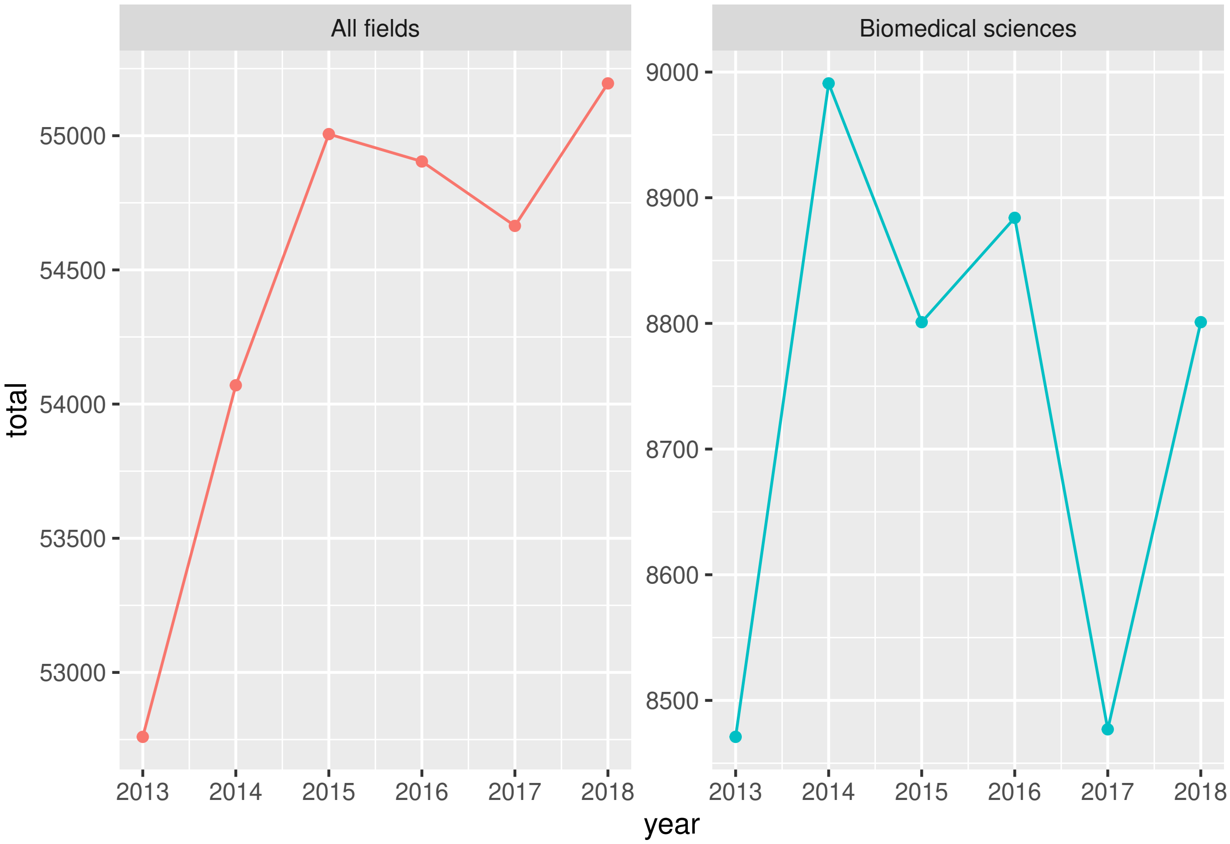
2. Standardize names and filter Notice that there are very many fields, and that the names of some fields have changed through the years (e.g., `Neurosciences, neurobiology`, `Neurosciences and neurobiology`). The file `lookup_fields_filter.csv` contains two columns: retain all the records for the fields specified in the table, and use the column `name_to_use` to standardize the names of the fields. You should end up with 18 fields (all well-represented at U of C) as well as the data for `All fields` and `Biomedical sciences`.

```
tb_sed
```

```
## # A tibble: 120 x 4
##    field                                   total  year name_to_use
##    <chr>                                   <dbl> <dbl> <chr>
##  1 All fields                              55195  2018 All fields
##  2 Biological and biomedical sciences       8801  2018 Biomedical sciences
##  3 Anatomy, developmental biology            158  2018 Developmental biology
##  4 Biochemistry (biological sciences)        811  2018 Biochemistry
##  5 Bioinformatics                            203  2018 Bioinformatics
##  6 Biometrics and biostatistics              233  2018 Biostatistics
##  7 Biophysics (biological sciences)          152  2018 Biophysics
##  8 Cancer biology                            355  2018 Cancer biology
##  9 Cell, cellular biology, and histology     218  2018 Cell biology
## 10 Computational biology                     146  2018 Computational biology
## # ... with 110 more rows
```

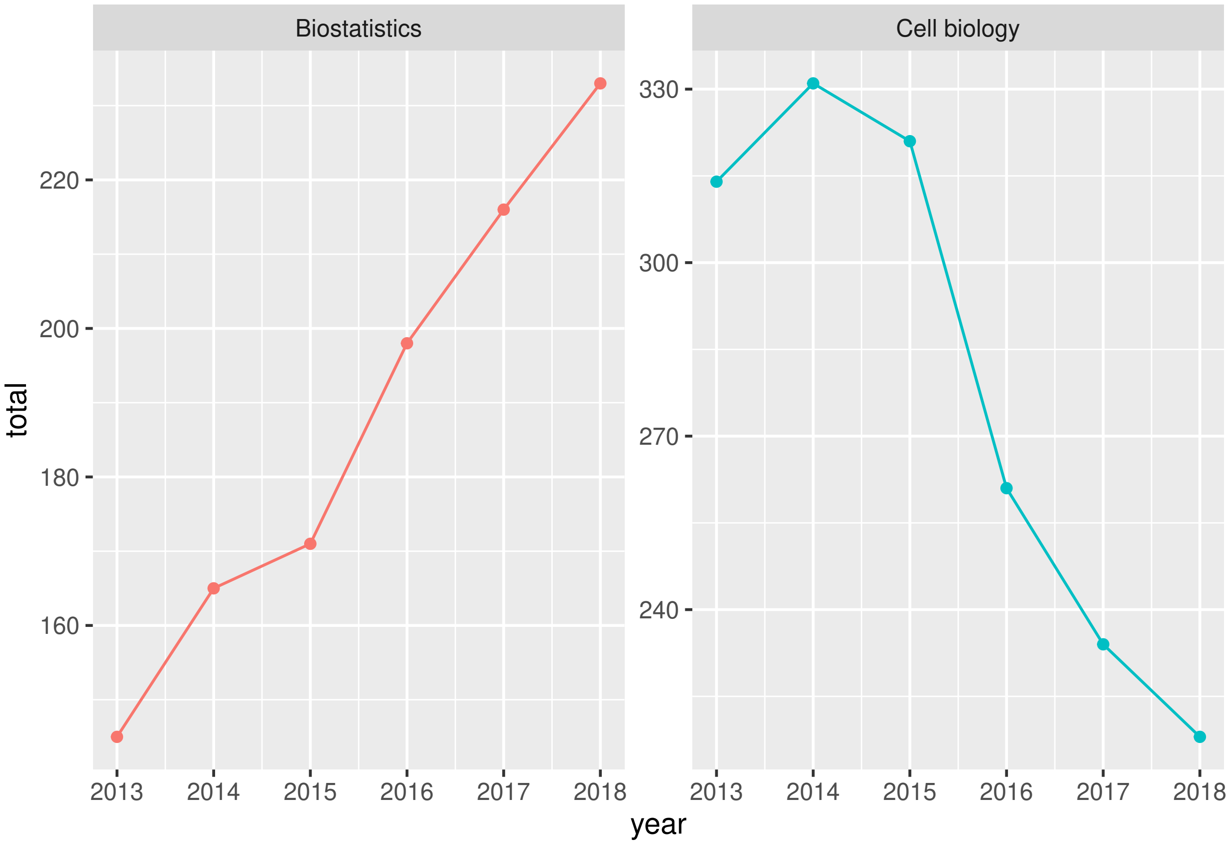
3. Plot the time series Write a generic function for plotting the number of PhDs awarded for all the fields in a tibble. For example, here are the trends for all PhDs and all PhDs in biomedical sciences:

```
tb_sed %>%
  filter(name_to_use %in% c("All fields", "Biomedical sciences")) %>%
  plot_PhD_in_time()
```



4. Fields that have changed the most Some fields have grown considerably in the past 6 years, while some have shrunk. For example:

```
tb_sed %>%
  filter(name_to_use %in% c("Cell biology", "Biostatistics")) %>%
  plot_PhD_in_time()
```



Find the fields for which the ratio between the maximum number of PhDs and the minimum number of PhDs for the period considered is the largest.

5. Correlation between time series [Optional] Compute the correlation (using the function `cor`) between the time series of any two fields. Which fields have changed in synchrony? Plot the matrix of correlations using `geom_tile`.

6. Order the matrix [Optional, requires some math] Find a good ordering for the matrix by plotting the field according the the eigenvector of the correlation matrix corresponding to the largest eigenvalue (the function `eigen` computes eigenvalues and eigenvectors of a squared matrix).

Hints & Nifty tricks

- If you don't want to store the downloaded zip file, use a temporary file (it will be deleted by `R` automatically once you call `unlink()`)
- Some lines are empty: use something like `filter(!is.na(field))` to get rid of them.
- For each year, you only need to store the number of PhD awarded.
- To force a certain order for the labels in the graph, transform them to factors, using `factor`: `my_tibble <- my_tibble %>% mutate(my_labs = factor(my_labs, levels = my_order))`

Data Jujutsu III – Papers from UofC*

Stefano Allesina & Graham Smith *University of Chicago*

Description of the data

I have collected information on the 13,140 papers published between 2011 and 2020 by researchers with a UofC affiliation in the field of biology—including all papers in multidisciplinary journals (as such, some papers from other fields are included). We are going to explore this data set to highlight the variety of research programs and publications produced by our faculty. We will use these data to test the hypothesis that `open access` publishing leads to more citations than paywalled publishing.

Recap of linear regression in R

To be able to complete the challenge, you need to know how `R` approaches linear regressions. Take $y = \{y_1, y_2, \ldots, y_n\}$ to be the variable that we want to model (containing numeric values), and suppose that we want to model y as a function of a numeric covariate $x = \{x_1, x_2, \ldots, x_n\}$. The simplest model we can choose is a linear regression:

$$y_i = \alpha + \beta x_i + \epsilon_i$$

where α is an intercept parameter, β is a slope parameter, and ϵ_i is the error we make when we predict y_i given x_i, α and β. Importantly, when we are writing this model we typically assume that ϵ_i is sampled independently from a Normal distribution with mean 0 and variance σ^2. If this is the case, we can explicitly solve for the "best" values of the parameters α, β and σ (i.e., compute their maximum likelihood estimates).

In `R`, which was born for statistics, coding the linear regression is very easy:

```r
y <- rnorm(100, mean = 1, sd = 0.5) # generate random data
x <- y + rnorm(100, mean = 0, sd = 0.1)
# fit the model
my_model <- lm(y ~ x) # this encodes the model above
```

To determine the quality of fit, run

```r
summary(my_model)
```

```
##
## Call:
## lm(formula = y ~ x)
##
```

*This document is included as part of the Advanced Computing packet for the U Chicago BSD qBio6 boot camp 2020. **Current version**: August 12, 2020; **Corresponding author**: sallesina@uchicago.edu.

```
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.229791 -0.073213  0.002596  0.066801  0.216159
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.02446    0.02053   1.191    0.236
## x            0.98086    0.01874  52.336   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.09639 on 98 degrees of freedom
## Multiple R-squared:  0.9655, Adjusted R-squared:  0.9651
## F-statistic:  2739 on 1 and 98 DF,  p-value: < 2.2e-16
```

In particular, the `Adjusted R-squared`, ranging between 0 and 1, is a good indication (for the simple case above, it corresponds to the proportion of variance explained by the model).

When covariates are not numeric values, but rather categorical variables (e.g., strings, factors), then the model takes form:

$$y_i = \alpha + \beta_1 z_{i1} + \beta_2 z_{i2} + \ldots + \beta_k z_{ik} + +\epsilon_i$$

where z_{ik} is a variable taking value 1 when the observation i belongs to category k and 0 otherwise. As such, in the model, a certain "slope" β_k is turned on when the variable belongs to category k, and switched off otherwise. In `R`, the syntax is the same.

You can include interaction terms in the model. For example, suppose that you have two numerical covariates, v and w. Then if you write:

```
lm(y ~ v + w)
```

You are fitting a model having form:

$$y_i = \alpha + \beta_1 v_i + \beta_2 w_i + \epsilon_i$$

If you write:

```
lm(y ~ v*w)
```

you are fitting a more complicated model:

$$y_i = \alpha + \beta_1 v_i + \beta_2 w_i + \beta_3 v_i w_i + \epsilon_i$$

Finally, if you write:

```
lm(y ~ v:w)
```

You only retain the interaction:

$$y_i = \alpha + \beta_3 v_i w_i + \epsilon_i$$

Similarly, if you are dealing with categories, writing $v : w$ in the code amounts to constructing a new category, with one value for each combination of v and w.

More advanced: when you have categorical values, one of them serves as a baseline value (technically, the coefficient gets absorbed into the slope). You can set manually the baseline by calling the function `relevel`, specifying a new reference category (`ref`). For example:

```
data("warpbreaks")
# contrast
warpbreaks$tension <- relevel(warpbreaks$tension, ref = "L")
summary(lm(breaks ~ wool + tension, data = warpbreaks))
# with
warpbreaks$tension <- relevel(warpbreaks$tension, ref = "M")
summary(lm(breaks ~ wool + tension, data = warpbreaks))
```

Note that the models have identical fit, but that the values corresponding to the tension categories ("L" for low, "M" for medium and "H" for high) have changed.

The challenge

1. *Read the data* The data are stored in the file `All_UofC_Bio_2011-20.csv`. Read the file, and rename the columns for easier typing:

```
au = Authors
au_ids = ‘Author(s) ID‘
year = Year
journal = ‘Source title‘
cits = ‘Cited by‘
article = ‘Document Type‘
oa = ‘Access Type‘
```

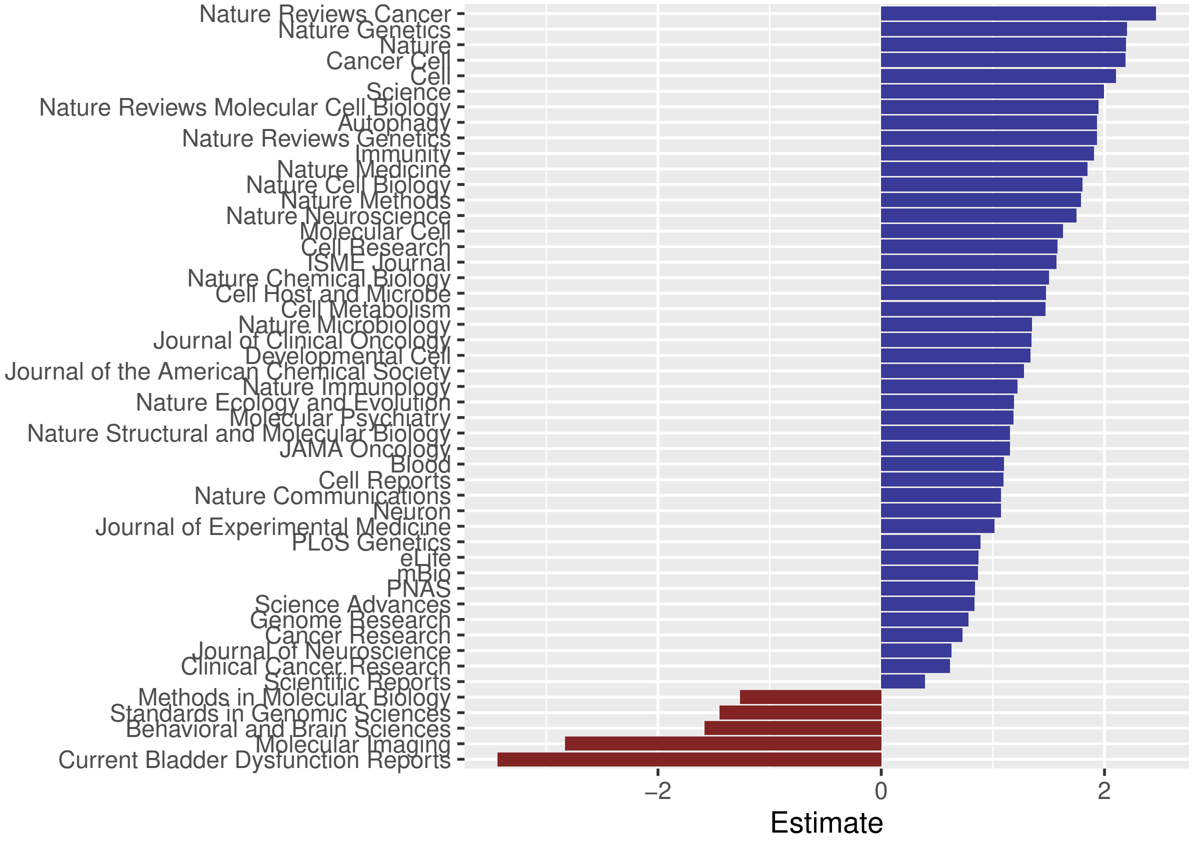
2. *Distribution of citations by year* Several studies have shown that the number of citations per paper in a given year is approximately lognormal (with better fit for older years).

Remove the papers with 0 citations, and plot the histogram for the log number of citations faceting by year. You should see that older years yield (approximately) a normal distribution. With this transformation at hand, you can attempt modeling the citations as a function of other bibliometric variables.

To start off, fit a linear model `log(cits) ~ I(2010 - year)`. Specifying `I(2010 - year)` builds a new variable containing the year -2010, so that a paper from 2011 would take value 1, from 2012 would take 2, etc. Because we expect the number of citations to grow with the age of the paper, the coefficient associated with this covariate measure the growth in number of citations for each year (to be specific, $e^{\beta} - 1$ is the proportion of increase for papers that are 1 year old).

3. Multi-authored papers The number of authors per paper varies dramatically. Count the number of authors per paper, and show that including a covariate specifying whether the paper has more than 12 authors improves the fit (store this variable in the column `multi`, and run `lm(log(cits) ~ year:multi)`). Similarly, including whether the article is a research article or a review improves the fit (column `article`).

4. Top journals Of course, research papers published in high-visibility journals (such as Nature and Science) tend to receive many more citations. We can model each journal separately as a categorical variable, and look at the distribution of effect sizes on citations: fit the model `log(cits) ~ I(2010 - year) + multi + article + journal`, and plot the effect sizes from most positive to most negative. Include only the journals with a strongly significant effect (e.g., a p-value $< 10^{-6}$ to avoid problems with multiple hypothesis testing), and draw a barplot such as the one below (obtained setting the baseline journal as "PLoS ONE" using `relevel`):



4. Effects of open access Some of the journals have an "open access option", which typically costs top dollars. Will this give your article more visibility (and therefore citations)? To test this effect, find all the journals that have published both open access and paywalled articles in the same year, and contrast the citation counts between the two subsets. To do so, extract all the papers for the journal/year combinations where you have paywalled and open access papers. Test the effect of open access by regressing:

```
log(cits) ~ year:journal:multi:article + open
```

where `year:journal:multi` fits the mean number of log citations for each year, journal and accounting for multiauthored papers and reviews, and `open` tests the effect of having published with the open access option. Is the effect positive?

5. Most productive researchers [Optional] Count how many times does each author ID appears in the data. Find the most productive authors in biology, and try extracting their names from the `au` column.

Data visualization tutorial: exploring data and telling stories using ggplot2*

Peter Carbonetto *University of Chicago*

In this lesson, you will use ggplot2 to create effective visualizations of data. The ggplot2 package is a powerful plotting interface that extends the base plotting functions in R. You will learn that creating effective visualizations hinges on good data preparation. (In reality, good data preparation can take days or weeks, but we can still illustrate some useful data preparation practices.) The main difference with Advanced Computing 1 is that we take a more in-depth look at ggplot2 and plotting strategies.

Motivation

Why plot? One reason: to gain insight into your data. This is "exploratory data visualization." Another reason: to tell a story (e.g., for a scientific paper). Both require iteration and refinement to get right. So you need to learn to create plots efficiently.

For this reason, we recommend the *programmatic approach* to data visualization. This will allow you to:

1. Create an endless variety of plots.
2. Reuse code to quickly create and revise plots.

ggplot2 is the the most powerful—and increasingly popular—approach to creating plots in R.

Setup

Download the tutorial materials to your computer, and make sure you know where to find them.

Before starting the tutorial, I suggest quiting applications that are not needed, and other "clutter", to reduce distractions.

Launch RStudio. It is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu. Also, make sure your R working directory is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this.

If you have not already done so, install these packages:

```
install.packages("ggplot2")
install.packages("cowplot")
install.packages("ggrepel")
```

*This document is included as part of the Data Visualization tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2020. **Current version**: August 10, 2020; **Corresponding author**: pcarbo@uchicago.edu. Thanks to John Novembre, Stephanie Palmer, Stefano Allesina and Matthew Stephens for their guidance.

```
install.packages("htmlwidgets")
install.packages("plotly")
```

Hands-on exericse: Do smaller dogs live longer?

In this tutorial, we will make use some data that was made available by the authors of a 2008 *Genetics* article, “Single-nucleotide-polymorphism-based association mapping of dog stereotypes.” These data are stored in a CSV file that was included with the rest of the tutorial materials.

Import the data into R

By now, you should know how to import data from a CSV file. This code reads the data from a CSV file and creates a data frame, `dogs`. (Make sure your working directory is set to the location of these data, otherwise R will not be able to find the file.)

```
dogs <- read.csv("dogs.csv",stringsAsFactors = FALSE)
```

A first look at the data

After loading data into R for the first time, I recommend getting a basic understanding of the data frame and its contents. Here are some simple commands I often use:

```
nrow(dogs)
ncol(dogs)
names(dogs)
head(dogs)
tail(dogs)
summary(dogs)
class(dogs$aod)
class(dogs$height)
class(dogs$weight)
class(dogs$shortcoat)
```

Note that “aod” stands for “age of death”.

Let’s take a closer look at the `shortcoat` column:

```
unique(dogs$shortcoat)
```

Feel free to explore the data further.

The often overlooked scatterplot

In this tutorial, we will learn about ggplot2 through one of the most basic data visualizations: the scatterplot.

The scatterplot is easily overlooked because it is so simple. But it can be one of the most effective ways to visualize scientific data. Occasionally, embellishments (e.g., varying colors, shapes, sizes, labels) can produce stunning visualizations.

Typically, the purpose of the scatterplot is to investigate whether there is an interesting—or surprising—relationship between two (continuous) variables. But the scatterplot has many other uses, particularly in exploratory data analysis. For example, the scatterplot can highlight problems with the data, e.g., "outliers". It is also useful for investigating the distribution of the data.

Although simple, we can explore the key features of ggplot2 via the scatterplot, and, in fact, its simplicity is helpful in not distracting us from our main goal, which is to learn about how to create good plots with ggplot2.

In exploratory data analysis, it is usually helpful to have a focus. Here, our aim is to investigate the anecdotal claim that dogs representing breeds of small size such as Chihuahuas live longer than those from larger-sized breeds such as Saint Bernards.

Let's begin by creating a scatterplot of weight vs. longevity using the base R function, `plot`. (No packages needed here.)

```
plot(dogs$weight,dogs$aod)
```

Question: Based on this plot, how would you describe, qualitatively, the relationship between weight and longevity?

Our first ggplot2 (with ugly code)

Now let's recreate this same scatterplot using ggplot2. The benefits of ggplot2 over `plot` will not be immediately clear.

```
library(ggplot2)
p1 <- ggplot(dogs,aes_string(x = "weight",y = "aod"))
p1 <- ggplot_add(geom_point(),p1)
```

Where is the plot? We need to call `print`, which tells R to draw the plot to the screen:

```
print(p1)
```

This code is indeed quite cumbersome, and some simpler code can accomplish the same thing:

```
p1 <- ggplot(dogs,aes_string(x = "weight",y = "aod")) +
  geom_point()
```

For the moment I want to focus on the "uglier" code because it highlights better the key elements of a ggplot2 plot:

1. The first input is a data frame.

2. The second input is an "aesthetic mapping" that defines how columns are mapped to features of the plot (axes, shapes, colors, *etc*).
3. A "geom", short for "geometric object", specifies the type of plot. ggplot2 has an excellent on-line reference (ggplot2.tidyverse.org) explaining all the "geoms", from bar charts to contour plots, with code examples for each.
4. Rather than draw directly to the screen, ggplot2 creates a *ggplot object*, which can be drawn to the screen at any time by calling `print`.

The distinguishing feature of ggplot2—and what makes ggplot2 so powerful—is that plots are created by *adding layers*. This layering is what allows for an infinite variety of plots to be created with ggplot2. Once you have mastered this layering, you will find that you can use ggplot2 to create sophisticated, high-quality plots with little effort. Further, the layering approach means that ggplot2 is easily extendible, and many R packages have been developed to enhance ggplot2. (We will use two of these packages, ggrepel and cowplot.)

In the remainder, we will explore this layering approach to plotting.

Some improvements

Our plot can be improved. Here we add "layers" to adjust plot's appearance, including:

- Clearly label the axes.
- Add a title.
- Change the location of the axis tick marks.
- Choose a more attractive theme (I'm partial to the cowplot theme).

Can you think of other ways the plot can be improved?

```
library(cowplot)
p1 <- ggplot(dogs,aes_string(x = "weight",y = "aod")) +
  geom_point() +
  scale_x_continuous(breaks = seq(0,160,40)) +
  labs(x = "body weight (lbs)",
       y = "longevity (years)",
       title = "longevity vs. size") +
  theme_cowplot()
```

Plot the best-fit line

It has been estimated that an increase of 28 lbs in a dog's body weight corresponds to about a 1-year drop in expected lifespan (with a maximum lifespan of about 13 years). Let's see how well this estimate agrees with the data by plotting the line that best fits these data (the "least-squares" estimate).

```
fit <- lm(aod ~ weight,dogs)
```

Add two "abline" layers to compare the estimates:

```
p2 <- p1 +
  geom_abline(color = "dodgerblue",linetype = "dashed",
              intercept = coef(fit)[1],slope = coef(fit)[2]) +
  geom_abline(color = "darkorange",linetype = "dashed",
              intercept = 13,slope = -1/28)
```

Notice how easy it was to add layers to the existing plot. You can't do this quite so easily with other plotting interfaces. Also notice that this geom, like most geoms, has properties such as color and size that can be adjusted.

Here I created a new plot object, `p2`, rather than overwrite the existing one, `p1`. Now that I have both plots, I can decide later on which one I want to keep.

How well do the two linear regressions agree?

Add the breed names

It would be helpful if we could tell which breeds are being plotted. Adding text to a ggplot is done in two steps: (1) map a column to the "label" aesthetic, (2) add a `geom_text` or `geom_label` geom.

Adding all the breed names will not work, however—there simply isn't enough real estate on the plot to accommodate all the breed names. Instead, a "smarter" function is `geom_text_repel` from the ggrepel package, which adds the labels in a way that makes them more readable, and only adds them to the plot when possible. Let's appreciate again how simply this is accomplished.

```
library(ggrepel)
p3 <- p2 + geom_text_repel(mapping = aes_string(label = "breed"),
                           size = 2.5,color = "gray")
```

I chose to draw the labels in light gray so as to not drown out the scatterplot. Feel free to adjust the size, color and other properties.

Also notice that the labels are automatically redrawn as the plot is resized—give it a try.

Which breeds are outliers from the linear trend?

A surprising subtlety with color

Other aesthetics—color, size, shape, *etc*—can also be used enhance a scatterplot and tell a good story. In the last chapter of our scatterplot, we will visualize the `shortcoat` column with color, and in the process discover a complication with adding color to a plot.

The `shortcoat` column, you may recall, had values of 0 or 1 (with a few NAs).

```
dogs$shortcoat
```

In principle, varying the color of the points should be straightforward.

```
p4 <- ggplot(dogs,aes_string(x = "weight",y = "aod",label = "breed",
                             color = "shortcoat")) +
  geom_point() +
  theme_cowplot()
```

It “worked” in the sense that we were able to create a plot with color, but what we did was not completely correct. What is incorrect?

This isn’t a problem with the data itself. Rather, it is a problem of *data representation*: the discrete-valued, or *categorical* data are stored as a *continuously-valued* variable (numeric). The key is to change how the data are represented, and convert to a categorical variable, which in R is called a “factor”:

```
dogs$shortcoat <- factor(dogs$shortcoat)
```

This illustrates the more general point that careful data preparation is important for creating good plots.

```
p4 <- ggplot(dogs,aes_string(x = "weight",y = "aod",label = "breed",
                             color = "shortcoat")) +
  geom_point() +
  theme_cowplot()
```

One the few complaints I have with ggplot2 is that the default colors choices are poor. So you will often need to make adjustments to the color scheme. One rule-of-thumb is that “warmer” colors (e.g., orange, red) tends to draw the reader’s attention. There are several good resources on use of color in data visualization, and I will mention a couple here: Color Brewer (https://colorbrewer2.org); and a short article, “Color blindness”, by Bang Wong in *Nature Methods* (2011). For more extensive discussion, see “Fundamentals of data visualization” by Claus Wilke. In our scatterplot, the best color choice is less clear, so I will let you experiment with different choices. To override the color defaults, add a “scale_color_manual” layer, for example,

```
p5 <- p4 + scale_color_manual(values = c("firebrick","tomato"),
                              na.value = "black")
```

There are several different ways to specify colors, e.g., by name or RGB value. I prefer the named colors; to get the full list of named colors in R, run `colors()`. Here, I chose two similar colors because I didn’t want `shortcoat` to be a major focus of the plot. Also, notice that the color of NAs is controlled by a separate argument.

The function that controls the color of a discrete variable has an odd name: `scale_color_manual`. This is because, in ggplot2, all methods that control the mapping of variables to colors, shapes, sizes, axes, *etc*, start with `scale_`. For example, plotting weight on the square-root scale could help to better space out the breeds with smaller weights, and can be done by setting the `trans` argument (short for “transformation”) in `scale_x_continuous`:

```
p5 + scale_x_continuous(trans = "sqrt")
```

Let's add back in the breed names and the regression lines for our final plot:

```
p5 <- p5 +
  geom_text_repel(size = 2.5,color = "gray") +
  geom_abline(color = "dodgerblue",linetype = "dashed",
              intercept = coef(fit)[1],slope = coef(fit)[2]) +
  geom_abline(color = "darkorange",linetype = "dashed",
              intercept = 13,slope = -1/28)
```

Obviously, we only touched the surface of ggplot2. But once you are comfortable with these basic elements, you will find that almost everything else in ggplot2 is a variation of what we covered in this lesson.

Optional exercise: Try varying shape instead of color. Use `scale_shape_manual` to select the shapes. Run `plot(0:23,pch = 0:23)` to look up the R shape numbers.

Optional exercise: Try varying size of the points instead of their color. What problem do you run into, and what is the solution?

Save your work

This is a good point to save our work in an image file that can be shared with others.

```
ggsave("dogs.pdf",p4,height = 4,width = 4.5)
ggsave("dogs.png",p4,height = 4,width = 4.5)
```

Question: How does the PDF and PNG differ? When should you save the plot as a PDF, and when should you save as a PNG?

Programming challenge: Mapping the genetic basis of physiological and behavioral traits in outbred mice

In this programming challenge, you will use simple visualizations to gain insight into biological data.

You are working in a lab studying studying the genetics of physiological and behavioral traits in mice. The lab has just completed a large study of mice from an outbred mouse population, "CFW" ("Carworth Farms White"). The aim of the study is to identify genetic contributors to variation in behaviour and musculoskeletal traits.

Note: These challenges are roughly ordered in increasing level of complexity. Do not be discouraged if you have difficulty completing every one.

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or the UofC-hosted Google Drive.) The aim is not just to complete the challenges, but also to do collaboratively; all team members should be included, and should have the opportunity to contribute and learn from each other.

Instructions

- Locate the files for this exercise on your computer (see "Materials" below).
- Make sure your R working directory is set to the same directory containing the tutorial materials; use `getwd()` to check this.
- Some of the programming challenges require uploading an image file containing a plot. Use `ggsave` to save your plot as a file. Any standard image format (e.g., PDF, PNG) is acceptable.
- No additional R packages are needed beyond what you used in the hands-on exercises above.

Materials

- **pheno.csv:** CSV file containing physiological and behavioral phenotype data on 1,219 male mice from the CFW outbred mouse stock. Data are from Parker *et al*, 2016. Use **readpheno.R** to read the phenotype data from the CSV file into a data frame. After filtering out some of the samples, this script should create a new data frame, `pheno`, containing phenotype data on 1,092 samples (rows).
- **hmdp.csv:** CSV file containing bone-mineral density measurements taken in 878 male mice from the Hybrid Mouse Diversity Panel (HMDP). Data are from Farber *et al*, 2011. To load the data into your R environment, run this code:

  ```
  hmdp <- read.csv("hmdp.csv", stringsAsFactors = FALSE)
  ```

  This will create a data frame, `hmdp`, containing BMD data on 878 mice (rows).
- **gwscan.csv:** CSV file containing results of a "genome-wide scan" for abnormal BMD. Association p-values were computed using GEMMA 0.96. To read the results of the genome-wide scan, run the following code:

  ```
  gwscan <- read.csv("gwscan.csv", stringsAsFactors = FALSE)
  gwscan <- transform(gwscan, chr = factor(chr, 1:19))
  ```

  This will create a data frame, `gwscan`. Each row of the data frame is a genetic variant (a single nucleotide polymorphism, or "SNP"). The columns are chromosome ("chr"), base-pair position on the chromosome ("pos"), and the p-value for a test of association between variant genotype and trait value ("abnormalBMD"). The value stored in the "abnormalBMD" column is $-\log_{10}(P)$, where P is the p-value.
- **geno_rs29477109.csv:** CSV file containing estimated genotypes at one SNP (rs29477109) for 1,038 CFW mice. Use the following code to read the genotype data into your R environment:

  ```
  geno <- read.csv("geno_rs29477109.csv", stringsAsFactors = FALSE)
  geno <- transform(geno, id = as.character(id))
  ```

  This will create a new data frame, `geno`, with 1,038 rows (samples). The genotypes are encoded as "dosages"—that is, the expected number of times the alternative allele is observed in the genotype. This will be an integer (0, 1, 2), or a real number between 0 and 2 when there is some uncertainty in the estimate of the genotype. For this SNP, the reference allele is T and the alternative allele is C. Therefore, dosages 0, 1 and 2 correspond to genotypes TT, CT and CC, respectively (genotypes CT and TC are equivalent).

- **wtccc.png:** Example genome-wide scan ("Manhattan plot") from Fig. 4 of the WTCCC paper. The *p*-values highlighted in green show the regions of the human genome most strongly associated with Crohn's disease risk.

A couple tips

- Some "geoms" you may find useful: `geom_point`, `geom_histogram`, `geom_boxplot`.
- In some cases it may be useful to convert to a *factor*.

Part A: Exploratory analysis of muscle development and conditioned fear data

Your first task is to create plots to explore the data.

1. A basic initial step in an exploratory analysis is to visualize the distribution of the data. It is often convenient if the distribution is normal, or "bell shaped".
   - Visualize the empirical distribution of tibialis anterior (TA) muscle weight (column "TA") with a histogram. Units are mg. *Hint:* Try using function `geom_histogram`.
   - Is the distribution of TA weight roughly normal? Are there mice with unusually large or unusually small values ("outliers")? If so, how many "outliers" are there? (Unusually small or large values can lead to misleading results in some statistical tests.)
2. It is also important to understand relationships among measured quantities. For example, the development of the tibia bone (column "tibia") could influence TA muscle weight. Create a scatterplot (`geom_point`) to visualize the relationship between TA weight and tibia length. (Tibia length units are mm.) Based on this plot, what can you say about the relationship between TA weight and tibia length? Quantify this relationship by fitting a linear model, before and after removing the outlying TA values. (*Hint*: Use the `lm` and `summary` functions. See also the "r.squared" return value in `help(summary.lm)`.)
3. The "AvToneD3" column contains data collected from a behavioral test called the "Conditioned Fear" test.
   - Visualize the empirical distribution of AvToneD3 ("freezing to cue") with a histogram. Is the distribution of AvToneD3 approximately normal?
   - Freezing to cue is a proportion (a number between 0 and 1). A common way to obtain a more "normal" quantity is to transform it using the "logit" function[1]. Visualize the empirical distribution of the logit-transformed phenotype. Is the transformed phenotype more "bell shaped"? After the transformation, do you observe unusually small or unusually large values?
   - A common concern with behavioral tests is that the testing devices can lead to measurement error. It is especially a concern when multiple devices are used, as the devices can give slightly different measurements, even after careful calibration. Create a plot

[1] R code: `logit <- function(x) log((x + 0.001) / (1 - x + 0.001))`

to visualize the relationship between (transformed) freezing to cue and the device used ("FCbox" column). *Hint:* Try a boxplot (`geom_boxplot`). Based on this plot, does the apparatus used affect these behavioral test measurements?

Part B: Exploratory analysis of bone-mineral density data

Now you will examine data on bone-mineral density (BMD) in mice. This is a trait that is important for studying human diseases such as osteoporosis (units are mg/cm^2).

- Plot the distribution of BMD in CFW mice (see column "BMD"). What is most notable about the distribution?
- Compare these data against BMD measurements taken in a "reference" mouse population, the Hybrid Mouse Diversity Panel. To compare, create two histograms, and draw them one on top of the other. What difference do you observe in the BMD distributions? For a correct comparison, you will need to account for: (1) BMD in CFW mice was measured in the femurs of male mice only; (2) BMD in HMDP mice was recorded in g/cm^2. *Hints:* Functions `xlim` and `labs` from the ggplot2 package, and `plot_grid` from the cowplot package, might be useful for creating the plots. The `binwidth` argument in `geom_histogram` may also be useful.

Part C: Mapping the genetic basis of osteopetrotic bones

A binary trait, "abnormal BMD", was defined that signals whether an individual mouse had "abnormal", or osteopetrotic, bones. It takes a value of 1 when BMD falls on the "long tail" of the distribution (BMD greater than 90 mg/cm^2), otherwise zero.

GEMMA was used to carry out a "genome-wide association study" (GWAS) for this trait; that is, support for association with abnormal BMD was evaluated at 79,824 genetic variants (single nucleotide polymorphisms, or "SNPs") on chromosomes 1–19. At each SNP, a p-value quantifies the support for an association with abnormal BMD.

1. Your first task is to get an overview of the association results by creating a "Manhattan plot". Follow as closely as possible the provided prototype, **wtccc.png**, which shows a genome-wide scan for Crohn's disease. (Don't worry about highlighting the strongest p-values in green.) *Hints:* Replicating some elements of this plot may be more challenging than others, so start with a simple plot, and try to improve on it. Recall the adage that creating plots requires relatively little effort *provided the data are in the right form*—consider adding appropriate columns to the `gwscan` data frame. Functions from the ggplot2 package that you may find useful for this exercise include `geom_point`, `scale_color_manual` and `scale_x_continuous`.
   - In your plot, you should observe that the most strongly associated SNPs cluster closely together in small regions of the genome. This is common—it is due to a genetic phenomenon known as linkage disequilibrium (LD). It is a consequence of low recombination rates between markers in small populations. How many SNPs have "strong" statistical support for association with abnormal BMD, specifically with a $-\log_{10} p$-value > 6? How many distinct regions of the genome are strongly associated with abnormal BMD at this p-value threshold?
   - What p-value does a $-\log_{10} p$-value of 6 correspond to?
   - Using your plot, identify the "distinct region" (this is called a "quantitative trait locus", or QTL) with the strongest association signal. What is, roughly, the size of the

QTL in Megabases (Mb) if we define the QTL by base-pair positions of the SNPs with $-\log_{10} p$-value > 6? Using the UCSC Genome Browser, get a rough count of the number of genes that are transcribed in this region. Within this QTL, Parker *et al*, 2016 identified *Col1a1* as a candidate BMD gene. Was this gene one of the genes included in your count? *Hint:* All SNP positions are based on NCBI Mouse Genome Assembly 38 (mm10, December 2011).

2. Your next task is to visualize the relationship between genotype and phenotype. From the genome-wide scan of abnormal BMD, you should find that rs29477109 is the SNP most strongly associated with abnormal BMD. Here you will look closely at the relationship between BMD and the genotype at this SNP. In developing your visualization, consider that:
   - The samples listed in the phenotype and genotype tables are not the same. So you will need to align the two tables to properly show analyze the relationship. *Hint:* Function `match` could be useful for this.
   - The genotypes, stored in file **geno_rs29477109.csv**, are encoded as "dosages" (numbers between 0 and 2). You could start with a scatterplot of BMD vs. dosage. But ultimately it is more effective if the genotypes (CC, CT and TT) are plotted instead. *Hints:* In effect, what you need to do is convert from a continuous variable (dosage) to a discrete variable (genotype). One approach is to create a `factor` column from the "dosage" column. For dosages that are not exactly 0, 1 or 2, you could simply round to the nearest whole number. A boxplot is recommended; see function `geom_boxplot`.

   Based on your plot, how would describe (in plain language) the relationship between the genotype and BMD?

Notes

Code summary

This is the minimal code reproducing the final scatterplot:

```
library(ggplot2)
library(ggrepel)
library(cowplot)
dogs <- read.csv("dogs.csv",stringsAsFactors = FALSE)
dogs$shortcoat <- factor(dogs$shortcoat)
fit <- lm(aod ~ weight,dogs)
p5 <- ggplot(dogs,aes_string(x = "weight",y = "aod",label = "breed",
             color = "shortcoat")) +
  geom_point() +
  geom_text_repel(size = 2.5,color = "gray") +
  geom_abline(color = "dodgerblue",linetype = "dashed",
              intercept = coef(fit)[1],slope = coef(fit)[2]) +
  geom_abline(color = "darkorange",linetype = "dashed",
              intercept = 13,slope = -1/28) +
  scale_color_manual(values = c("firebrick","tomato"),
                     na.value = "black") +
  theme_cowplot()
```

An interactive plot

Here is an example of a plot that allows the data to be explored *interactively*:

```
library(plotly)
library(htmlwidgets)
p <- plot_ly(data = dogs,type = "scatter",mode = "markers",
             x = ~weight,y = ~aod,color = ~shortcoat,
             text = ~sprintf("%s\n (%0.1f lbs, %0.1f years)",breed,weight,aod),
             marker = list(line = list(color = "white",width = 1),size = 9),
             hoverinfo = "text",width = 600,height = 500)
p <- layout(p,
            xaxis = list(title = "weight (lbs)"),
            yaxis = list(title = "longevity (years)"),
            hoverlabel = list(bgcolor = "white",bordercolor = "black"))
saveWidget(p,"dogs.html",selfcontained = TRUE)
```

Plotly is a powerful package for creating interactive plots, with an interface similar to ggplot2.

Useful online resources

- ggplot2 reference, where you will also find a ggplot2 cheat sheet. (This cheat sheet is also included in the tutorial packet.)
- Fundamentals of Data Visualization by Claus Wilke.

Sources

- The dogs breeds data were downloaded from the *Genetics* journal website.
- The CFW phenotype and genotype data were downloaded from Data Dryad.

License

Except where otherwise noted, all instructional material in this repository is made available under the Creative Commons Attribution license (CC BY 4.0). And, except where otherwise noted, the source code included in this repository are made available under the OSI-approved MIT license. For more details, see the `LICENSE.md` file included in the tutorial packet.

Reproducible research with workflowr

workflowr version 1.6.2

John Blischak and Matthew Stephens

2020-08-16

Introduction

The workflowr R package makes it easier for you to organize, reproduce, and share your data analyses. This short tutorial will introduce you to the workflowr framework. You will create a workflowr project that implements a small data analysis in R, and by the end you will have a working website that you can use to share your work. Please follow the setup instructions in the next section prior to the workshop.

Workflowr combines literate programming with R Markdown and version control with Git to generate a website containing time-stamped, versioned, and documented results. By the end of this tutorial, you will have a website hosted on GitHub Pages that contains the results of a reproducible statistical analysis.

Setup

1. Install R
2. Install RStudio
3. Install workflowr from CRAN:

   ```
   install.packages("workflowr")
   ```

4. Create an account on GitHub

To minimize the possibility of any potential issues with your computational setup, you are encouraged to update your version of RStudio (`Help -> Check for Updates`) and update your R packages:

```
update.packages()
```

If you do encounter any issues during the tutorial, consult the Troubleshooting section for solutions to the most common problems.

Organize your research

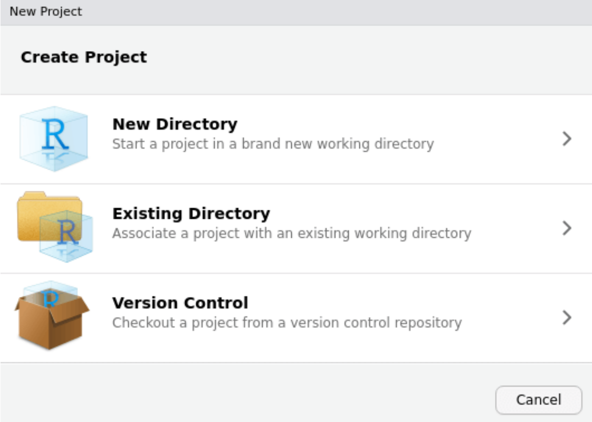
To help you stay organized, workflowr creates a project directory with the necessary configuration files as well as subdirectories for saving data and other project files. This tutorial uses the RStudio project template for workflowr, but note that the same can be achieved via the function `wflow_start()`.

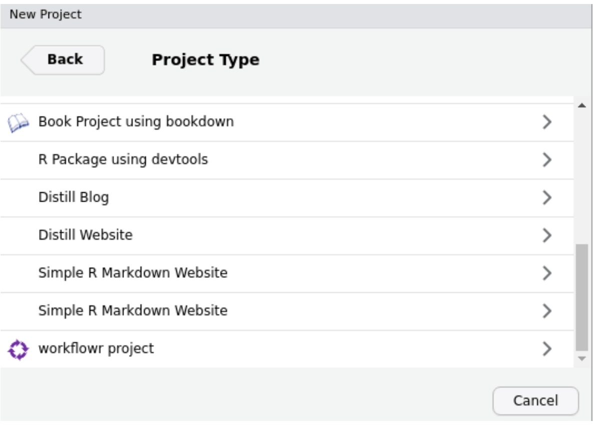
To start your workflowr project, follow these steps:

1. Open RStudio.
2. In the R console, run `wflow_git_config()` to register your name and email with Git. This only has to be done once per computer. If you've used Git on this machine before, you can skip this step. For a better GitHub experience, use the same email you used to register your GitHub account.

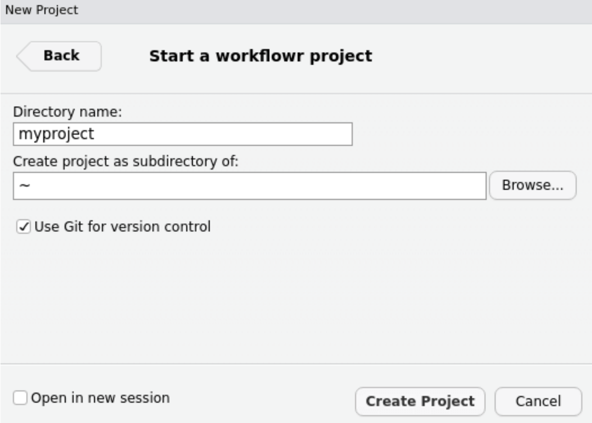
```r
library(workflowr)
wflow_git_config(user.name = "First Last", user.email = "email@domain.com")
```

3. In the menu bar, choose `File -> New Project`.
4. Choose `New Directory` and then scroll down the list of project types to select `workflowr project`. If you don't see the workflowr project template, go to Troubleshooting.


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel



5. Type `myproject` (or a more inventive name if you prefer) as the directory name, choose where to save it on your computer, and choose `Create Project`.



Directory name:
myproject
Create project as subdirectory of:
~
☑ Use Git for version control
□ Open in new session

RStudio will create a workflowr project `myproject` and opened the project in RStudio. Under the hood, RStudio is running a workflowr command `wflow_start()` - so if you prefer to start a new project from the console instead of using the RStudio menus then you could use `wflow_start()`.

Take a look at the workflowr directory structure in the Files pane, which should be something like this:

```
myproject/
|-- .gitignore
|-- .Rprofile
|-- _workflowr.yml
|-- analysis/
|   |-- about.Rmd
|   |-- index.Rmd
|   |-- license.Rmd
|   |-- _site.yml
|-- code/
|   |-- README.md
|-- data/
|   |-- README.md
|-- docs/
|-- myproject.Rproj
|-- output/
|   |-- README.md
|-- README.md
```

The most important directory for you to pay attention to now is the `analysis/` directory. This is where you should store all your analyses as R Markdown (Rmd) files. Other directories created for your convenience include `data/` for storing data, and `code/` for storing long-running or supplemental code you don't want to include in an Rmd file. Note that the `docs/` directory is where the website HTML files will be created and stored by workflowr, and should not be edited by the user.

Build your website

The files and directories created by workflowr are already almost a website! The only thing missing are the crucial `html` files. Take a look in the `docs/` directory where the html files for your website need to be created… notice that it is sadly empty.

In workflowr the html files for your website are created in the `docs/` directory by knitting (or "building") the `.Rmd` files in the `analysis/` directory. When you knit or build those files – either by using the knitr button, or by typing `wflow_build()` in the console – the resulting html files are saved in the docs directory.

The `docs/` directory is currently empty because we haven't run any of the `.Rmd` files yet. So now let's run these files. We will do it both ways, using both the knit button and using `wflow_build()`:

1. Open the file `analysis/index.Rmd` and knit it now. You can open it by using the files pane, or by typing `wflow_open("analysis/index.Rmd")` in the R console. You knit the file by pressing the knit button in RStudio.
2. There are also two other `.Rmd` files in the `analysis` directory. Build these by typing `wflow_build()` in the R console. This will build all the R Markdown files in `analysis/`, and save the resulting html files in `docs/`. (Note, it won't re-build `index.Rmd` because you have not changed it since running it before, so it does not need to.[1]

Ignore the warnings in the workflowr report for now; we will return to these later.

[1] The default behavior when `wflow_build()` is run without any arguments is to build any R Markdown file that has been edited since its corresponding HTML file was last built.

Collect some data!

To do an interesting analysis you will need some data. Here, instead of doing a time-consuming experiment, we will use a convenient built-in data set from R. While not the most realistic, this avoids any issues with downloading data from the internet and saving it correctly. The data set `ToothGrowth` contains the length of the teeth for 60 guinea pigs given 3 different doses of vitamin C either via orange juice (`OC`) or directly with ascorbic acid (`VC`).

1. To get a quick sense of the data set, run the following in the R console.

   ```
   data("ToothGrowth")
   head(ToothGrowth)
   ```

   ```
   ##    len supp dose
   ## 1  4.2   VC  0.5
   ## 2 11.5   VC  0.5
   ## 3  7.3   VC  0.5
   ## 4  5.8   VC  0.5
   ## 5  6.4   VC  0.5
   ## 6 10.0   VC  0.5
   ```

   ```
   summary(ToothGrowth)
   ```

   ```
   ##       len        supp         dose      
   ##  Min.   : 4.20   OJ:30   Min.   :0.500  
   ##  1st Qu.:13.07   VC:30   1st Qu.:0.500  
   ##  Median :19.25           Median :1.000  
   ##  Mean   :18.81           Mean   :1.167  
   ##  3rd Qu.:25.27           3rd Qu.:2.000  
   ##  Max.   :33.90           Max.   :2.000  
   ```

   ```
   str(ToothGrowth)
   ```

   ```
   ## 'data.frame':    60 obs. of  3 variables:
   ##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
   ##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
   ##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
   ```

2. To mimic a real project that will have external data files, save the `ToothGrowth` data set to the `data/` subdirectory using `write.csv()`.

   ```
   write.csv(ToothGrowth, file = "data/teeth.csv")
   ```

Understanding paths

Look at that last line of code. Where will the file be saved on your computer? To understand this very important issue you need to understand the idea of "relative paths" and "working directory".

Before explaining these ideas, let us consider a different way we could have saved the file. Suppose we had typed

```
    write.csv(ToothGrowth, file = "C:/Users/GraceHopper/Documents/myproject/data/teeth.csv")
```

Then it is clear exactly where on the computer we want the file to be saved. Specifying the file location this very explicit way is called specifying the "full path" to the file. It is conceptually simple. But it is also a pain for many reasons – it is more typing, and (more importantly) if we move the project to a different computer it will likely no longer work because the paths will change!

Instead we typed

```
write.csv(ToothGrowth, file = "data/teeth.csv")
```

Specifying the file location this way is called specifying the "relative path" because it specifies the path to the file *relative to the current working directory*. This means the full path to the file will be obtained by appending the specified relative path to the (full) path of the current working directory. For example, if the current working directory is `C:/Users/GraceHopper/Documents/myproject/` then the file will be saved to `C:/Users/GraceHopper/Documents/myproject/data/teeth.csv`. If the current working directory is `C:/Users/Matt124/myproject` then the file will be saved to `C:/Users/Matt124/myproject/data/teeth.csv`.

So, what is your current working directory? When you start or open a workflowr project in RStudio (e.g. by clicking on `myproject.Rproj`) RStudio will set the working directory to the location of the workflowr project on your computer. So your current working directory should be the location you chose when you started your workflowr project. You can check this by typing `getwd()` in the R console.

Notice how, by using relative paths, the code used here works for you whatever operating system you are on and however your computer is set up! *You should always use relative paths where possible because it can help make your code easier for others to run and easier for you to run on different computers and different operating systems.*

Create a new analysis

So, now we have some data, we are ready to perform a small analysis. To start a new analysis in RStudio, use the `wflow_open()` command.

1. In the R console, open a new R Markdown file by typing

   ```
   wflow_open("analysis/teeth.Rmd")
   ```

   Notice that we again used a relative path! Relative paths are good for opening files as well as saving files. This command should create a new `.Rmd` file in the `analysis` subdirectory of your workflowr project, and open it for editing in RStudio. The file looks pretty much like other `.Rmd` files, but in the header note that workflowr provides its own custom output format, `workflowr::wflow_html`. The other minor difference is that `wflow_open()` adds the editor option `chunk_output_type: console`, which causes the code to be executed in the R console instead of within the document. If you prefer the results of the code chunks be embedded inside the document while you perform the analysis, you can delete those lines (note that this has no effect on the final results, only on the display within RStudio).

2. Copy the code chunk below and paste it at the bottom of the file `teeth.Rmd`. The code imports the data set from the file you previously created[2]. Execute the code in the R console by clicking on the Run button or using the shortcut `Ctrl/CMD+Enter`.

   ~~~
   ```{r import-teeth}
   teeth <- read.csv("data/teeth.csv", row.names = 1)
   head(teeth)
   ```
   ~~~

   Note: if you copy and paste this chunk, make sure to remove any spaces before each of the backticks (` ``` `) so that they will be correctly recognized as indicating the beginning and end of a code chunk.

3. Next create some boxplots to explore the data. Copy the code chunk below and paste it at the bottom of the file `teeth.Rmd`. Execute the code to see create the plots.

   ~~~
   ```{r boxplots}
   ~~~

[2] Note that the default working directory for a workflowr project is the root of the project. Hence the relative path is `data/teeth.csv`. The working directory can be changed via the workflowr option `knit_root_dir` in `_workflowr.yml`. See `?wflow_html` for more details.

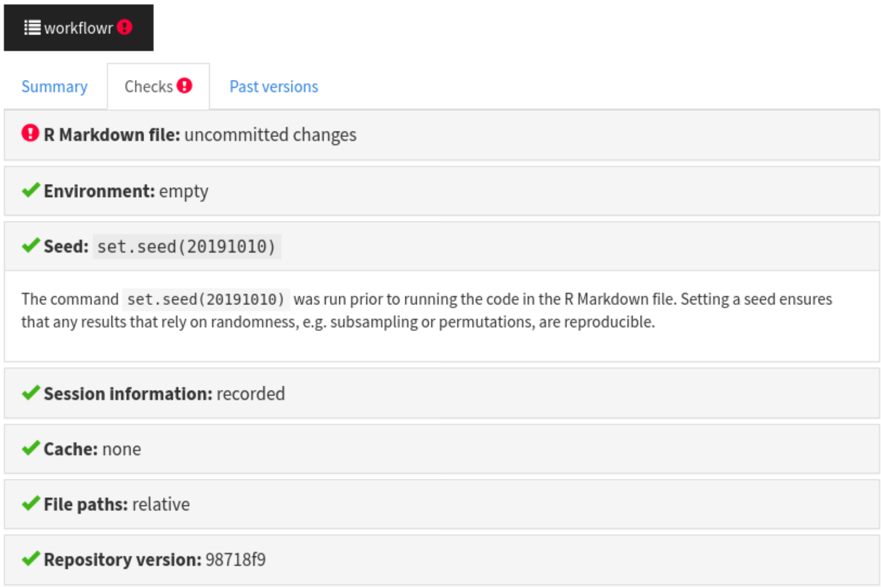
~~~
boxplot(len ~ dose, data = teeth)
boxplot(len ~ supp, data = teeth)
boxplot(len ~ dose + supp, data = teeth)
```
~~~

4. To compare the tooth length of the guinea pigs given orange juice versus those given vitamin C, you could perform a permutation-based statistical test. This would involve comparing the observed difference in teeth length due to the supplement method to the observed differences calculated from random permutations of the data. The basic idea is that if the observed difference is an outlier compared to the differences generated after permuting the supplement method column, it is more likely to be a true signal not due to chance alone. We are not going to perform the full permutation test here, but we will just demonstrate the idea of a permutation. Copy the code chunk below, paste it at the bottom of of the file `teeth.Rmd`, and execute it. Try executing it several times – does it give you a different answer each time?

    ~~~
    ```{r permute}
    # Observed difference in teeth length due to supplement method
    mean(teeth$len[teeth$supp == "OJ"]) - mean(teeth$len[teeth$supp == "VC"])
    # Permute the observations
    supp_perm <- sample(teeth$supp)
    # Calculate mean difference in permuted data
    mean(teeth$len[supp_perm == "OJ"]) - mean(teeth$len[supp_perm == "VC"])
    ```
    ~~~

5. In the R console, run `wflow_build()`. Note the value of the observed difference in the permuted data.

6. In RStudio, click on the Knit button. Has the value of the observed difference in the permuted data changed? It should be identical. This is because workflowr always sets the same seed prior to running the analysis.[3] To better understand this behavior as well as the other reproducibility safeguards and checks that workflowr performs for each analysis, click on the workflowr button at the top and select the "Checks" tab.



! **R Markdown file:** uncommitted changes

✓ **Environment:** empty

✓ **Seed:** `set.seed(20191010)`

The command `set.seed(20191010)` was run prior to running the code in the R Markdown file. Setting a seed ensures that any results that rely on randomness, e.g. subsampling or permutations, are reproducible.

✓ **Session information:** recorded

✓ **Cache:** none

✓ **File paths:** relative

✓ **Repository version:** 98718f9

You can see the value of the seed that was set using `set.seed()` before the code was executed.

[3] Note that everyone in the workshop will have the same result because by default workflowr uses a seed that is the date the project was created as YYYYMMDD. You can change this by editing the file `_workflowr.yml`.

Publish your analysis!

You should also notice that workflowr is still giving you a warning: it says you have “uncommitted changes” in your .Rmd file. The term “commit” is a term from version control: it basically means to save a snapshot of the current version of a file so that you could return to it later if you wanted (even if you changed or deleted the file in between).

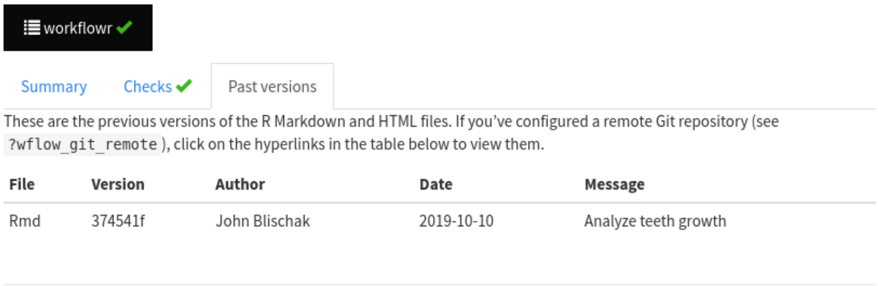
So, workflowr is warning you that you haven’t saved a snapshot of your current analysis. If this analysis is something you are currently (somewhat) happy with then you should save a snapshot that will allow you to go back to it at any time in the future (even if you change the .Rmd file between now and then). In workflowr we use the term “publish” for this process: any analysis that you “publish” will be one that you can go back to in the future. You will see that it is pretty easy to publish an analysis so you should do it whenever you create a first working version, and whenever you make a change that you might want to keep. Don’t wait to think that it is your “final” version before publishing, or you will never publish!

1. Publish your analysis by typing:

   ```
   wflow_publish("analysis/teeth.Rmd", message = "Analyze teeth growth")
   ```

   The function `wflow_publish()` performs three steps: 1) commits (snapshots) the .Rmd files, 2) rebuilds the Rmd files to create the html file and figures, and 3) commits the HTML and figure files. This guarantees that the results in each html file is always generated from an exact, known version of the Rmd file (you can see this version embedded in the workflowr report). An informative message will help you find a particular version later.

2. Open the workflowr report of `teeth.html` by clicking on the button at the top of the page. Navigate to the tab “Past versions”. Note that the record of all past versions will be saved here. Once the project has been added to GitHub (you will do this in the next section), the “Past versions” tab will include hyperlinks for convenient viewing of the past versions of the Rmd and HTML files.



These are the previous versions of the R Markdown and HTML files. If you’ve configured a remote Git repository (see `?wflow_git_remote`), click on the hyperlinks in the table below to view them.

File	Version	Author	Date	Message
Rmd	374541f	John Blischak	2019-10-10	Analyze teeth growth

Checking your status

When you are working on several analyses over a period of time it can be difficult to keep track of which ones need attention, etc. You can use `wflow_status()` to check on all your files.

1. In the R console, run `wflow_status()`. This will show you the status of each of the Rmd files in your workflowr project. You should see that `teeth.rmd` has status “Published” because you just published it. But the other `.Rmd` files have status “Unpublished” because you haven’t published them yet. Also you will notice a comment that the file `data/teeth.csv` is “untracked”. This basically means that the data file has not had a snapshot kept, which is dangerous as our analyses obviously depend on the version of the data we use. . . .

2. In the R console, run the command below to “publish” these other files [4].

[4] The command uses the wildcard character `*` to match all the Rmd files in `analysis/`. If this fails on your computer, try running the more verbose command: `wflow_publish(c("analysis/index.Rmd", "data/teeth.csv"), message = "Analyze teeth growth")`

```
    wflow_publish(c("analysis/*Rmd", "data/teeth.csv"), message = "Publish data and other files")
    ```

3. Navigate to check html files in the `docs` directory, you should find that they all have a green light and no warnings.
4. And run `wflow_status()` again to confirm all is OK. Everything is published!

Share your results

So, now you have a website, with an analysis in it. But it is only on your computer, not the internet. To share your website with the world we will use the free service GitHub Pages.

1. In the R console, run the function `wflow_use_github()`. The only required argument is your GitHub username. The name of the repository will automatically be named the same as the directory containing the workflowr project, in this case "myproject".

    ```
    wflow_use_github("your-github-username")
    ```

    When the function asks if you would like it to create the repository on GitHub for you, enter `1`. This should open your web browser so that you can authenticate with GitHub and then give permission for workflowr to create the repository on your behalf. Additionally, this function connects to your local repository with the remote GitHub repository and inserts a link to the GitHub repository into the navigation bar. If this fails to create a GitHub repository, go to Troubleshooting.

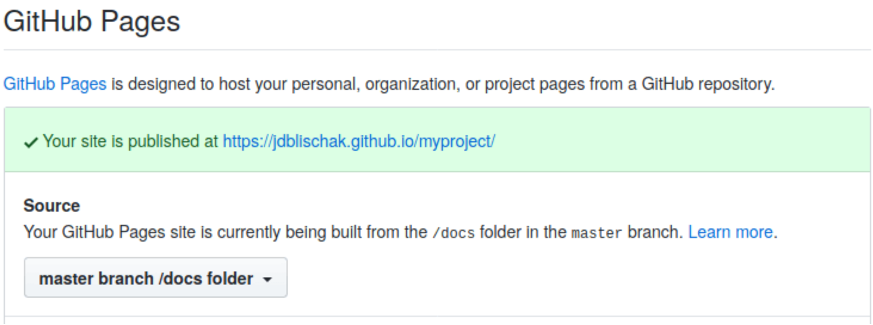
2. To update your workflowr website to use GitHub links to past versions of the files (as well as update the navigation bar to include the GitHub link), republish the files. (You would not have to do this in future)

    ```
    wflow_publish(republish = TRUE)
    ```

3. To send your project to GitHub, run `wflow_git_push()`. This will prompt you for your GitHub username and password. If this fails, go to Troubleshooting.

    ```
    wflow_git_push()
    ```

4. On GitHub, navigate to the Settings tab of your GitHub repository[5]. Scroll down to the section "GitHub Pages". For Source choose "master branch /docs folder". After it updates, scroll back down and click on the URL. If the URL doesn't display your website, go to Troubleshooting.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at https://jdblischak.github.io/myproject/

Source

Your GitHub Pages site is currently being built from the `/docs` folder in the `master` branch. Learn more.

Index your new analysis

Unfortunately your home page is not very inspiring. Also there is not an easy want to find that nice analysis you did! A great way to keep track of analyses and make them easy to find is to keep an index on your

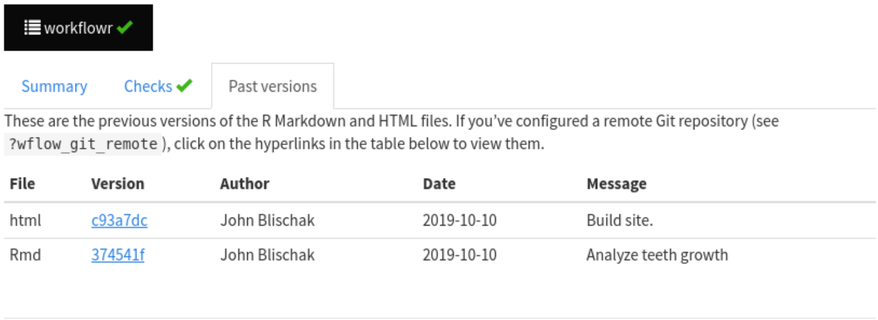
[5] If your GitHub repository wasn't automatically opened by `wflow_git_push()`, you can manually enter the URL into the browser: `https://github.com/username/myproject`.

website homepage. The homepage is created by `analysis/index.Rmd`, so we are now going to edit this file to add a link to our new analysis.

1. Open the file `analysis/index.Rmd`. You can open it from the Files pane or run `wflow_open("analysis/index.Rmd")`.
2. Copy the line below and paste it at the bottom of the file `analysis/index.Rmd`. This text uses "markdown" syntax to create a hyperlink to the tooth analysis. The text between the square brackets is displayed on the webpage, and the text in parentheses is the relative path to the teeth webpage. Note that you don't need to include the subdirectory `docs/` because `index.html` and `teeth.html` are both already in `docs/`. (In an html file relative paths are specified relative to the current page which in this case will be `index.html`.) Also note that you need to use the file extension `.html` since that is the file that needs to be opened by the web browser.

   ```
   * [Teeth growth analysis](teeth.html)
   ```

3. Maybe you would like to write a short introductory message in your index file e.g. "Welcome to my first workflowr website"!
4. You might also want to add a bit more details on what the tooth growth analysis did – a little detail in your index can be really helpful when it starts getting bigger…
5. Run `wflow_build()` and then confirm that clicking on the link "Teeth growth" takes you to your teeth analysis page.
6. Run `wflow_publish()` to publish this new index file.
7. Run `wflow-status()` to check everything is OK.
8. Run `wflow_git_push()` to push the changes to GitHub.
9. Now go to your GitHub page again, and check out your website! (It can take a couple of minutes to refresh after pushing, so you may need to be patient). Navigate to the tooth analysis. Click on the links in the "Past versions" tab to see the past results. Click on the HTML hyperlink to view the past version of the HTML file. Click on the Rmd hyperlink to view the past version of the Rmd file on GitHub. Enjoy!



These are the previous versions of the R Markdown and HTML files. If you've configured a remote Git repository (see `?wflow_git_remote`), click on the hyperlinks in the table below to view them.

File	Version	Author	Date	Message
html	c93a7dc	John Blischak	2019-10-10	Build site.
Rmd	374541f	John Blischak	2019-10-10	Analyze teeth growth

Conclusion

You have successfully created and shared a reproducible research website. The key commands are a pretty short list: `wflow_build()`, `wflow_publish()`, `wflow_status()`, and `wflow_git_push()`. Using the same workflowr commands, you can do the same for one of your own research projects and share it with collaborators and colleagues.

To learn more about workflowr, you can read the following vignettes:

- Customize your research website
- Migrating an existing project to use workflowr
- How the workflowr package works

- Frequently asked questions
- Hosting workflowr websites using GitLab
- Sharing common code across analyses
- Alternative strategies for deploying workflowr websites

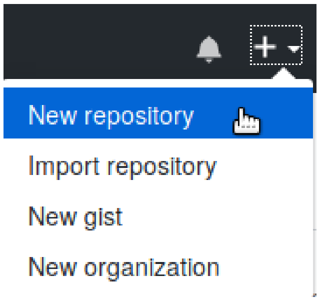
Troubleshooting

I don't see the workflowr project as an available RStudio Project Type.

If you just installed workflowr, close and re-open RStudio. Also, make sure you scroll down to the bottom of the list.

The GitHub repository wasn't created automatically by `wflow_use_github()`.

If `wflow_use_github()` failed unexpectedly when creating the GitHub repository, or if you declined by entering `n`, you can manually created the repository on GitHub. After logging in to GitHub, click on the "+" in the top right of the page. Choose "New repository". For the repository name, type `myproject`. Do not change any of the other settings. Click on the green button "Create repository". Once that is completed, you can return to the next step in the tutorial.



I wasn't able to push to GitHub with `wflow_git_push()`.

Unfortunately this function has a high failure rate because it relies on the correct configuration of various system software dependencies. If this fails, you can push to Git using another technique, but this will require that you have previously installed Git on your computer. For example, you can use the RStudio Git pane (click on the green arrow that says "Push"). Alternatively, you can directly use Git by running `git push` in the terminal.

My website isn't displaying after I activated GitHub Pages.

It is not uncommon for there to be a short delay before your website is available. One trick to try is to specify the exact page that you want at the end of the URL, e.g. add `/index.html` to the end of the URL.

Statistics for a data rich world—some explorations[*]

Stefano Allesina (adapted by Lin Chen) *University of Chicago*

Goal: More and more often we need to analyze large and complex data sets. However, the statistical methods we've been taught in college have evolved in a data-poor world. Modern biology requires new tools, which can cope with the new questions and methods that arise in a data-rich world. Here we are going to discuss problems that often arise in the analysis of large data sets. We're going to review hypothesis testing (and what happens when we have many hypotheses) and discuss model selection. We're going to see the effects of selective reporting and p-hacking, and how they contribute to the *reproducibility crisis* in the sciences. **Audience:** Biologists with some programming background.

I. Review of hypothesis testing

Statistics is the science of collecting and analyzing data from samples in order to estimate and make inference regarding the parameters in the population. A sample is a smaller and random subset of the population of interest and the sample is collected to represent the population. Hypothesis testing is a major component of statistical inference.

The basic idea of hypothesis testing is the following: we have devised an hypothesis on our system, which we call H_1 (the "alternative hypothesis"). We have collected our data, and we would like to test whether the data are consistent (or inconsistent) with the so-called "null-hypothesis" (H_0), which is associated with a contradiction to the hypothesis we would like to prove.

The simplest example is that of a bent coin: my friend Aiden likes to bet on a coin toss with people, and he often chooses head and wins. I suspect that he has a bent coin in favor of head (H_1: the coin is bent). We therefore toss the coin several times and check whether the number of heads we observe is consistent with the null hypothesis of a fair coin (H_0: the coin is a fair coin).

In `R` we can toss many coins in no time at all. Call `p` the probability of obtaining a head, and initially toss a fair coin (`p = 0.5`) a thousand times:

First, when simulating a data, we always want to set seed to make sure the results are reproducible.

```
set.seed(222)
p <- 0.5 # probability of a head (fair coin)
flips <- 1000 # number of times we flip the coin
data <- sample(c("H", "T"),
                 flips, prob = c(p, 1 - p),
                 replace = TRUE)
heads <- sum(data == "H")
```

[*] This document is included as part of the Statistics for large data sets packet for the U Chicago BSD qBio6 boot camp, @MBL, 2020. **Current version**: 2020; **Corresponding author**: sallesina@uchicago.edu. Adapted by Lin Chen (lchen@health.bsd.uchicago.edu).

There is also a faster way to flip the coins and count the heads by assuming that the number of heads out of 1000 flips follows a binomial distribution:

```r
heads <- rbinom(1, flips, p)
```

If the coin is fair, we expect approximately 500 heads, but of course we might have small variations due to the randomness of the process. We therefore need a way to distinguish between "bad luck" and an incorrect hypoythesis.

What is customarily done is to compute the probability of recovering the observed or a more extreme version of the pattern under the null hypothesis: if the probability is very small, it implies that when the null hypothesis is true, there is a very small probability (i.e., very unlikely) that you can observe things as or more extreme than what you have observed in the current data. We call this probability a *p-value*. A small *p-value* (typically less than 0.05) indicates strong evidence against the null hypothesis, so you reject the null hypothesis and conclude that the data is inconsistent with the null hypothesis. A large *p-value* (> 0.05) only means that when the null hypothesis is true, you are likely to observe what has been observed in the current data but that is not enough to prove the null hypothesis is true, so you fail to reject the null hypothesis and the conclusion is inconclusive. When the null hypothesis is indeed true or when the alternative hypothesis is true but you do not have enough power to reject the null, you may end up with a large p-value.

For example, if the coin is fair, the number of heads should follow the binomial distribution. The probability of observing a larger number of heads than what we've got is therefore

```r
one.sided.pvalue <- 1 - pbinom(heads, flips, 0.5)
```

Here and in the following sections, we calculated a one-sided p-value testing $H_0 : p = 0.5$ versus $H_A : p > 0.5$. Note that in many other cases, we recommend a two-sided test, and a two-sided p-value together with a 95% confidence interval can be obtained via:

```r
heads <- rbinom(1, flips, p)
pvalue <- binom.test(heads, flips, 0.5, alternative="two.sided")
```

What if we repeat the tossing many, many times?

```r
# flip 1000 coins 1000 times, and count the number of heads in each of the 1000 experiments
# produce histogram of number of heads
heads_distribution <- rbinom(1000, flips, p)
hist(heads_distribution, main = "distribution number heads", xlab = "number of heads")
```

You can see that it is very unlikely to get more than 560 (or less than 440) heads when flipping a fair coin 1000 times. Therefore, if we were to observe say 400 heads (or 600), we would tend to believe that the coin is biased (though of course this could have happened by chance if we are repeating the tossing 1 trillion times!).

Type I and type II errors

When testing an hypothesis, we can make two types of errors:

- **Type I error**: reject H_0 when it is in fact true. Also known as false positive.
- **Type II error**: fail to reject H_0 when in fact it is not true. Also known as false negative.

We call α the probability of making a type I error (or type I error rate), and β as type II error rate. And power is in fact $1 - \beta$. We can calculate the p-value based on the data and compare the p-value with the significance threshold α. The p-value quantifies how strongly the data contradicts the null hypothesis, and if $p < \alpha$, we reject the null hypothesis. Type I and Type II error rates are inversely related. In choosing a significance level for a test, you are actually deciding how much you want to risk committing a type I error — rejecting the null hypothesis when it is. The more stringent α is (0.01 versus 0.05) in controlling type I error rate, the less likely you would make a rejection decision and consequently the power will be reduced too.

The distribution of p-values

Suppose that we are tossing each of several fair coins 1000 times. For each, we compute the corresponding (one-sided) p-value testing the null hypothesis `p = 0.5` against the alternative `p > 0.5`. How are the p-values distributed?

```r
ncoins <- 2500
heads <- rbinom(ncoins, flips, p)
pvalues <- 1-pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", freq = FALSE)
abline(h = 1, col = "red", lty = 2)
```

As you can see, if the data were generated under the null hypothesis, the distribution of the p-values would be approximately uniform between 0 and 1. This means that if we set $\alpha = 0.05$, we would reject the null hypothesis 5% of the time (even though in this case we know the hypothesis is correct!).

What is the distribution of the p-values if we are tossing biased coins? We will find an enrichment in small p-values, with stronger effects for larger biases:

```r
p <- 0.52 # the coin is biased
heads <- rbinom(ncoins, flips, p)
pvalues <- 1 - pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", main = paste0("p =  ", p), freq = FALSE)
abline(h = 1, col = "red", lty = 2)

p <- 0.55 # the coin is biased
heads <- rbinom(ncoins, flips, p)
pvalues <- 1 - pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", main = paste0("p =  ", p), freq = FALSE)
abline(h = 1, col = "red", lty = 2)
```

II. The challenges with p-values

Selective reporting

Articles reporting positive results are easier to publish than those containing negative results. Authors might have little incentive to publish negative results, which could go directly into the file-drawer.

This tendency is evidenced in the distribution of p-values in the literature: in many disciplines, one finds a sharp decrease in the number of tests with p-values just below 0.05 (which is customarily–and arbitrarily–chosen as a threshold for "significant results"). For example, we find many a sharp decrease in the number of reported p-values of 0.051 compared to 0.049–while we expect the p-value distribution to decrease smoothly.

Selective reporting leads to irreproducible results: we always have a (small) probability of finding a "positive" result by chance alone. For example, suppose we toss a fair coin many times, until we find a "signficant" result.

On the other hand, more and more journals would require us to report effect size estimates and confidence intervals – "Were this procedure to be repeated on numerous samples, the fraction of calculated confidence intervals (which would differ for each sample) that encompass the true population parameter would tend toward 90%."(source: Cox D.R., Hinkley D.V. (1974) Theoretical Statistics, Chapman & Hall, p49, p209.)

Problem: p-hacking

The problem is well-described by Simonsohn et al. (J. Experimental Psychology, 2014): "While collecting and analyzing data, researchers have many decisions to make, including whether to collect more data, which method to use, which measure(s) to analyze, which covariates to use, what to do with outliers and missing data, and so on. If these decisions are not made in advance but rather are made as the data are being analyzed, then researchers may make them in ways that self-servingly increase their odds of publishing. Thus, rather than placing entire studies in the file-drawer, researchers may file merely the subsets of analyses that produce nonsignificant results. We refer to such behavior as *p-hacking*." The term p-hacking describes the conscious or subconscious manipulation of data in a way that produces a desired p-value.

The same authors showed that with careful p-hacking, almost anything can become significant (read their hylarious article in Psychological Science, where they show that listening to a song can change the listeners' age!).

Discussion on p-values

Selective reporting and p-hacking are only two of the problems associated with the widespread use and misuse of p-values. The discussion in the scientific community on this issue is extremely topical. I have collected some of the articles on this problem in the `readings` folder. Importantly, in 2016 the American Statistical Association released a statement on p-values every scientist should read.

Reproducibility crisis

P-values and hypothesis testing contribute considerably to the so-called *reproducibility crisis* in the sciences. A survey promoted by *Nature* magazine found that "More than 70% of researchers have tried and failed to reproduce another scientist's experiments, and more than half have failed to reproduce their own experiments."

This problem is due to a number of factors, and addressing it will likely be one of the main goals of science in the next decade.

Exercise: p-hacking

Go to goo.gl/a3UOEF and try your hand at p-hacking, showing that your favorite party is good (bad) for the economy.

III. Multiple comparisons (also known as multiple testing)

The problem of multiple comparisons arises when we perform multiple statistical tests. Since each test is subject to some small chance of producing a false positive result, when jointly considering many many tests, the chances of producing some false positive findings are much higher.

Suppose we perform our coin tossing exercise, flipping 1000 coins 1000 times each. For each coin, we determine whether our data differs significantly from what expected by contrasting our p-value with a significance level $\alpha = 0.05$.

Even if the coins are all perfectly fair, we would expect to find approximately $0.05 \cdot 1000 = 50$ coins that lead to the rejection of the null hypothesis.

In fact, we can calculate the probability of making at least one type I error (reject the null when in fact it is true). This probability is called the Family-Wise Error Rate (FWER). It can be computed as 1 minus the probability of making no type I error at all. If we set $\alpha = 0.05$, and assume the tests to be independent, the probability of making no errors in m tests is $1 - (1 - 0.05)^m$. Therefore, if we perform 10 tests, we have about 40% probability of making at least a mistake; if we perform 100 tests, the probability grows to more than 99%. If the tests are not independent, we can still say that in general $FWER \leq m\alpha$.

This means that setting an α per test does not control for FWER.

Moving from tossing coins to biology, consider the following examples:

- **Gene expression** In a typical RNAseq experiment, we compare the differential expression levels of tens of thousands of genes in the treatment and control tissues.
- **GWAS** In Genome-Wide Association Studies we want to find single-nucleotide polymorphisms (SNPs) associated with a given phenotype. It is common to test millions of SNPs for signficant associations.
- **Identifying binding sites** Identifying candidate binding sites for a transcriptional regulator requires scanning the whole genome, yielding tens of millions of tests.

Organizing the tests in a table

Suppose that we're testing m hypotheses. Of these, an unknown subset m_0 is true, while the remaining $m_1 = m - m_0$ are false. We would like to correctly call the true/false hypotheses (as much as possible). We can summarize the results of our tests in a table, of which the elements are unobservable:

What we would like to know is $m_1 = T + S$ and $m_0 = U + V$. Then V is the number of type I errors (rejected H when in fact it is true), and T is the number of type II errors (failed to reject a false H). However, we can only observe $V + S$ (the number of "discoveries"), and $U + T$ (number of "failures").

The type I error rate is $E[V]/m$ (where $E[X]$ stands for expectation). When there are many tests (m) being considered, controlling for type I error rates at 0.05 means that by random chance, one could make $0.05 \times m$ false positive findings even if all m tests are under the null. For example, when testing genetic associations between 10 million genetic variants to the risk of a disease, if still using 0.05 as the significance threshold, there could be 500k significant findings by random chance even if the disease is not heritable and has no genetic association. Apparently, we need to choose a much more strigent significance threshold to account for the number of tests, and there are some other error measures. The Family-wise error rate is defined as $P(V > 0)$. Another quantity of interest is the False Discovery Rate (FDR), measured as the proportion of true discoveries $FDR = E[V/(V+S)]$ when $V + S > 0$. FDR measures the proportion of falsely rejected hypotheses.

Importantly, FWER guards against any single false positive finding among all m tests, and is a more stringent significance criteria than FDR in multiple comparison problems. It also means that when we control for FWER, we're automatically controlling for the FDR, but not vice versa. It should be noted that when a large number of tests is performed, controlling FWER could be quite conservative and may lose power.

Methods for multiple testing correction: Bonferroni correction

One of the simplest and most widely-used procedures to control for FWER is Bonferroni's correction. This procedure controls for FWER in the case of independent or dependent tests. It is typically quite conservative, especially when the tests are not independent (in practice, it becomes "too conservative" when the number of tests is moderate to high). Fundamentally, for a desired FWER α we choose as a significance treshold of α/m for each single test, where m is the number of tests we're performing. Equivalently, we can "adjust" the p-values as $q_i = \min(m \cdot p_i, 1)$, and call significant the values $q_i < \alpha$. In R it is easy to perform this correction:

```r
original_pvals <- c(0.012, 0.06,  0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "bonferroni")
print(adjusted_pvals)
```

With these adjusted p-values, and an $\alpha = 0.05$, we would still single out as significant the fourth test, but not the first. The strength of Bonferroni is its simplicity, and the fact that we can perform the operation in a single step. Moreover, the order of the tests does not matter.

Other procedures: the Holm–Bonferroni method (or the Holm method)

There are several refinements of Bonferroni's correction, some of which use the sequence of ordered p-values. For example, the Holm method starts by sorting the p-values in increasing order $p_{(1)} \leq p_{(2)} \leq p_{(3)} \leq \dots p_{(m)}$. The hypothesis $H_{(i)}$ is rejected if $p_{(j)} \leq \alpha/(m-j+1)$ for all $j = 1, \dots, i$. Equivalently, we can adjust the p-values as $q_{(i)} = \min(1, \max((m-i+1)p_{(i)}, q_{(i-1)}))$. In this way, we use the most stringent threshold to determine whether the smallest p-value is significant, the next smallest p-value uses a slightly higher threshold and so on. The Holm method is uniformly more powerful than the Bonferroni correction.

For example, using the same p-values above:

```r
original_pvals <- c(0.012, 0.06,  0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "holm")
print(adjusted_pvals)
```

We see that we would be calling the first test significant, contrary to what obtained with Bonferroni.

The function `p.adjust` offers several choices for p-value correction. Also, the package `multcomp` provides a quite comprehensive set of functions for multiple hypothesis testing.

An example: testing mixed coins

We're going to test these concepts by tossing repeatedly many coins. In particular, we're going to toss 1000 times 50 biased coins ($p = 0.55$) and 950 fair coins ($p = 0.5$). For each coin, we're going to compute a p-value, and count the number of type I, type II, etc. errors when using unadjusted p-values as well as when correcting using the Bonferroni or Holm procedure.

```r
toss_coins <- function(p, flips){
  # toss a coin with probability p of landing on head several times
  # return a data frame with p, number of heads, pval and
  # H0 = TRUE if p = 0.5 and FALSE otherwise
  heads <- rbinom(1, flips, p)
  pvalue <- 1 - pbinom(heads, flips, 0.5)
  if (p == 0.5){
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = TRUE))
  } else {
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = FALSE))
  }
}

# To ensure everybody gets the same results, we're setting the seed
set.seed(8)
data <- data.frame()
# the biased coins
for (i in 1:50) data <- rbind(data, toss_coins(0.55, 1000))
# the fair coins
```

```
for (i in 1:950) data <- rbind(data, toss_coins(0.5, 1000))
# here's the data structure
head(data)
```

Now we write a function that adjusts the p-values and builds the table above

```
get_table <- function(data, adjust, alpha = 0.05){
  # produce a table counting U, V, T and S
  # after adjusting p-values for multiple comparisons
  data$pval.adj <- p.adjust(data$pval, method = adjust)
  data$reject <- FALSE
  data$reject[data$pval.adj < alpha] <- TRUE
  return(table(data[,c("reject","H0")]))
}
```

First, let's see what happens if we don't adjust the p-values:

```
no_adjustment <- get_table(data, adjust = "none", 0.05)
print(no_adjustment)
```

We correctly declared 48 of the biased coins "significant", but we also incorrectly called 2 biased coins "not significant" (Type II error). More worringly, we called 45 fair coins biased when they were not (Type I error). To control for the family-wise error rate, we can correct using Bonferroni:

```
bonferroni <- get_table(data, adjust = "bonferroni", 0.05)
print(bonferroni)
```

With this correction, we dramatically reduced the number of type I errors (from 45 to 0), but at the cost of increasing type II errors (from 2 to 40) and losing power. In this way, we would make only 10 discoveries instead of 50.

In this case, Holm's procedure does not help:

```
holm <- get_table(data, adjust = "holm", 0.05)
print(holm)
```

More sophisticated methods, for example the Benjamini-Hochberg (BH) procedure based on controlling false discovery rate (FDR=E(false discoveries/signifiant tests), where E stands for expectation), can reduce the type II errors and improve power, at the cost of a few and estimable type I errors:

```
BH <- get_table(data, adjust = "BH", 0.05)
print(BH)
```

FDR and q-values

Inspired by the need for controlling for FDR in genomics, Storey and Tibshirani (PNAS 2003) have proposed the idea of a q-value, measuring the probability that a feature that we deemed significant turns out to be not significant after all.

One uses p-values to control for the false positive rate (# false positive / total test): when determining significant p-values we control for the rate at which null features in the data are called significant. The False Discovery Rate (# false positive / # signficant test), on the other hand, measures the rate at which results that are deemed significant are truly null. While setting $PCER = 0.05$ we are stating that about 5% of the truly null features will be called significant, an $FDR = 0.05$ means that among the features that are called significant, about 5% will turn out to be null.

They proposed a method that uses the ensamble of p-values to determine the approximate (local) FDR. The idea is simple. If you plot your histogram of p-values when you have few true effect, and many nulls, you will see something like:

```r
hist(data$pval, breaks = 25)
```

where the right side of the histogram is close to a uniform distribution. We could use the high p-values to find how tall the histogram would be if all effects were null, thereby estimating the proportion of truly null features $\pi_0 = m_0/m$.

Storey has built an `R`-package for this type of analysis:

```r
# To install:
#install.packages("devtools")
#library("devtools")
#install_github("jdstorey/qvalue")
library("qvalue")
qobj <- qvalue(p = data$pval)
```

Here's the estimation of the π_0

```r
hist(qobj)
```

which is quite good (in this case we know that $\pi_0 = 0.95$). The small p-values under the dashed line represent our false discoveries. Even better, through randomizations one can associate a q-value to each test, representing the probability of making a mistake when calling a result significant (formally, the q-value is the minimum FDR that can be attained when calling that test significant).

For example:

```r
table((qobj$pvalues < 0.05) & (qobj$qvalues < 0.05), data$H0)
```

Note that the estimation of FDR is unstable if the demoniator (# significant test) is expected to be small. Therefore, you may notice that FDR was widely used in detecting differentially expressed genes in diseased versus normal samples where the expected number of non-null tests is large. In contast, in GWAS, researchers use the Bonferroni-adjusted p-value threshold of 5×10^{-8} to declare significance.

IV. Linear regression, logistic regression, and model selection

Linear regression

In statistics, linear regression is an approach to model the linear relationship between one response variable (or dependent variable) and one or more explanatory variables (or independent variables, or predictor). If there is only one exlanatory variable, it is called simple linear regression. If the linear regression involves more than one explanatory variable, it is a multiple linear regression.

```r
# create fake data for a simple linear regression
set.seed(5)
x <- 1:20
y <- 3 + 0.5 * x + rnorm(20)
plot(y ~ x)
```

We can fit a simple linear regression to the data

```r
model1 <- lm(y ~ x)
summary(model1)
plot(y~x)
points(model1$fitted.values~x, type = "l", col = "blue")
```

In a data rich world, often, we need to select a model out of a set of reasonable alternatives with different combinations and/or (even nonlinear) patterns of explanatory variables. However, we run the risk of overfitting the data (i.e., fitting the noise as well as the pattern). The best fitted model for the current data from the current sample may not be the best model representing the pattern in the population of interest. Here is a simple example of a overfitted regression:

For the above data, we can also fit a more complex polynomial function of x.

```r
model2 <- lm(y ~ poly(x, 7))
```

Let's see the residuals etc.

```r
summary(model1)
summary(model2)
plot(y~x)
points(model1$fitted.values~x, type = "l", col = "blue")
points(model2$fitted.values~x, type = "l", col = "red")
```

Our second model has a much greater R^2, i.e., more variation in the response variable can be explained by the model, but the second model also has many more parameters. The first model is more parsimonious. Which is a model we should choose?

Model selection tries to address this and similar problems. Most model fitting and model selection procedures are based on likelihoods (e.g., Bayesian models, maximum likelihood, minimum description length). The likelihood $L(\theta|D, M)$ is (proportional to) the probability of observing the data D under the model M and parameters θ. Because likelihood can be very small when you have much data, typically one works with log-likelihoods. For example:

```r
logLik(model1)
logLik(model2)
```

Typically, more complex models will yield better (less negative) log-likelihoods and a better fit for the current data. However, more parameters would also increase the variation of the model. A complex model may not best represent the population (not to say computational burden). We will need to find a balance between bias and variance. We therefore want to penalize more complex models in some way.

AIC

One of the simplest methods to select among competing models is the Akaike Information Criterion (AIC). It penalizes models accoding to the **number of parameters**: $AIC = 2p - 2\log L(\theta|D, M)$, where p is the number of parameters. Note that **smaller** values of AIC stand for "better" models. In `R` you can compute AIC using:

```r
AIC(model1)
AIC(model2)
```

As you can see, AIC would favor the first (and simpler) model, which is also the model we used to simulate the data.

AIC is rooted in information theory and measures (asymptotically) the loss of information when using the model instead of the data. There are several limitations of AIC: a) it only holds asymptotically (i.e., for very large data sets; for smaller data you need to correct it); it penalizes each parameter equally (i.e., parameters that have a large influence on the likelihood have the same weight as parameters that do no influence the likelihood much); it can lead to overfitting, favoring more complex models in simulated data generated by simpler models.

BIC

In a similar vein, BIC (Bayesian Information Criterion) uses a slightly different penalization: $BIC = \log(n)p - 2\log L(\theta|D, M)$, where n is the number of data points. You may see that for large data, BIC penalizes a complex model more than AIC. Again, smaller values stand for "better" models:

```r
BIC(model1)
BIC(model2)
```

Here in this simple example, AIC and BIC agree with each other, and the simpler model is favored.

Logistic regression

Logistic regression is often used to model the nonlinear relationship (modeled by a logistic function) between a binary response variable and a linear combination of explantory variables. When the outcome is binary, for example pass/fail, win/lose, alive/dead, yes/no, one would consider a logistic regression. It can be extended to model several classes of a categorical variable as response.

We will start with an example. Fox et al. (Research Integrity and Peer Review, 2017) analyzed the invitations to review for several scientific journals, and found that “The proportion of invitations that lead to a submitted review has been decreasing steadily over 13 years (2003–2015) for four of the six journals examined, with a cumulative effect that has been quite substantial”. Their data is stored in `../data/FoxEtAl.csv`. We’re going to build models trying to predict whether a reviewer will agree (or not) to review a manuscript.

```r
# read the data
reviews <- read.csv("../data/FoxEtAl.csv", sep = "\t")
# take a peek
head(reviews)
# set NAs to 0
reviews[is.na(reviews)] <- 0
# how big is the data?
dim(reviews)
# that's a lot! Let's take 5000 review invitations for our explorations;
# we will fit the whole data set later
set.seed(101)
small <- reviews[order(runif(nrow(reviews))),][1:5000,]
```

The response variable of interest is a reviewer i agreeing to review a manuscript or not, and is binary; and so we decided to use a logistic regression. Call π_i the probability that a reviewer i agree to review a manuscript. We model $logit(\pi_i) = \log(\pi_i/(1-\pi_i))$ as a linear function.

Constant rate

As a null model we build a model in which the probability of agreeomg to review does not change in time/for journals:

```r
# suppose the rate at which reviewers agree is a constant
mean(small$ReviewerAgreed)
# fit a logistic regression
model_null <- glm(ReviewerAgreed~1, data = small, family = "binomial")
summary(model_null)
# interpretation:
exp(model_null$coefficients[1]) / (1 + exp(model_null$coefficients[1]))
```

Declining trend

We now build a model in which the probability to review declines steadily from year to year:

```r
# Take 2003 as baseline
model_year <- glm(ReviewerAgreed~I(Year - 2003), data = small, family = "binomial")
#The I() function acts to convert the argument to "as.is"
summary(model_year)
```

Journal dependence

Reviewers might be more likely to agree for more prestigious journals:

```r
# Take the first journal as baseline
model_journal <- glm(ReviewerAgreed~Journal, data = small, family = "binomial")
summary(model_journal)
```

Model journal and year

Finally, we can build a model combining both features: we fit a parameter for each journal/year combination

```r
# Take the first journal as baseline, the colon mark stands for interaction term only.
model_journal_yr <- glm(ReviewerAgreed~Journal:I(Year-2003),
                        data = small, family = "binomial")
#summary(model_journal_yr)
```

Likelihoods

In `R`, you can extract the log-likelihood from a model object calling the function `logLik`

```r
logLik(model_null)
logLik(model_year)
logLik(model_journal)
logLik(model_journal_yr)
```

Interpretation: because we're dealing with binary data, the likelihood is the probability of correctly predicting the agree/not agree for all the 5000 invitations considered. Therefore, the probability of guessing a (random) invitation correctly under the first model is:

```r
exp(as.numeric(logLik(model_null)) / 5000)
```

while the most complex model yields

```r
exp(as.numeric(logLik(model_journal_yr))/ 5000)
```

We didn't improve our guessing much by considering many parameters! This could be due to specific data points that are hard to predict, or mean that our explanatory variables are not sufficient to model our response variable.

AIC

We can also calculate AIC for logistic models. In `R` you can compute AIC using:

```r
AIC(model_null)
AIC(model_year)
AIC(model_journal)
AIC(model_journal_yr)
```

As you can see, the model `model_journal_yr` has the smallest AIC among all and is preferred here.

BIC

In a similar vein, BIC (Bayesian Information Criterion) uses a slightly different penalization: $BIC = \log(n)p - 2\log L(\theta|D,M)$, where n is the number of data points. Again, smaller values stand for "better" models:

```r
BIC(model_null)
BIC(model_year)
BIC(model_journal)
BIC(model_journal_yr)
```

Note that according to BIC, `model_year` is favored. As mentioned before, BIC would penalize a complex model more.

Cross validation

One very robust method to perform model selection, often used in machine learning, is cross-validation. The idea is simple: split the data in three parts: a small data set for exploring; a large set for fitting; a small set for testing (for example, 5%, 75%, 20%). You can use the first data set to explore freely and get inspired for a good model. The data will be then discarded. You use the largest data set for accurately fitting your model(s). Finally, you validate your model or select over competing models using the last data set.

Because you haven't used the test data for fitting, this should dramatically reduce the risk of overfitting. The downside of this is that we're wasting precious data. There are less expensive methods for cross validation, but if you have much data, or data are cheap, then cross-validation has the virtue of being fairly robust.

Let's try our hand at cross-validation. First, we split the data into three parts:

```r
reviews$cv <- sample(1:3, nrow(reviews), prob = c(0.05, 0.75, 0.2), replace = TRUE)
dataexplore <- reviews[reviews$cv == 1,]
datafit <- reviews[reviews$cv == 2,]
datatest <- reviews[reviews$cv == 3,]
# We've already done our exploration.
# Let's fit the data using model_journal
# and model_journal_yr, which seem to be the most promising
cv_model1 <- glm(ReviewerAgreed~I(Year-2003), data = datafit, family = "binomial")
cv_model2 <- glm(ReviewerAgreed~Journal:I(Year-2003), data = datafit,
      family = "binomial")
```

Now that we've fitted the models, we can use the function predict to find the fitted values for the testdata:

```
mymodel <- cv_model1
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

repeat for the other model

```
mymodel <- cv_model2
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

Cross validation supports the choice of the more complex model here.

Other approaches

Bayesian models are gaining much traction in biology. The advantage of these models is that you can get a posterior distribution for the parameter values, reducing the need for p-values and AIC. The downside is that fitting these models is computationally much more expensive (you have to find a distribution of values instead of a single value).

There are three main ways to perform model selection in Bayesian models:

- **Reversible-jump MCMC** You build a Monte Carlo Markov Chain that is allowed to "jump" between models. You can choose a prior for the probability of being in each of the models; the posterior distribution gives you an estimate of how much the data supports each model. Upside: direct measure. Downside: difficult to implement in practice – you need to avoid being "trapped" in a model.
- **Bayes Factors** Ratio between the probability of two competing models. Can be computed analytically for simple models. Can also be interpreted as the average likelihood when parameters are chosen according to their prior (or posterior) distribution. Upside: straightforward interpretation — it follows from Bayes theorem; Downside: in most cases, one needs to approximate it; can be tricky to compute for complex models.
- **DIC** Similar to AIC and BIC, but using distributions instead of point estimates.

Another alternative paradigm for model selection is Minimum-Description Length. The spirit is that a model is a way to "compress" the data. Then you want to choose the model whose total description length (compressed data + description of the model) is minimized.

A word of caution

The “best” model you’ve selected could still be a terrible model (best among bad ones). Out-of-fit prediction (such as in the cross-validation above) can give you a sense of how well you’re modeling the data.

When in doubt, remember the (in)famous paper in Nature by Tatem et al. 2004, which used some flavor of model selection to claim that, according to their linear regression, in the 2156 Olympics the fastest woman would run faster than the fastest man. One of the many memorable letters that ensued reads:

> Sir — A. J. Tatem and colleagues calculate that women may out-sprint men by the middle of the twenty-second century (Nature 431,525; 2004). They omit to mention, however, that (according to their analysis) a far more interesting race should occur in about 2636, when times of less than zero seconds will be recorded.
>
> In the intervening 600 years, the authors may wish to address the obvious challenges raised for both time-keeping and the teaching of basic statistics.
>
> Kenneth Rice

Prediction for population outside the current data is prophecy.

V. Programming Challenge

P-hacking COVID-19

To show firsthand how p-hacking and overfitting are possible, we want you to show how these practices can lead to completely inane results.

You can download today’s data on the geographic distribution of COVID-19 cases worldwide from the source below. The data is updated daily and contains the lastest publicly available data on COVID-19 by country.The data report the “cumulative number for 14 days of COVID-19 cases per 100k individuals” for each country or territory. There are also several other covariates in the data set.

The challenge is to build an analysis pipeline that produces a “significant” p-value for a relationship between COVID-19 cases and another variable, where the relationship can’t possibly be causal. Prepare an Rmarkdown document with the results. At the end of the document write a paragraph to explain your “findings”. Argue for an underlying non-statistical explanation for your group’s fake result, and/or critique your statistcal approach and why your group got an apparently significant p-value.

Hint: At certain dates in the pandemic, COVID-19 cases counts had a statistically significant (but not a real causal) relationship with the alphabetic position of the first letter of the country’s name. You might also consider other meaningless aspects of country name (e.g., vowels versus consonants, letters before M versus after).

Useful sample code:

```r
# read data
library(utils)

#read the Dataset sheet into “R”. The dataset will be called "data".
data <- read.csv("https://opendata.ecdc.europa.eu/covid19/casedistribution/csv",
                 na.strings = "", fileEncoding = "UTF-8-BOM")

# the code below shows how you may recode the country name's
#first letter to a numeric alphabetic position.
data1 <- data[(data$dateRep=="11/08/2020"), ]
CountryAb <- as.integer(as.factor(substr(data1$countriesAndTerritories,1,1)))

# build a linear model for the relationship between Cumulative_number_for_14_days_of_COVID.19_ca
model1<- lm(Cumulative_number_for_14_days_of_COVID.19_cases_per_100000~CountryAb, data=data1)
summary(model1)
# On 8/11/2020, we did not find a significant relationship
# between alphabetic order of country name and COVID-19 cases.
#But what if we keep looking at other dates?
```

Workshops

RNAseq Workshop → Mengjie Chen
Immunology Workshop → Aly Khan
Population Genetics Workshop → John Novembre

RNA-seq data analysis workshop

Instructor: **Mengjie Chen**
Course Assistant: **Kate Farris**

Welcome

This tutorial is going to walk you through basic RNA-seq data analysis using R. We will touch upon topics including quality control, differential expression analysis and downstream functional analysis.

Introduction

Transcriptomics is the study of the complete set of transcripts within a cell, which aims to document all species of transcripts, specifically mRNAs (along with ncRNAs), and quantify the gene expression levels during a particular biological process, in a development stage or under a set of unique pathological conditions like cancer. RNA-seq is a sequencing technique which uses next-generation sequencing (NGS) to reveal the presence and quantity of RNA in a biological sample at a given moment, analyzing the continuously changing cellular transcriptome. With maturation of NGS technologies, RNA-seq has become the main assay for transcriptomics studies.

The basic steps of RNA-seq data analysis are: assessing read quality, pre-processing (trimming), alignment to reference genome or assembly, quantification (of expression levels), and downstream functional analysis. In this workshop, we will skip some pre-processing steps and assume reads haven been aligned to the reference genome.

Step 1: Counting reads in genes

We will examine 8 samples from the `airway` package, which are from the paper by Himes et al: "RNA-seq Transcriptome Profiling Identifies CRISPLD2 as a Glucocorticoid Responsive Gene that Modulates Cytokine Function in Airway Smooth Muscle Cells".

To install these packages you can use the code (or if you are compiling the document, remove the `eval=FALSE` from the chunk.)

```r
install.packages("devtools")
install.packages("pheatmap")
BiocManager::install(c("airway","Rsamtools","Rsubread","DESeq2","vsn",
                       "org.Hs.eg.db","GenomicFeatures", "clusterProfiler"))
```

This workshop will focus on a summarized version of an RNA-seq experiment: a count matrix, which has genes along the rows and samples along the columns. The values in the matrix are the number of reads which could be uniquely aligned to the exons of a given gene for a given sample. We will demonstrate how to build a count matrix for a subset of reads from an experiment, and then use a pre-made count matrix, to avoid having students download the multi-gigabyte BAM files containing the aligned reads. A new pipeline for building count matrices, which skips the alignment step, is to use fast pseudoaligners such as `Sailfish`, `Salmon` and `kallisto`, followed by the `tximport` package. See the package vignette for more details. Here, we will continue with the read counting pipeline.

First, make variables for the different BAM files and GTF file. Use the `sample.table` to contruct the BAM file vector, so that the count matrix will be in the same order as the `sample.table`.

```r
library(airway)
dir <- system.file("extdata", package="airway", mustWork=TRUE)
csv.file <- file.path(dir, "sample_table.csv")
```

```r
sample.table <- read.csv(csv.file, row.names=1)
bam.files <- file.path(dir, paste0(sample.table$Run, "_subset.bam"))
gtf.file <- file.path(dir, "Homo_sapiens.GRCh37.75_subset.gtf")
```

Next we create an `Rsamtools` variable which wraps our BAM files, and create a transcript database from the GTF file. We can ignore the warning about `matchCircularity`. Finally, we make a `GRangesList` which contains the exons for each gene.

```r
library(Rsamtools)
bam.list <- BamFileList(bam.files)
library(GenomicFeatures)
txdb <- makeTxDbFromGFF(gtf.file, format="gtf")
exons.by.gene <- exonsBy(txdb, by="gene")
```

The following code chunk creates a `SummarizedExperiment` containing the counts for the reads in each BAM file (columns) for each gene in `exons.by.gene` (the rows). We add the `sample.table` as column data. Remember, we know the order is correct, because the `bam.list` was constructed from a column of `sample.table`.

```r
library(GenomicAlignments)
se <- summarizeOverlaps(exons.by.gene, bam.list,
                        mode="Union",
                        singleEnd=FALSE,
                        ignore.strand=TRUE,
                        fragments=TRUE)
colData(se) <- DataFrame(sample.table)
```

Exercise: Can you check what is in the object `exons.by.gene`? Can you check what is in the object `se`?

Step 2: Visualizing sample-sample distances

We now load the full `SummarizedExperiment` object, counting reads over all the genes.

```r
library(airway)
data(airway)
airway
colData(airway)
rowRanges(airway)
```

The counts matrix is stored in assay of a `SummarizedExperiment`.

```r
head(assay(airway))
```

Note that, on the un-transformed scale, the high count genes have high variance. That is, in the following scatter plot, the points start out in a tight cone and then fan out toward the top right. This is a general property of counts generated from sampling processes, that the variance typically increases with the expected value. We will explore different scaling and transformations options below.

Exercise: Can you plot the first two columns of `assay(airway)`? What do you observe? Can you plot samples from two different cell lines? Can you plot samples from two different treatment groups? What do you observe?

```r
plot(assay(airway)[,1:2], cex=.1)
```

Step 3: Creating a DESeqDataSet object

We will use the DESeq2 package to normalize the sample for sequencing depth. The DESeqDataSet object is just an extension of the SummarizedExperiment object, with a few changes. The matrix in assay is now accessed with counts and the elements of this matrix are required to be non-negative integers (0,1,2,. . .).

We need to specify an experimental design here, for later use in differential analysis. The design starts with the tilde symbol ~, which means, model the counts (log2 scale) using the following formula. Following the tilde, the variables are columns of the colData, and the + indicates that for differential expression analysis we want to compare levels of dex while controlling for the cell differences.

```
library(DESeq2)
dds <- DESeqDataSet(airway, design= ~ cell + dex)
head(dds)
```

Step 4: Normalization for sequencing depth

The goal of normalization: to make RNA-seq samples comparable by taking into account differences in sequencing depths of RNA-seq libraries. Typical normalization methods include quantile normalization, median of ratios and trimmed mean of M values. Normalization method in DEseq2 is median of ratios, which counts for sequencing depth and RNA composition.

Size factors in DEseq2 are calculated by the median ratio of samples to a pseudo-sample (the geometric mean of all samples). In other words, for each sample, we take the exponent of the median of the log ratios in this histogram.

```
loggeomeans <- rowMeans(log(counts(dds)))
hist(log(counts(dds)[,1]) - loggeomeans,
     col="grey", main="", xlab="", breaks=40)
```

The size factor for the first sample:

```
exp(median((log(counts(dds)[,1]) - loggeomeans)[is.finite(loggeomeans)]))
```

```
sizeFactors(dds)[1]
```

In DEseq2, a size factor will be estimated for each sample.

```
dds <- estimateSizeFactors(dds)
sizeFactors(dds)
```

```
colSums(counts(dds))
```

```
plot(sizeFactors(dds), colSums(counts(dds)))
abline(lm(colSums(counts(dds)) ~ sizeFactors(dds) + 0))
```

Exercise: Is your calculated exponent of the median of the log ratios the same as output of function sizeFactors? Can you repeat the analysis for sample 5? Based on the above plot, what is the relationship between sizeFactors(dds) and column sum of count matrix? Are they equal? Why?

Make a matrix of log normalized counts (plus a pseudocount):

```
log.norm.counts <- log2(counts(dds, normalized=TRUE) + 1)
```

Another way to make this matrix, and keep the sample and gene information is to use the function normTransform. The same matrix as above is stored in assay(log.norm).

```
log.norm <- normTransform(dds)
```

Examine the log counts and the log normalized counts (plus a pseudocount).

```
rs <- rowSums(counts(dds))
par(mfrow=c(1,2))
boxplot(log2(counts(dds)[rs > 0,] + 1)) # not normalized
boxplot(log.norm.counts[rs > 0,]) # normalized
```

Exercise: How do you think of the normalization results? Are you satisfied? What happened after normalization? Make a scatterplot of log normalized counts against each other. Did you note the fanning out of the points in the lower left corner, for points less than 2ˆ5=32? Would you concern about it? Can you repeat for Sample 3 vs. Sample 5?

Make a scatterplot of log normalized counts against each other.

```
plot(log.norm.counts[,1:2], cex=.1)
```

Step 5: Stabilizing count variance

We will use a sophisticated transformation to address the variance for low counts. It uses the variance model for count data to shrink together the log-transformed counts for genes with very low counts. For genes with medium and high counts, the `rlog` is very close to `log2`.

We use the argument `blind=FALSE` which means that the global dispersion trend should be estimated by considering the experimental design, but the design is not used for applying the transformation itself. See the `DESeq2` vignette for more details.

```
rld <- rlog(dds, blind=FALSE)
plot(assay(rld)[,1:2], cex=.1)
```

Another transformation for stabilizing variance in the DESeq2 package is `varianceStabilizingTransformation`. These two tranformations are similar, the rlog might perform a bit better when the size factors vary widely, and the `varianceStabilizingTransformation` is much faster when there are many samples.

```
vsd <- varianceStabilizingTransformation(dds, blind=FALSE)
plot(assay(vsd)[,1:2], cex=.1)
```

We can examine the standard deviation of rows over the mean for the log plus pseudocount and the rlog. Note that the genes with high variance for the log come from the genes with lowest mean. If these genes were included in a distance calculation, the high variance at the low count range might overwhelm the signal at the higher count range.

```
library(vsn)
meanSdPlot(log.norm.counts, ranks=FALSE)
```

Exercise: Can you compare the scatter plot of sample 1 and 2 before and after variance stablization transformation? Can you make a meanSdPlot for rlog and VST, respectively, and then compare those with untransformed data? Can you use the same ranges for Y axis? Does the transformation fix your previous problem on high variance of low counts?

Hints for changing the Y ranges.

```
  library("ggplot2")
  msd <- meanSdPlot(assay(vsd), ranks=FALSE)
  msd$gg + ggtitle("") + scale_y_continuous(limits = c(0, 1))
```

Next we will introduce two useful analyses for visualization of sample to sample differences/distance. The principal components (PCA) plot is a useful diagnostic for examining relationships between samples. In PCA, the high dimensional data are projected into a lower dimensional space (usually 2D), where the largest variability is retained.

```
plotPCA(log.norm, intgroup="dex")
```

In addition, we can plot a dendrogram based on hierarchical clustering on Euclidean distance matrix.

```
plot(hclust(dist(t(log.norm.counts))), labels=colData(dds)$dex)
```

Exercise: Can you make a PCA plot for rlog? Can you make a PCA plot for VST? How do you interpret PC1 and PC2 before and after variance Can you make a dendrogram for rlog and VST? Based on HC plot, can you comments on the effect of variance stablizing transformation?

Step 6: Differential gene expression

1) Modeling counts using a negative binomial distribution

We will now perform differential gene expression on the counts, to try to find genes in which the differences in expected counts across samples due to the condition of interest rises above the biological and technical variance we observe.

We will use an overdispersed Poisson distribution – called the negative binomial – to model the raw counts in the count matrix. The model will include the size factors into account to adjust for sequencing depth. The formula will look like:

$$K_{ij} \propto \mathrm{NB}(s_{ij} q_{ij}, \alpha_i)$$

where K_{ij} is a single raw count in our count table, s_{ij} is a size factor or more generally a normalization factor, q_{ij} is proportional to gene expression (what we want to model with our design variables), and α_i is a dispersion parameter.

For the negative binomial, the variance parameter is called disperison, and it links the mean value with the expected variance. The reason we see more dispersion than in a Poisson is mostly due to changes in the proportions of genes across biological replicates – which we would expect due to natural differences in gene expression.

```
par(mfrow=c(3,1))
n <- 10000
brks <- 0:400
hist(rpois(n,lambda=100),
     main="Poisson / NB, disp=0",xlab="",breaks=brks,col="black")
hist(rnbinom(n,mu=100,size=1/.01),
     main="NB, disp = 0.01",xlab="",breaks=brks,col="black")
hist(rnbinom(n,mu=100,size=1/.1),
     main="NB, disp = 0.1",xlab="",breaks=brks,col="black")
```

The square root of the dispersion is the coefficient of variation – SD/mean – after subtracting the variance we expect due to Poisson sampling.

```
disp <- 0.5
mu <- 100
v <- mu + disp * mu^2
sqrt(v)/mu
sqrt(v - mu)/mu
sqrt(disp)
```

A number of methods for assessing differential gene expression from RNA-seq counts use the negative binomial distribution to make probabilistic statements about the differences seen in an experiment. A few such methods are edgeR, DESeq2, and DSS. Other methods, such as limma+voom find other ways to explicitly model the mean of log counts and the observed variance of log counts.

DESeq2 performs a similar step to limma in using the variance of all the genes to improve the variance estimate for each individual gene. In addition, DESeq2 shrinks the unreliable fold changes from genes with low counts, which will be seen in the resulting MA-plot.

2) Experimental design and running DESeq2

Remember, we had created the DESeqDataSet object earlier using the following line of code (or alternatively using DESeqDataSetFromMatrix).

```
dds <- DESeqDataSet(airway, design= ~ cell + dex)
```

First, we setup the design of the experiment, so that differences will be considered across time and protocol variables. We can read and if necessary reset the design using the following code.

```
design(dds)
design(dds) <- ~ cell + dex
```

The last variable in the design is used by default for building results tables (although arguments to results can be used to customize the results table), and we make sure the “control” or “untreated” level is the first level, such that log fold changes will be treated over control, and not control over treated.

```
dds$dex <- relevel(dds$dex, "untrt")
levels(dds$dex)
```

The following line runs the DESeq2 model. After this step, we can build a results table, which by default will compare the levels in the last variable in the design, so the dex treatment in our case:

```
dds <- DESeq(dds)
res <- results(dds)
```

3) Examining results tables

```
head(res)
table(res$padj < 0.1)
```

A summary of the results can be generated:

```
summary(res)
```

For testing at a different threshold, we provide the alpha to results, so that the mean filtering is optimal for our new FDR threshold.

```
res2 <- results(dds, alpha=0.05)
table(res2$padj < 0.05)
```

4) Visualizing results

The MA-plot provides a global view of the differential genes, with the log2 fold change on the y-axis over the mean of normalized counts:

```
plotMA(res, ylim=c(-4,4))
```

We can also test against a different null hypothesis. For example, to test for genes which have fold change more than doubling or less than halving:

```r
res.thr <- results(dds, lfcThreshold=1)
plotMA(res.thr, ylim=c(-4,4))
```

A sorted results table:

```r
resSort <- res[order(res$padj),]
head(resSort)
```

Examine the counts for the top gene, sorting by p-value:

```r
plotCounts(dds, gene=which.min(res$padj), intgroup="dex")
```

A more sophisticated plot of counts:

```r
library(ggplot2)
data <- plotCounts(dds, gene=which.min(res$padj), intgroup=c("dex","cell"), returnData=TRUE)
ggplot(data, aes(x=dex, y=count, col=cell)) +
  geom_point(position=position_jitter(width=.1,height=0)) +
  scale_y_log10()
```

Connecting by lines shows the differences which are actually being tested by results given that our design includes cell + dex

```r
ggplot(data, aes(x=dex, y=count, col=cell, group=cell)) +
  geom_point() + geom_line() + scale_y_log10()
```

A heatmap of the top genes:

```r
library(pheatmap)
topgenes <- head(rownames(resSort),20)
mat <- assay(rld)[topgenes,]
mat <- mat - rowMeans(mat)
df <- as.data.frame(colData(dds)[,c("dex","cell")])
pheatmap(mat, annotation_col=df)
```

Exercise: Can you plot histogram for p-values? With what you have learned from previous lectures, is the histrogram with a desired shape? Can you make a heatmap based on top 50 genes? What do you observe?

5) Getting alternate annotations

We can then check the annotation of these highly significant genes:

```r
library(org.Hs.eg.db)
keytypes(org.Hs.eg.db)
anno <- select(org.Hs.eg.db, keys=topgenes,
               columns=c("SYMBOL","GENENAME"),
               keytype="ENSEMBL")
anno[match(topgenes, anno$ENSEMBL),]
```

Exercise: Based on the annotation, what is biological functions of top genes? Can you run a GO enrichment analysis using top 100 genes? You can use the following package or this website (http://geneontology.org/page/go-enrichment-analysis).

```r
library(clusterProfiler)
top100genes <- head(rownames(resSort), 100)
annoList <- anno[match(top100genes, anno$ENSEMBL),]
ego <- enrichGO(gene          = annoList$ENSEMBL,
                universe      = rownames(resSort),
                OrgDb         = org.Hs.eg.db,
                keyType       = 'ENSEMBL',
                ont           = "CC",
                pAdjustMethod = "BH",
                pvalueCutoff  = 0.01,
                qvalueCutoff  = 0.05,
                readable      = TRUE)
head(ego)
```

6) Looking up different results tables

The `contrast` argument allows users to specify what results table should be built. See the help and examples in `?results` for more details:

```
results(dds, contrast=c("cell","N061011","N080611"))
```

Exercise: Would you check the biological functions of top significantly differential expressed genes between cell line N061011 and N080611? Would you perform GO enrichment analysis on top genes as well? Any significant findings? Do you want to make any conclusion? Why or why not?

Hints:

```
res_contrast <- results(dds, contrast=c("cell","N061011","N61311"))
res_contrastSort <- res_contrast[order(res_contrast$padj),]
top100genes <- head(rownames(res_contrastSort), 100)
annoList <- anno[match(top100genes, anno$ENSEMBL),]
ego <- enrichGO(gene          = annoList$ENSEMBL,
                OrgDb         = org.Hs.eg.db,
                universe      = rownames(res_contrastSort),
                keyType       = 'ENSEMBL',
                ont           = "CC",
                pAdjustMethod = "BH",
                pvalueCutoff  = 0.01,
                qvalueCutoff  = 0.05,
                readable      = TRUE)
head(ego)
```

Follow-up activities

`DESeq2` can be used to analyze time course experiments, for example to find those genes that react in a condition-specific manner over time, compared to a set of baseline samples. Here we demonstrate a basic time course analysis with the fission data package, which contains gene counts for an RNA-seq time course of fission yeast (Leong et al. 2014). The yeast were exposed to oxidative stress, and half of the samples contained a deletion of the gene `atf21`. We use a design formula that models the strain difference at time 0, the difference over time, and any strain-specific differences over time (the interaction term strain:minute).

```
library("fission")
data("fission")
ddsTC <- DESeqDataSet(fission, ~ strain + minute + strain:minute)
```

The following chunk of code performs a likelihood ratio test, where we remove the strain-specific differences over time. Genes with small p values from this test are those which at one or more time points after time 0 showed a strain-specific effect.

```
ddsTC <- DESeq(ddsTC, test="LRT", reduced = ~ strain + minute)
```

Exercise: Can you visualize results from the above analysis? Can you perform functional analysis? Can you propose other tests on this dataset?

Acknowledgement

This workshop contains online materials written by Rafael Irizarry and Michael Love.

Session information

Here is the session information.

```
devtools::session_info()
```

Workshop on Computational Immunobiology

Aly A. Khan[1], Evan Kiefl[2]
[1]Course Instructor; aakhan@uchicago.edu
[2]Course Assistant; ekiefl@uchicago.edu

There's a joke about immunology, *which Jessica Metcalf of Princeton recently told me. An immunologist and a cardiologist are kidnapped. The kidnappers threaten to shoot one of them, but promise to spare whoever has made the greater contribution to humanity. The cardiologist says, "Well, I've identified drugs that have saved the lives of millions of people." Impressed, the kidnappers turn to the immunologist. "What have you done?" they ask. The immunologist says, "The thing is, the immune system is very complicated ... " And the cardiologist says, "Just shoot me now."* — Yong, Ed. "Immunology Is Where Intuition Goes to Die", The Atlantic, Aug. 2020.

Welcome. This workshop will introduce a few useful concepts for computationally thinking about the immune system. The field of immunology is currently in the midst of a data-driven revolution, where advances in computer science and high-throughput technologies are allowing us to approach questions that previously could not be answered using purely experimental approaches or standard reductionist techniques. In this workshop we will explore two concepts:

1. What are some ways to characterize the phenotypic diversity and population structure of immune cells?
2. How can we harness statistical modeling to make useful predictions about the immune system?

BSD courses: Other courses that may broaden your experience in quantitative and computational immunology include: Microbial 'Omics (BIOS 25420) and Quantitative Immunobiology (IMMU 34800).

What is the immune system? The immune system can be thought of as a loosely connected network of cells that interact to solve problems beyond the individual capacities of each single cell. At the same time, the immune system contains the capacity to exchange messages between cells, collectively coordinate action, and retain memory of past events. The field of immunology has developed a rich and specialized body of knowledge detailing the shared biological mechanisms underlying host defense, transplantation, autoimmunity, tumor immunology, allergy and other clinical challenges.

The role of the immune system in host defense is often framed by an attacked host (e.g. self) against alien invaders (e.g. non-self; viruses, bacteria, cancer cells, etc). This conflict comes with serious consequences. The immune system must precisely function to distinguish between self and non-self, as collateral damage and misfiring can lead to serious harm (e.g. autoimmunity, allergy, etc). Strikingly, we still do not fully understand how the immune system accomplishes this exquisite role or why certain individuals develop autoimmunity.

As you progress in your graduate training, we hope you take note that there are a myriad of exciting opportunities available to biologists willing to cross and explore the traditional boundaries between immunology and computer science.

1. Visualizing the population structure of immune cells

This section is adapted from Neu et al., Trends in Immunology, 2017.

In the past decade advances in computing and next-generation sequencing technologies have ushered in a new era of discovery in immunology. In particular, single cell RNA-seq (scRNA-seq) has enabled an unprecedented view of gene expression in single cells. A key challenge lies in visualizing single cell gene expression data in a biologically meaningful way while remaining robust to the high levels of noise that is present in single cell data.

scRNA-seq is RNA-seq performed on an individual cell. The cellular mRNA is amplified through oligos specific for the 5′ or 3′ tail of mRNA molecules or random hexamers.

One of the most compelling applications of single-cell genomics to immunology resides in characterizing the population structure of single cells. Visualization of scRNA-seq data can help to identify rare and intermediate subpopulations that are often overlooked with bulk

RNA-seq data. The goal of visualization algorithms is to project high-dimensional data into a low-dimensional space, thus resolving cellular groups based on their transcriptional similarity without any predetermined markers dictating their identity. In this section, we review two standard dimensionality reduction algorithms used to visualize scRNA-seq data.

Dimensionality reduction is a process to reduce the number of variables to a compressed set of principal variables. More specifically, dimensionality reduction can be understood as projecting the data from the original high-dimensional space into much lower-dimensional space, while (roughly) capturing the concerned statistical properties (e.g., variation, distribution) and/or structure property (e.g., clusters). High-dimensional data after dimensionality reduction are easier to store and faster for downstream computation. Moreover, when projecting the data to two- or three-dimensional space, it is also easier for visualization.

Exercise 1.1

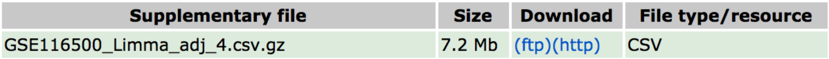
Exercise 1.1 — Overview discussion

Our goal for this exercise is to load and examine some real world scRNA-seq data. If you are new to scRNA-seq, let's take a moment to consider how and what format such data could be stored in. Let's also discuss what might be some prerequisite steps for analyzing it, such as quality control. We can also review concepts such as data sparsity, dimensionality reduction, and batch correction.

Exercise 1.1 — Data wrangling

Let's begin by downloading some scRNA-seq data. For this exercise we will be using data published as part of a study examining certain B cells in humans after influenza vaccination (Neu et al., JCI, 2019). We will download pre-processed supplementary data from the GEO database: `https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE116500`. Scroll to the bottom of the page to locate the supplementary data.

GEO is a database managed by the NIH and functions as a public repository of high throughput sequencing and microarray data.



Supplementary file	Size	Download	File type/resource
GSE116500_Limma_adj_4.csv.gz	7.2 Mb	(ftp)(http)	CSV

Figure 1: The supplementary data can be found at the bottom of the page, and has has already been pre-processed.

The resulting csv.gz file contains single cell gene expression data for nearly 300 B cells. Let's load the data in R:

```r
# load data
geo <- read.csv('./data/GSE116500_Limma_adj_4.csv.gz',
    row.names = 1, header = TRUE)
```

How many genes and cells are in the data? Let's try looking at the dimensions of the data:

```
# number of rows (Genes) and columns (cells)
dim(geo)
num_genes <- dim(geo)[1]
num_cells <- dim(geo)[2]
```

To made things easier for subsequent steps, let's take the transpose of the data so that each row denotes a cell and each column denotes a gene. The transpose of a matrix is an operation which flips a matrix over its diagonal; that is, it switches the row and column indices. This can facilitate certain types of linear algebraic operations and calculations which operate on columns by default.

```
# current data with genes by cells
dim(geo)

# transpose data to cells by genes
geo <- t(geo)
dim(geo)
```

Our assumption is that most gene expression in scRNA-seq data contains random noise due to technical variation. We would like to focus our analyses on genes with high variability, which may have a biological basis. For the purposes of this exercise we simply calculate variance of gene expression as a way to rank genes, but we note other methods (e.g. coefficient of variation) can be used as well.

```
# Let's identify highly variable genes based on
# variance.
VARs <- apply(geo, 2, var)

# What is the distribution of variances?
hist(VARs)
```

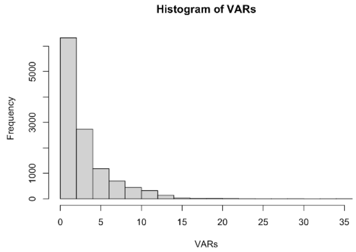


Figure 2: Histogram of gene expression variances.

Exercise 1.1 — Results discussion

What does our analysis of gene expression variance suggest to us about what is changing inside these cells? How should this observation inform our downstream tasks, such as data visualization?

Exercise 1.2

Exercise 1.2 — Overview discussion

Our goal for this exercise is to visualize our scRNA-seq data. Taking into account our observations about gene expression variances, how might we want to filter or preprocess our data? What is our intuition about good low-dimensional projections? Also, how can we evaluate the quality of our visualization?

Exercise 1.2 — Data Visualization

As we discussed earlier, we would like to focus our analyses on genes with high variability, which may have a biological basis. Let's select some top highly variable genes.

```r
# we usually pick the top 1000 - 2000 highly
#   variable genes (HVGs)
hvgs <- names(sort(VARs, decreasing = TRUE))[1:2000]
geo.hvg <- as.data.frame(geo[,hvgs])
dim(geo.hvg)
```

We should now have a matrix of 295 cells with the top 2000 most highly variable genes. One fact that we did not reveal earlier is that these cells are plasmablasts, which are B cells that secrete antibodies. We will now classify the type (or isotype) of antibodies these B cells are secreting by comparing the expression of IgA and IgG genes. In order to help qualitatively examine various types of dimensionality reduction techniques, we will classify and label each cell with their most highly expressed isotype:

Immunoglobulin isotype can be thought of as a molecular classification for antibodies.

```
# There are multiple genes that encode subclasses
# of the two isotypes in this data: IgA and IgG

# Grab max IgA values across all genes
IgA <- cbind(geo.hvg$IGHA1,geo.hvg$IGHA2)
IgA_max <- as.matrix(apply( IgA, 1, max))
#Set column name to 'IgA'
colnames(IgA_max) <- 'IgA'

# Grab max IgG values across all genes
IgG <- cbind(geo.hvg$IGHG1,geo.hvg$IGHG2,
    geo.hvg$IGHG3,geo.hvg$IGHG4)
IgG_max <- as.matrix(apply( IgG, 1, max))
#Set column name to 'IgG'
colnames(IgG_max) <- 'IgG'

#Determine if IgA is higher or IgG is higher
Ig <- cbind(IgA_max,IgG_max)
Ig_max <- colnames(Ig)[(apply( Ig, 1, which.max))]
```

We have now classified cells as IgA or IgG expressing plasmablasts.

Principal component analysis (PCA) is a linear dimensionality reduction algorithm that is often the first-step in visualizing high-dimensional data. We will need to use some functions from ggplot2 in order to visualize our scRNA-seq data using PCA. Let's load the R package:

PCA is a linear dimensionality reduction algorithm, used to project high dimension data into a few 'components' that capture most of the variability in the data. It is a popular visualization technique that can help identify patterns or connections between samples.

```
library("ggplot2")
```

As a general rule of thumb, if you get an error message saying there is no package titled ggplot2 you may need to first install the appropriate package:

```
install.packages("ggplot2")
```

PCA takes an input of correlations between cells based on gene expression data, and identifies principal components corresponding to linear combinations of genes, which cumulatively capture the variability of the total dataset. When the data is projected against the first few components, which account for the largest amount of variation, distinct populations can be visually and biologically interpreted. Let's perform PCA:

```r
# Let's perform PCA
geo.pca <- prcomp(geo.hvg, center = TRUE,
    scale = TRUE)
plot_pc_data <- data.frame(
    PC1=geo.pca[["x"]][,'PC1'],
    PC2=geo.pca[["x"]][,'PC2'])
```

Let's use the top two principal components to visualize our data:

```r
#plot PCA results with Ig status
ggplot(plot_pc_data, aes(x=geo.pca[["x"]][,'PC1'],
    y=geo.pca[["x"]][,'PC2'], color=Ig_max)) +
    geom_point(shape=1) + theme_minimal() +
    geom_point(aes(color = Ig_max)) +
    theme(legend.position = "top")
```

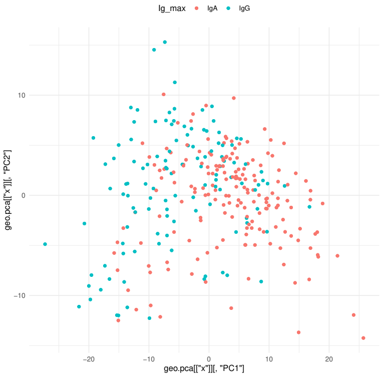


Figure 3: Visualizing our scRNA-seq data with PCA.

Do IgA cells and IgG cells separate well using PCA? Could we use other principal components?

T-DISTRIBUTED STOCHASTIC NEIGHBOR EMBEDDING (T-SNE) is a widely used nonlinear dimensionality reduction algorithm. Unlike PCA, which seeks to capture variance in data, t-SNE seeks to explicitly preserve the local structure of the original data. t-SNE constructs a probability distribution to describe the data set such that pairs of similar cells are assigned a high probability, while dissimilar pairs are assigned a much smaller probability. Thus, cells that are similar in the high-dimensional space will cluster together (due to high probability) in low-dimensional space. This ability to explicitly maintain clustering of similar cells is an advantage of t-SNE over direct linear transformation such as PCA. This approach is very effective with scRNA-seq data, and has been used

t-SNE is a nonlinear dimensionality reduction method, which seeks to preserve the local structure of data in high-dimensional space when projected into low-dimensional space.

to resolve transcriptionally distinct populations that are indistinguishable with PCA. We will need to use some functions from Rtsne, so let's load the R package:

```
library("Rtsne")
```

If you get an error message saying there is no package titled Rtsne you may need to first install the package.

```
# Let's use the top 10 PC for t-SNE
geo.tsne <- Rtsne(geo.hvg, theta = .001,
        perplexity = 30, initial_dims = 10)
plot_tsne_data <- data.frame(
    tsne1=geo.tsne$Y[,1],
    tsne2=geo.tsne$Y[,2])
```

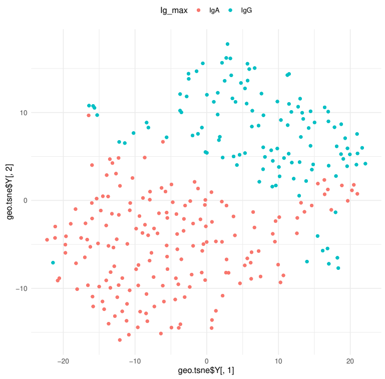


Figure 4: Visualizing our scRNA-seq data with t-SNE.

Let's visualize our data:

```
#plot tsne results with Ig status
ggplot(plot_tsne_data, aes(x=geo.tsne$Y[,1],
    y=geo.tsne$Y[,2], color=Ig_max)) +
    geom_point(shape=1) + theme_minimal() +
    geom_point(aes(color = Ig_max)) +
    theme(legend.position = "top")
```

Do IgA cells and IgG cells separate well using t-SNE?

Exercise 1.2 — Results discussion

What does separability mean? What is "good" separability? Does it matter? How could we perform clustering on our visualizations? What would that tell us about the population structure of cells in the data? What might happen if we keep IgA and IgG as our cell labels but we remove expression information for all the underlying Ig genes and then visualize the data?

Conclusion

While single cell transcriptional profiles have high dimensionality due to the thousands of genes profiled, their intrinsic dimensionalities are typically much lower. Thus, unsupervised low dimensional projections can reveal salient structure in scRNA-seq datasets. However, the choice of dimensionality reduction algorithms used for visualization needs careful thought in immunology.

2. *Peptide-MHC interactions*

Nearly every cell in our body is decorated with a class of molecules known as major histocompatibility complex (MHC). The function of the MHC is to bind peptide fragments (antigens) derived from pathogens and display them on the cell surface for recognition by components of the immune system. In particular, alien peptides may be recognized by cytotoxic T cells, which can destroy infected cells. Understanding the binding affinity of MHC proteins and the repertoire of cognate peptide ligands is important for advancing our understanding of the antigenic landscape in infectious diseases, autoimmunity, vaccine design, and cancer immunotherapy.

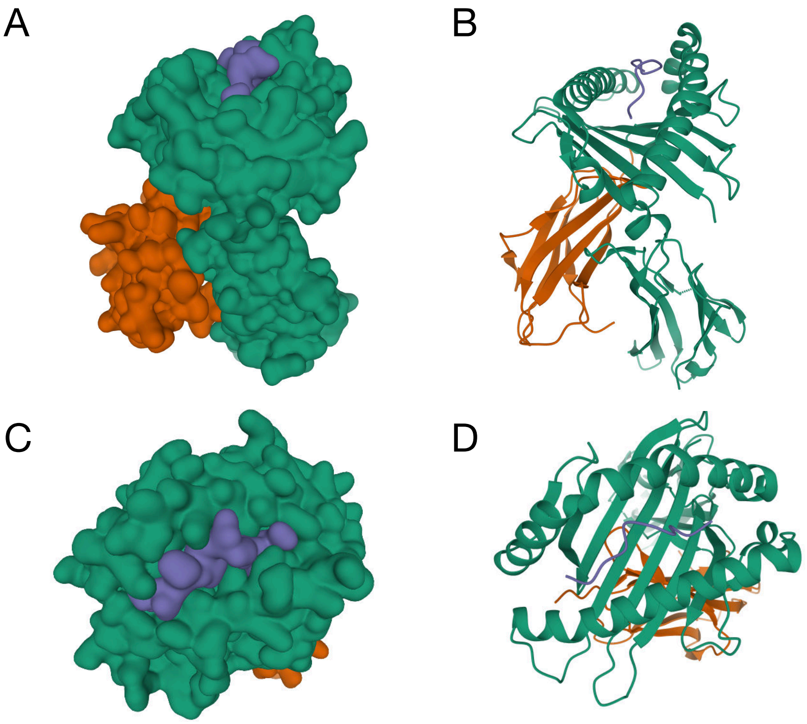


Figure 5: Structure of a peptide-MHC complex. (A) Front view of the crystal structure 1I4F depicting a space-filled molecular surface representation of the MHC (HLA-A*02:01) protein bound with a peptide (derived from MAGEA4, in purple), and (B) a backbone ribbon representation of the MHC. The heavy chain of MHC (alpha), which contains the binding cleft, is depicted in green. The supporting light chain (Beta-2 Microglobulin) is depicted in brown. (C,D) Top view of the complex showing the surface of the MHC (green) and the exposed surface of the bound peptide (purple). The exposed part of the peptide is referred to as the "TCR-interacting surface" of the peptide-MHC complex.

Due to the importance of this process, peptide-MHC binding has been experimentally studied in a variety of ways. For example, the relative binding ability of different peptides to a specific MHC molecule can be directly assessed by competition experiments. The result of such experiments is a set of relative binding energies for different MHC-peptide combinations. In this section, we will model the repertoire of peptide ligands associated with a specific MHC molecule.

Exercise 2.1

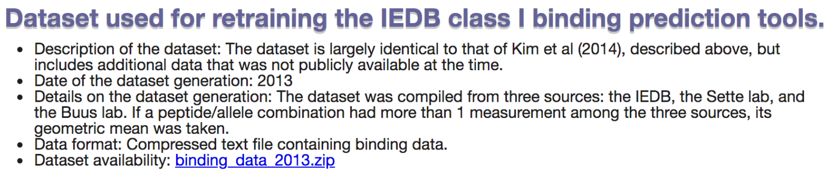
Exercise 2.1 — Overview discussion

Our goal for this exercise is to load some HLA-peptide sequence and binding affinity data. If you are new to analyzing sequence data (or string data structures), let's take a moment to consider how this data structure differs from numerical data structures, and what are some operations we might want to perform on biological strings, such as sequence alignment. We can also review concepts such as edit distance and k-mers.

Exercise 2.1 — Data wrangling

High-throughput screening through competition experiments have resulted in large datasets cataloging binding affinities between various MHC molecules and peptides. Let's begin by downloading some peptide-MHC binding data. For this exercise we will be using data from the The Immune Epitope Database (IEDB) `http://tools.iedb.org/mhci/download/`

IEDB is a public database of immune epitope information. The database contains data related to antibody and T cell epitopes for humans, non-human primates, rodents, and other animal species. In particular, the database contains extensive MHC class I binding data from a variety of different antigenic sources.



Dataset used for retraining the IEDB class I binding prediction tools.

- Description of the dataset: The dataset is largely identical to that of Kim et al (2014), described above, but includes additional data that was not publicly available at the time.
- Date of the dataset generation: 2013
- Details on the dataset generation: The dataset was compiled from three sources: the IEDB, the Sette lab, and the Buus lab. If a peptide/allele combination had more than 1 measurement among the three sources, its geometric mean was taken.
- Data format: Compressed text file containing binding data.
- Dataset availability: binding_data_2013.zip

Figure 6: The binding data is available in the MHC class I section at the top (binding_data_2013.zip).

Let's download: `binding_data_2013.zip`

Once the peptide-MHC binding data is downloaded, you can uncompress the file. The resulting tab-delimited file contains nearly 200,000 peptide-MHC combinations. Let's load the data using R:

```r
# load data
iedb <- read.csv('./bdata.20130222.mhci.txt',
    header = TRUE, sep = "\t", as.is =TRUE)

# let's use head to view a snippet of the data
head(iedb)
```

You should see the first few lines of the file, including the header for the columns. Let's take a moment to interpret what the values mean for each of the columns:

species This is the species from which a specific MHC allele was evaluated for peptide binding.

mhc This is the specific MHC allele.

peptide_length MHC class I molecules bind peptides that are predominantly 8-10 amino acid in length. Traditionally, there has been a focus on 9mer peptides when mapping HLA-I restricted T cell epitopes.

sequence This is the sequence of the peptide.

inequality This reflects the uncertainty for some of the peptide MHC binding data, where there some reported affinities are either an upper-bound or lower-bound to the true binding affinity.

meas The predicted output is given in units of IC50 nM. Therefore a lower number indicates higher affinity. As a rough guideline, peptides with IC50 values <50 nM are considered high affinity.

Exercise 2.1 — Results discussion

WHY IS THIS INTERESTING? Several T-cell based cancer immunotherapies are being engineered to drive antitumor immune responses for specific antigens presented by the human MHC allele HLA-A*02:01. In cancer, somatic mutations altering the amino acid sequence of endogenous protein coding genes can result in the generation of tumor-specific HLA-presented antigenic peptide epitopes (or neo-antigens). These neo-antigens have the potential to activate cytotoxic T lymphocytes (CD8+ T cells) of the host immune system through HLA class I molecules, thereby provoking an anti-tumor immune response.

It stands to reason that if we knew the binding specificity of a given MHC, we could evaluate different somatic mutations in a cancer sample and determine if it could be presented by the cancer. Given the data available, can we model the repertoire of high affinity peptides that are presented by the human HLA-A*02:01 allele?

Exercise 2.2 — Overview discussion

Our goal for this exercise is to infer the pattern of amino acid specificity in high affinity peptides for a specific HLA. In other words, what positions and letters show a bias or preference for binding HLA-A*02:01? Let's assume each position of a peptide binds independently to the HLA molecule, how could we use intuition from statistics to approach this question? What are the possible letters or alphabet used in the peptides? How can we calculate what are the observed frequencies for a given position?

Exercise 2.2 — Data Visualization

One way to visualize the repertoire of high affinity peptides that can bind to HLA-A*02:01 is to use a sequence logo plot. First, the relative frequency of each amino acid at each position is calculated. This can be referred to as a positional weight matrix (PWM). Second, the logo plot depicts the relative frequency of each character by stacking characters on top of each other, with the height of each character proportional to its relative frequency. The

A **PWM** is a type of scoring matrix in which amino acid substitution scores are inferred separately for each position from a collection of aligned protein sequences.

total height of the letters depicts the information content of the position, in bits. Here, we will use an R package called ggseqlogo to calculate the position specific frequencies for all high affinity 9mer peptides and visualize the sequence logo.

Notably, the Matthew Stephens Lab at UChicago has also developed a sequence logo tool called Logolas.

```
# First let's install a seqlogo tool
install.packages("ggseqlogo")
library(ggseqlogo)

# You can also install Logolas
# BiocManager::install("Logolas")
# library(Logolas)
```

Let's select peptides from our HLA of interest:

```
# let's select human,  'HLA-A*02:01',
# peptides of length 9
# and binding affinity < 50
filtered_iedb = subset(iedb, species=='human'
    & mhc=='HLA-A*02:01'
    & meas < 50
    & peptide_length == 9)
```

Next, let's try to model the distribution in a position specific manner:

```
# let's grab the peptide sequences
listOfSequences = filtered_iedb[,4]

# number of sequences
numSequences = length(listOfSequences)

# length of each sequence
lengthOfSequence = nchar(listOfSequences[1])

# find unique characters in list of sequences
aminoAcidsVocab = unique(strsplit(
    paste(listOfSequences, collapse = ''),
    "")[[1]])
```

```r
# create empty PFM matrix of zeros with dimensions
# of 20 x length of each sequence
PFM = matrix(0L, length(aminoAcidsVocab),
    lengthOfSequence)

# for loop through each sequence
for (sequence in listOfSequences) {

  # transform character vector to vector of
  # single characters # for looping
  sequenceString = strsplit(sequence, "")[[1]]

  # for loop through each amino acid in sequence
  for (index in seq_along(sequenceString)) {

    # increment value at PFM[amino acid
    # at position, positon]
    PFM[match(sequenceString[index],
        aminoAcidsVocab), index] =
    PFM[match(sequenceString[index],
        aminoAcidsVocab), index] + 1
    }
  }

# give rownames with amino acid letter
rownames(PFM) = aminoAcidsVocab
```

Let's pass the list of 9mers to `ggseqlogo` to visualize sequence logo:

```r
ggseqlogo(peptide_sequences, seq_type='aa',
    as.is =TRUE)
```

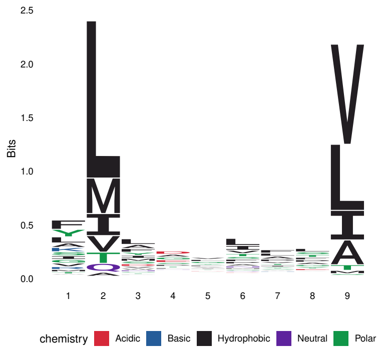


Figure 7: Sequence logo of high affinity peptides for HLA-A*02:01.

Exercise 2.2 — Results discussion

What positions of the high affinity peptides to seem to be highly specific for binding HLA-A*02:01? How does this compare with high affinity binding specificities of other

HLA, for example, HLA-C*06:02?

Conclusion

Although we have made tremendous progress in modelling peptide-MHC interactions over the past several decades, connecting which T cells interact with which MHC-bound antigens remains a challenge. An efficent solution to this problem would have broad applications in improving our understanding of T cells in health, autoimmunity, and cancer (and potentially a free trip to Sweden). This remains a challenging task, in part, due to the enormous number of potential T cell receptors, the diversity of the MHC and bound antigen peptides. However, new computational methods may help resolve TCR-pMHC interactions by integrating and learning complex patterns from diverse high-throughput experimental approaches.

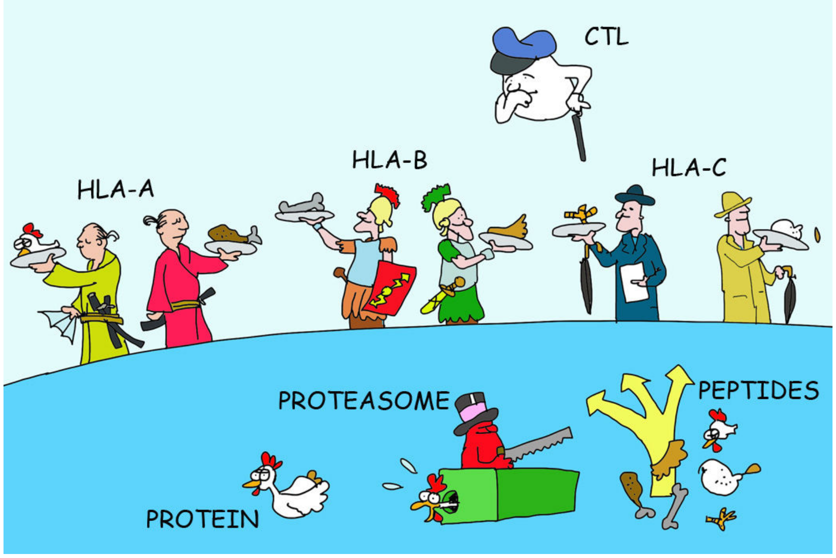


Figure 8: MHC class I overview (Rock, Kenneth L., Eric Reits, and Jacques Neefjes. "Present yourself! By MHC class I and MHC class II molecules." Trends in immunology 37.11 (2016): 724-737.) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5159193/figure/F1/

Population genetics workshop

Instructor: **John Novembre**

Welcome

This excercise is going to expose you to several basic ideas in probability and statistics as well as show you the utility of using R for basic statistical analyses. We'll do so in the context of a basic population genetic analysis.

The scenario

As a biologist, you will learn what are the major patterns that are expected when the data you work with is clean. Using that expertise will save you from the mistake of misinterpreting error-prone data. In population genetics, there are a number of patterns that we expect to see immediately in our datasets. In this exercise you will explore one of those major patterns. Rather than give it away — let's begin some analysis and see what we find. In the narative that follows, we'll refine our thinking as we go.

Introductory terminology

- Single-nucleotide polymorphism (SNP): A nucleotide basepair that is *polymorphic* (i.e. it has multiple types or *alleles* in the population)
- Allele: A particular variant form of DNA (e.g. A partciular SNP may have the "A-T" allele in one DNA copy and "C-G" in another; We typically define a reference strand of the DNA to read off of, and then denote the alleles according to the reference strand base - so for example, these might be called simply the "A" and "C" alleles. In many cases we don't care about the precise base, so we might call these simply the A_1 and A_2 alleles, or the A or a alleles, or the 0 and 1 alleles.)
- Minor allele: The allele that is more rare in a population
- Major allele: The allele that is more common in a population
- Genotype: The set of alleles carried by an individual (E.g. AA, AC, CC; or AA, Aa, aa; or 0, 1, 2)
- Genotyping array: A technology based on hybridization with probes and florensence that allows genotype calls to be made at 100s of thousands of SNPs per individual at an affordable cost.

The data-set and basic pre-processing

We will look at Illumina 650Y genotyping array data from the CEPH-Human Genome Diversity Panel. This sample is a global-scale sampling of human diversity with 52 populations in total.

The data were first described in Li et al (Science, 2008) and the raw files are available from the following link: http://hagsc.org/hgdp/files.html. These data have been used in numerous subsequent publications (e.g Pickrell et al, Genome Research, 2009) and are an important reference set. A few technical details are that the genotypes were filtered with a GenCall score cutoff of 0.25 (a quality score generated by the basic genotype calling software). Individuals with a genotype call rate <98.5% were removed, with the logic being that if a sample has many missing genotypes it may be due to poor quality of the source DNA, and so none of the genotypes should be trusted. Beyond this, to prepare the data for the workshop, we have filtered down the indiviudals to a set of 938 unrelated individuals. (For those who are interested, the data are available as plink-formatted files `H938.bed` `H938.fam`, `H938.bim` from this link: http://bit.ly/1aluTln). We have also extracted the basic counts of three possible genotypes.

The files with these genotype frequencies are your starting points.

Note about logistics

We will use some functions from the `dplyr` and `ggplot2` and `reshape2` libaries so first let's load them:

```
library(dplyr)
library(ggplot2)
library(reshape2)
```

If you get an error message saying there is no package titled `dplyr`,`ggplot2`, or `reshape2` you may need to first run `install.packages("dplyr")`, `install.packages("ggplot2")`, or `install.packages("reshape2")` to install the appropriate package.

We will not be outputting files - but you may want to set your working directory to the `sandbox` sub-directory in case you want to output some files.

The `MBL_WorkshopJN.Rmd` file has the R code that you can run. Code is provided for most steps, but some you will need to devise for yourselves to answer the questions that are part of the workshop narrative.

Initial view of the data

Read in the data table:

```
# g <- read.table("../data/H938_chr15.geno", header=TRUE)
g <- read.table("../data/H938_Euro_chr15.geno", header=TRUE)
# g <- read.table("../data/H938_chr2.geno", header=TRUE)
```

It will be read in as a dataframe in R.

Then use the "head" command to see the beginning of the dataframe:

```
head(g)
```

You should see that there are columns each with distinct names.

```
CHR         SNP  A1  A2 nA1A1 nA1A2 nA2A2
```

- CHR: The chromosome number. In this case all SNPs are from chromosome 15.
- SNP: The rsid of a SNP is a unique identifier for a SNP and you can use the rsid to look up information about a SNP using online resource such as dbSNP or SNPedia.
- A1: The minor allele at the SNP.
- A2: The major allele.
- nA1A1 : The number of A1/A1 homozygotes.
- nA1A2 : The number of A1/A2 heterozygotes.
- nA2A2 : The number of A2/A2 homozygotes.

Calculate the number of observations at each locus

Next compute the total number of observations by summing each of the three possible genotypes. Here we use the `mutate` function from the `dplyr` library to do the addition and add a new column to the dataframe in one nice step. (Note: You could also use the `colSums` function from the base R library).

```
g <- mutate(g, nObs = nA1A1 + nA1A2 + nA2A2)
```

Run `head(g)` and confirm your dataframe `g` has a new column called `nObs`.

Now use the `summary` function to print a simple summary of the distribution:

```
summary(g$nObs)
```

The `ggplot2` library has the ability to make "quick plots" with the command `qplot`. If we pass it a single column it will make a histogram of the data for that column. Let's try it:

```
qplot(nObs, data = g)
```

Our data are from 938 individuals. When the counts are less than this total, it's because some individuals had array data that was difficult to call a genotype for and so no genotype was reported.

Question: Do most of the SNPs have complete data?
Question: What is the lowest count observed? Is this number in rough agreement with what we know about how the genome-wide missingness rate filter was set to >98.5% of all SNPs?

Calculating genotype and allele frequencies

Let's move on to calculating genotype and allele frequencies. For allele A_1 we will denote its frequency among all the samples as p_1, and likewise for A_2 we will use p_2.

```
# Compute genotype frequencies
g <- mutate(g, p11 = nA1A1/nObs , p12 = nA1A2/nObs, p22 = nA2A2/nObs )
# Compute allele frequencies from genotype frequencies
g <- mutate(g, p1 = p11 + 0.5*p12, p2 = p22 + 0.5*p12)
```

Question: With a partner or group member, discuss whether the equations in the code for p_1 and p_2 are correct and if so, why?

Run `head(g)` again and confirm that `g` now has the extra columns for the genotype and allele frequencies.

And let's plot the frequency of the major allele (A2) vs the frequency of the minor allele (A1). The `ggplot2` library has the ability to make "quick plots" with the command `qplot`. Let's try it here:

```
qplot(p1, p2, data=g)
```

Notice that $p_2 > p_1$ (be careful to inspect the axes labels here) This makes sense because A_1 is supposed to be the minor (less frequent) allele. Note also that there is a linear relationship between p_2 and p_1

Question: What is the equation decribing this relationship?

The relationship exists because there are only two alleles - and so their proportions must sum to 1. The linear relationship you found exists because of this constraint. It also provides a nice check on our work (if p_1 and p_2 didn't sum to 1 it would suggest something is wrong with our code!).

Plotting genotype on allele frequencies

Let's look at an initial plot of genotype vs allele frequencies. We could use the base plotting functions, but the following uses the `ggplot2` commands. These are a little trickier, but end up being very compact (we need fewer lines of code overall to achieve our desired plot). To use `ggplot2` commands effectively our data need to be what statisticans call "tidy" (in this case, that means with one row per point we will plot).

To do this, we first subset the data on the columns we'd like (using the `select` command and listing the set of columns we want), then we pass this (using the `%>%` operator) to the `melt` command which will reformat the data for us, and output it as `gTidy`:

```
gTidy <- select(g, c(p1,p11,p12,p22)) %>% melt(id='p1',value.name="Genotype.Proportion")
head(gTidy)
ggplot(gTidy) + geom_point(aes(x = p1,
                               y = Genotype.Proportion,
                               color = variable,
                               shape = variable))
```

Now let's look at the graph that we produced. There is some scatter in the relationship between genotype proportion and allele frequency for any given genotype, but at the same time there is a very regular underlying relationship between these variables.

Question: What are approximate relationships between p_{11} vs p_1, p_{12} vs p_1, and p_{22} vs p_1? (Hint: These look like parabolas, which suggests are some very simple quadratic functions of p_1).

You might start to recognize that these are the classic relationships that are taught in introductory biology courses. If you recall, under assumptions that there is no mutation, no natural selection, infinite population size, no population substructure and no migration, then the genotype frequencies will take on a simple relationship with the allele frequencies. That is: $p_{11} = p_1^2$, $p_{12} = 2p_1(1 - p_1)$ and $p_{22} = (1 - p_1)^2$. In your basic texts, they typicaly use p and q for the frequencies of allele 1 and 2, and present these *Hardy-Weinberg proportions* as: p^2,$2pq$, and q^2.

Another way to think of the Hardy-Weinberg proportions is in the following way. If the state of an allele (A_1 vs A_2) is *independent* within a genotype, then the probability of a particular genotype state (such as A_1A_1) will be determined by taking the product of the alleles within it (so $p_{11} = p_1p_1$ or p_1^2).

Let's add to the plot lines that represent Hardy-Weinberg proportions:

```
ggplot(gTidy)+
  geom_point(aes(x=p1,y=Genotype.Proportion,color=variable,shape=variable))+
  stat_function(fun=function(p) p^2, geom="line", colour="red",size=2.5) +
  stat_function(fun=function(p) 2*p*(1-p), geom="line", colour="green",size=2.5) +
  stat_function(fun=function(p) (1-p)^2, geom="line", colour="blue",size=2.5)
```

On average, the data follow the classic theoretical expectations fairly well. It is pretty remarkable that such a simple theory has some bearing on reality!

By eye, we can see that the fit isn't perfect though. There is a systematic deficiency of heterozygotes and excess of homozygotes. Why?

Let's look at this more closely and more formally…

Testing Hardy Weinberg

Pearson's χ^2-test is a basic statistical test that can be used to see if count data o_i conform to a particular expectation. It is based on the X^2-test statistic:

$$X^2 = \sum_i \frac{(o_i - e_i)^2}{e_i}$$

which follows a χ^2 distribution under the null hypothesis that the data are generated from a multinomial distribution with the expected counts given by e_i.

Here we compute the test statistic and obtain its associated p-value (using the `pchisq` function). We keep in mind that there is 1 degree of freedom (because we have 3 observations per SNP, but then they have to sum to a single total sample size, and we have to use the data once to get the estimated allele frequency, which reduces us down to 1 degree of freedom).

```
g <- mutate(g, X2 = (nA1A1-nObs*p1^2)^2 /(nObs*p1^2) +
              (nA1A2-nObs*2*p1*p2)^2 / (nObs*2*p1*p2) +
              (nA2A2-nObs*p2^2)^2 / (nObs*p2^2))
g <- mutate(g,pval = 1-pchisq(X2,1))
```

The problem of multiple testing

Let's look at the p-values for the first SNPs:

```
head(g$pval)
```

How should we interpret these? A p-value gives us the frequency at which the observed departure from expectations (or a more extreme departure) would occur if the null hypothesis (the SNP follows HW proportions) is true. As an agreed upon standard (of the frequentist paradigm for statistical hypothesis testing), if the observation is relatively rare under the null (e.g. p-value $< 5\%$), we reject the null hypothesis, and we would infer that the given SNP departs from Hardy-Weinberg expectations. This is problematic here though. The problem is that we are testing many, many SNPs (Use `dim(g)` to remind yourself how many rows/SNPs are in the dataset). Even if the null is universally true, 5% of our SNPs would be expected to be rejected using the standard frequentist paradigm. This is called the multiple testing problem. As an example, if we have 20,000 SNPs, that all obey the null hypothesis, we would on average naively reject the null for ~1000 SNPs based on the p-values < 0.05.

We clearly need some methods to deal with the "multiple testing problem". Two frameworks are the Bonferroni approach and false-discovery-rate (FDR) approaches. We will not say more about these here. Instead, we will do two simple checks to see though if our data are globally consistent with the null.

First, let's see how many tests have p-values less than 0.05. Is it much larger than the number we'd expect on average given the total number of SNPs and a 5% rate of rejection under the null?

```
sum(g$pval < 0.05, na.rm = TRUE)
```

Wow - we see many more. This is our first sign that although by eye these data show qualitiative similarities to HW, statistically they are not fitting Hardy-Weinberg well enough.

Let's look at this another way. A classic result from Fisher is that under the null hypothesis the p-values of a well-designed test should be distributed uniformly between 0 and 1. What do we see here?

```
qplot(pval, data = g)
```

The data show an enrichment for small p-values relative to a uniform distribution. Notice how the whole distribution is shifted towards small values - The data appear to systematically depart from Hardy-Weinberg.

Plotting expected vs observed heterozygosity

To understand this more clearly, let's make a quick plot of the expected vs observed heterozygosity (the proportion of heterozygotes):

```
qplot(2*p1*(1-p1), p12, data = g) + geom_abline(intercept = 0,
                                                 slope=1,
                                                 color="red",
                                                 size=1.5)
```

Most of the points fall below the y=x line. That is, we see a systematic deficiency of heterozygotes (and this implies a concordant excess of homozygotes). This general pattern is contributing to the departure from HW seen in the X^2 statistics.

Discussion: Population subdivision and departures from Hardy-Weinberg expectations

We might wonder why the departure from Hardy-Weinberg proportional is directional, in that, on average, we are seeing a deficiency of heterozygotes (and excess of homozygotes). One enlightening way to understand this is by thinking about what Sewall Wright (a former eminent University of Chicago professor) called "the correlation of uniting gametes". To produce an A_1A_1 individual we need an A_1-bearing sperm and an A_1-bearing egg to unite. If these events were independent of each other, we would expect A_1A_1 individuals at the rate predicted by multiplying probabilites, that is, p_1^2 (an idea we introduced above). However, what if uniting gametes are positively correlated, in that an A_1-bearing sperm is more likely to join with an

A_1-bearing egg? In this case we will have more A_1A_1 individuals than predicted by p_1^2, and conversely fewer A_1A_2 individuals than predicted by $2p_1p_2$. If our population is structured somehow such that A_1 sperm are more likely to meet with A_1 eggs, then we will have such a positive correlation of uniting gametes, and the resulting excess of homozygotes and deficiency of heterozygotes.

Given the HGDP data is from 52 sub-populations from around the globe, and alleles have some some probability of clustering within populations, a good working hypothesis for the deficiency of heterozygotes in this dataset is the presence of some population structure.

While statistically significant, the population structure appears to be subtle in absolute terms — based on our plots, we have seen the genotype proportions are not wildly off from HW proportions.

Question: As an exercise, compute the average deficiency of heterozygotes relative to the expectated proportion. This is the average of

$$\frac{2p_1(1-p_1)-p_{12}}{2p_1(1-p_1)}$$

What is this number for this data-set? A common "rule-of-thumb" for this deficiency in a global sample of humans is approximately 10%. Do you find this to be true from the data?

Just ~10% difference between expected and observed seems pretty remarkable given these samples are taken from across the globe. It is a reminder that human populations are not very deeply structured. Most of the alleles in the sample are globally widespread and not sufficiently geographically clustered to generate correlations among the uniting alleles. This is because all humans populations derived from an ancestral population in Africa around 100-150 thousand years ago, which is relatively small amount of time for variation across populations to accumulate.

Finding specific loci that show large departure from Hardy-Weinberg proportions

Now, let's ask if we can find any loci that are wild departures from HW proportions. These might be loci that have erroneous genotypes, or loci that cluster geographically in dramatic ways (such that they have few heterozygotes relative to expectations).

To find these loci, we'll compute the same relative deficiency you computed above, but let's look at it per SNP. This number is referred to as F by Sewall Wright and has connections directly to correlation coefficients (advanced exercise: Try to work this out!). If we assume there is no inbreeding within populations, this number is an estimator of F_{ST} (a quantity that appears often in population genetics).

Let's add this value to our dataframe and plot how it's value changes across the chromosome from one end to another:

```r
g <- mutate(g, F = (2*p1*(1-p1)-p12) / (2*p1*(1-p1)))
plot(g$F, xlab = "SNP number")
```

There are a few interesting SNPS that show either a very high or low F value.

Now, here's a trick. When a high or low F value is due to genotyping error, it likely only effects a single SNP. However, when there is some population genetic force acting on a region of the genome, it likely effects multiple SNPs in the region. So let's try to take a local average in a sliding window of SNPs across the genome, computing an average F over every 5 consecutive SNPs (in real data analysis we might use 100kb or 0.1cM windows).

The `stats::filter` command below calls the `filter` function from the `stats` library. The code instructs the function to take 5 values centered on a focal SNP, weighting them each by 1/5 and then taking the sum. In this way it produces a local average in a sliding window of 5 SNPs. Let's define the `movingavg` function and then make a plot of its values:

```r
movingavg <- function(x, n=5){stats::filter(x, rep(1/n,n), sides = 2)}
avgF <- movingavg(g$F)
plot(avgF, xlab="SNP number")
```

Wow — there appears to be one large spike where the average F is approximately 60% in the dataset!

Let's extract the SNP id for the largest value, and look at the dataframe:

```r
outlier=which (movingavg(g$F) == max(movingavg(g$F),na.rm=TRUE))
outlier=which.max (avgF)
g[outlier,]
```

Question: Which SNP is returned? By inserting the rs id into the UCSC genome browser (https://genome.ucsc.edu/), and following the links, find out what gene this SNP resides near. The gene names should start with "SLC.." What gene is it?

Question: Carry out a literature search on this gene using the term "positive selection" and see what you find. It's thought the high F value observed here is because natural selection led to a geographic clustering of alleles in this gene region. Discuss with your partners why this might or might not make sense.

Discussion: The outlier region

The region you've found is one of the most differentiated betwen human populations that is known. Notice in your literature search, how it is known to affect skin pigmentation and is thought to contribute to differences in skin pigmentation that are seen between human populations. Finding strong population structure for alleles that affect external morhological phenotypes is not uncommon when looking at other chromosomes. Some of the most differentiated genes that exist in humans are those that involve morphological phenotypes - such as skin pigmentation, hair color/thickness, and eye color (the genes OCA2/HERC2, SCL45A2, KITLG, EDAR all come to mind). Many of these are thought to have arisen due to direct or indirect effects of adaptation to local selective pressures (e.g. adaptation to varing levels of UV exposure, local pathogens, local diets, local mating preferences), though in most cases we still do not yet have a fully convincing understanding of their evolutionary histories. Regardless of the reasons, it is notable that many of the features that humans see externally in each other (i.e. the morphological differences) are controlled by genes that are outliers in the genome. At most variant SNPs, the patterns of variation are much closer to those of a single random mating populations than they are at variant sites like EDAR. Put another way, a genomic perspective shows us many of the differences people see in each other are in a sense, just skin-deep.

Wrap-up

Modern population genetics has a lot of additional tools on its workbench, but here using relatively simple and classical ideas combined with genomic-scale data, we have been able to observe and interpret some major features of human genetic diversity. We have also revisited some basic concepts of probablity and statistics such as indepedence vs correlation, the χ^2 test, and the problems of multiple testing. One remarkable thing we saw is that a very simple mathematical model based on assuming independence of alleles and genotypes can predict genotype proportions within ~10% of the true values. This gives us a hint of how simple mathematical models may be useful even in the face of biological complexity. Finally, we have gained more familiarity with R. We didn't discuss how genotyping errors might create Hardy-Weinberg departures, but if we were doing additional analyses, we could use Hardy-Weinberg departures to filter them from our data. It's common practice to do so, but with a Bonferonni correction and using data from within populations to do the filtering.

Follow-up activities

In the `data` folder, we are including data files that you can explore to gain more experience. These include global data for other chromosomes (`H938_chr*.geno`) and the same data but limited to European populations (`H938_Euro_chr*.geno`). Here are a few suggested follow-up activities. It may be wise to split the activities across class members and reconvene after carrying them out.

Follow-up actvity: Look at a chromosome from the European-restricted data - is the global deficiency in heterozygosity as strong as it was on the global scale? Before you begin, what would you expect to see?

Follow-up activity Using the European data, do you find any regions of the genome that are outliers for F on chromosome 2? Using genome browsers and/or literature searches, can you find what is the likely locus under selection for that region?

Follow-up activity: Using the global data or the European data, analyze other chromosomes – do you find other loci that show high F values?

References

Li, Jun Z, Devin M Absher, Hua Tang, Audrey M Southwick, Amanda M Casto, Sohini Ramachandran, Howard M Cann, et al. 2008. "Worldwide Human Relationships Inferred from Genome-Wide Patterns of Variation." Science 319 (5866): 1100–1104.

Pickrell, Joseph K, Graham Coop, John Novembre, Sridhar Kudaravalli, Jun Z Li, Devin Absher, Balaji S Srinivasan, et al. 2009. "Signals of Recent Positive Selection in a Worldwide Sample of Human Populations." Genome Research 19 (5): 826–37.