# Markdown and the `knitr` Package in R

Practice of Data Science

Communicating with Code

# Baggerly and Coombes

There's a really infamous talk from 2010, called

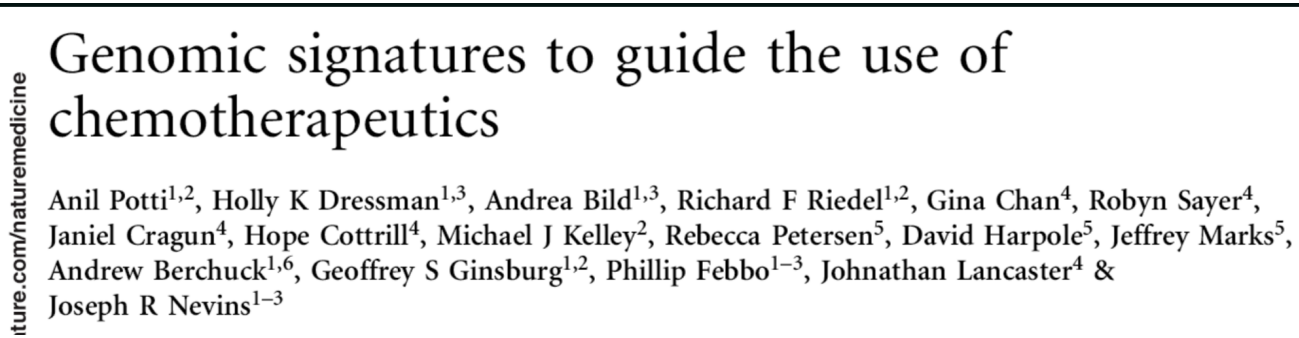
*The Importance of Reproducible Research in High-Throughput Biology: Case Studies in Forensic Bioinformatics*:
https://www.youtube.com/watch?v=7gYIs7uYbMo

by Keith Baggerly and Kevin R. Coombes.


Their talk illustrates what can go seriously wrong when people working with data are not transparent about how they get their results.

# Baggerly and Coombes

Baggerly and Coombes are bioinformaticians who study cancer. They attempted to replicate this study:



ture.com/naturemedicine

Genomic signatures to guide the use of chemotherapeutics

Anil Potti[1,2], Holly K Dressman[1,3], Andrea Bild[1,3], Richard F Riedel[1,2], Gina Chan[4], Robyn Sayer[4], Janiel Cragun[4], Hope Cottrill[4], Michael J Kelley[2], Rebecca Petersen[5], David Harpole[5], Jeffrey Marks[5], Andrew Berchuck[1,6], Geoffrey S Ginsburg[1,2], Phillip Febbo[1–3], Johnathan Lancaster[4] & Joseph R Nevins[1–3]

This paper claims to show how treatments for childhood leukemia should be tailored to patients based on the specific genomic information in the patient's DNA.

This was an important finding and was published in a prominent journal.

# Baggerly and Coombes

In order to replicate the study, Baggerly and Coombes started with the same raw dataset that the study's authors used. Their goal was to use the data to reproduce the study's results.

But the problem was that the authors **did not provide any scripts or discussion** of what they did to the data prior to running the tests.


As a result, Baggerly and Coombes had to reproduce the results in a "forensic" way: figuring out after-the-fact what the authors must have done to get these results.

# Baggerly and Coombes

Eventually, they were able to reproduce the results, but found the authors had made two errors:

1. The rows in the data refer to particular genes. Someone copy-and-pasted the cells in a way that left off the column headers. That created an **off-by-one** error where the data for a gene were listed one row above the gene name.

2. The treatment was coded as 1 or 2, but someone along the way confused what 1 and 2 meant. So the treated patients were reported as control, and vice versa.

That means that the reported positive effect for the treatment group is actually a positive effect for the control group. In other words, *the treatment harms people*.

# Baggerly and Coombes

By the time Baggerly and Coombes made this discovery, the research had moved forward to the stage of clinical drug trials. **Children with leukemia were being given harmful treatments based on mistaken research.**

In his talk, Baggerly describes all the ways that he and Coombes tried to sound the alarm on this research. But the stakeholders were reluctant to retract and end the research because of the implications for reputation and grant money.

The study continued for many months. It was only stopped when it was revealed that the principal investigator on the original study had lied on his CV about being a Rhodes Scholar.

# Baggerly and Coombes

**So, what exactly went so wrong here?**

1. Dumb, typical mistakes that people make with spreadsheets all the time (the off-by-one error, confusing the 1s and 2s).

2. Laziness: no effort to document the steps that were taken to prepare the data for analysis.

3. Self-interest and ego: covering up mistakes instead of risking the penalties of correcting them, thereby making the mistakes worse.

4. Magical thinking: because the work involves data, there's a tendency by most people to simply believe that the work is correct without digging in to it (not Baggerly and Coombes though!)

# Baggerly and Coombes

If we are going to be working with data, **how can we avoid the mistakes** that Baggerly and Coombes discovered?

1. Mistakes are inevitable. But, if we use code instead of point-and-clicking, it's easier to see mistakes and to go back and correct them.

2. If we document our steps as we go along, we'll be transparent and able to show anyone exactly what we did with the data.

3. We can feed our egos in a different way: clear and professional documentation looks impressive to others.

4. Working with code and explaining what each part of the code does goes a long way towards dispelling the anxiety people have about data, and overcomes magical thinking.

# Reproducible Research

Our goal: to give you the skills and practice you need to work with data in a way that

- Is easy to document
- Allows you to combine code, results, and text to better convey the context of what you are doing
- Looks *really good*

This morning we will discuss R markdown, one of the best tools we have for conducting transparent research.

This afternoon we will walk through an entire research pipeline using R markdown, documenting everything we need to do to raw data to prepare it for analysis, and including the final results in the document.

# Reproducible Research

Reproducibility — the ability to get the same research results as an original study by using the raw data and computer code provided by researchers.

There have been serious crises in many fields, including medicine, economics, and political science, because researchers failed to make their code and data available.

We are going to learn how to make a document that shows ALL the steps involved in a project, going **from raw data to final results**. We also want this document to be as easy to read and understand as possible.

**Practice on your own computer as we discuss the steps for creating a markdown document.**

# There are two ways to save R code

The first way is to save code in an **R script**.

- Pros: Scripts are easy to work with, run, save, and share.
- Cons: Scripts can be hard for even an advanced R user to read and understand.

The second way is with an **R markdown document** compiled with the `knitr` package. (If you've never used `knitr` before, install it by typing `install.packages("knitr")` into the console.)

- Pros: Creates a beautiful, readable document by placing text, code, and the output of the code all in the same document (this is also called weaving: hence the name `knitr`). Able to create HTML, PDF, or Word files.
- Cons: More syntax to learn in addition to R code. Might take a while to compile documents.

# The Markdown Language

You need to become experienced **using both methods** to save code. They are each useful in different situations.

Markdown is a programming language for formatting text, using minimal programming syntax. It's basically a lightwight version of HTML code. It's not just for R — it's for anything that involves text together with some other kind of code.

To start a markdown file: open R Studio, click File, then New File, then R Markdown. Give the document a title and author, and choose whether you want this code to produce an HTML, PDF, or Word file.

This will call up an example page with some text and code already in it. (You will end up deleting this example text and code and writing your own.)

# Choosing an Output and Compiling

While R scripts are saved as "**.R**" files, these markdown files are saved as "**.Rmd**" files. This file is separate from the HTML, PDF, or Word document you will create with this code.

First notice the "**Knit**" button. This button compiles the document – produces the desired output. To change the type of output, click the arrow to the right of Knit.

At the top of the .Rmd file, there's code separated on the top and bottom by three dashes. This is the header. You can change the title, author, etc. here. You can also change the output here:

- For HTML output, `output: html_document`
- For PDF output, `output: pdf_document`
- For Word output, `output: word_document`

# Using the Header to Set Options

To create a table of contents that lists up to 5 levels of sectioning:

```
output:
  html_document:
    toc: true
    toc_depth: 5
```

By default the table of contents appears at the top of the document, just under the title. But, you can also use a floating and collapsable table of contents window like this:

```
output:
  html_document:
    toc: true
    toc_depth: 5
    toc_float: true
    toc_collapsed: true
```

# Using the Header to Set Options

Important: You will see three dashes `---` at the beginning and end of the header. Do not delete these! The document will not compile if you do. (It won't even give you an intelligible error message)

To change the **overall theme** (colors, fonts, etc.) of the document, add the theme argument, like this:

```
output:
  html_document:
    theme: cerulean
```

Different themes are illustrated here:
https://www.datadreaming.org/post/r-markdown-theme-gallery/
Try a few right now.

Other options for the header are listed here:
http://rmarkdown.rstudio.com/html_document_format.html

# Publishing the HTML Output

If you choose to compile your markdown code as an HTML file, it displays in R Studio's makeshift browser. That's good enough for checking your work.

But remember, HTML code creates webpages. So you can click the "**Open in Browser**" button to see your code displayed in a web browser.

If you have your own webpage, you can post this HTML output to your webpage.

If you need space on the web to host this page, click on "**Publish**", then click "**RPubs**". RPubs is a free service, run by R Studio, that provides server space for your markdown documents. If you post online using RPubs, you can use a URL to share your work with your audience.

# Markdown syntax

To insert text into the output, just type into the .Rmd file and compile.

The goal of markdown is to give you an easy way to create stylized text, sections, equations, lists, tables, etc. quickly **by typing**.

For *italicized text*, place ONE star * before and after the text.

For **bold text**, place TWO stars ** before and after the text.

For ~~struck through text~~, place TWO tildes ~~ before and after the text.

```
*this will be italicized*
**this will be bold**
~~this will be struck out~~
```

# Markdown syntax

To start a new paragraph, push Enter/Return TWICE, so that there is a blank line separating the paragraphs.

For a **hyperlink**, either type the address itself (it will automatically become a link), or use syntax like this to place the link on top of other text:

```
[The best web comic](https://smbc-comics.com/)
```

For block quotes, push Enter/Return TWICE then start every line of the quote with `>` and a space.

```
Here's a profound quote:

> I'd rather have this bottle in front of me
> than a frontal lobotomy
```

# Sectioning

One of the most important ways to make a document readable is to use **sectioning** to organize the document.

Two hashtags (pound signs) `##` followed by a name denote a section.

Three hashtags `###` followed by a name denote a subsection.

Four hashtags `####` followed by a name denote a sub-subsection, and so on.

One hashtag is used for titles, but generally, these titles appear too large and look weird, so most people start at two hashtags.

If the `toc: true` option is specified, the section titles will appear in the **table of contents** automatically.

# Tabbing

Sometimes it makes sense for only one of several sub-sections to appear at a time. For example, suppose I wrote a paragraph in English and Spanish. I could have the two sections represented by tabs, in which the user can switch between the two tabs.

To create tabs, type `{.tabset}` immediately after the section title. Then all sub-sections within this section will exist in tabs.

`{.tabset .tabset-fade}` does the same thing, but include a nice fade-in animation when switching between tabs.
`{.tabset .tabset-fade .tabset-pills}` places the tabs into squares with rounded-edges.

# Equations

In statistics and data science, sometimes it makes sense to include mathematical equations in the document. In Microsoft Word, you have to point-and-click every symbol, but in Markdown, you can quickly **type out equations**.

To include an equation in-line, place a dollar sign `$` before and after the equation. To place the equation on its own line, place TWO dollar signs and a space before and after the equation.

Some special math characters:

- `^` exponentiation
- `_` subscripts
- `\ sqrt{5}` $\sqrt{5}$
- `\ frac{1}{2}` $\frac{1}{2}$

A list of the code for many other math symbols is here: http://reu.dimacs.rutgers.edu/Symbols.pdf

# Equations

For example, to include the quadratic formula in your document,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

type

```
$$ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} $$
```

(Here the `\ pm` refers to the "plus or minus" symbol)

# Lists

To create an unordered list (just bullets), type:

```
* Item 1
* Item 2
    * Item 2a
    * Item 2b
```

To create an ordered list (numbered), type:

```
1. Item 1
2. Item 2
3. Item 3
    a. Item 3a
    b. Item 3b
```

# Lists

Some *weird* things about lists in R Markdown:

1. It won't compile as a list unless there's an entire blank line (push enter twice) separating the list from the preceding text

2. To create a sublist, you need to **tab twice**. Once won't register differently from the other items

3. For numbered lists, the top level must be regular (Arabic) numbers (1,2,3,...). The next levels can be lowercase letters or lowercase Roman numerals

4. You can change the first number to anything you want. But the following numbers will **always count up by 1** from the first number, no matter what you type there

# Creating tables by hand

There are two kinds of tables: ones you enter by hand and ones you can create with R. To create a nice-looking table by hand, markdown has a peculiar notation.

In general, I don't like memorizing the specifics of this notation. Instead I use https://tableconvert.com/ (or another website like it). Then:

1. In the top toolbar, click the "Table" button to choose the desired dimensions
2. Fill in the table however you want
3. Select "Copy"
4. And paste the syntax into your R markdown document

The table will appear in a neatly formatted way when you compile.

# Graphics (from external files)

Sometimes you might want to display a graphic in your document that you aren't using R to create. For example, what kind of monster wouldn't want to include this graphic?

You can include graphics if you have the file on your local machine:

```
![A caption, if wanted, goes here](duck.jpg)
```

Or pull them directly off the web by entering the image's URL:

```
![](http://1funny.com/wp-content/uploads/2010/08/Baby-Animals-24.jpg)
```

# Embedding Code

Remember that the purpose of an R markdown file is to weave text, code, and the results of code together in one, readable document. There are three ways to include code in a document:

1. in-line for display only,
2. in-line and evaluated,
3. and evaluated inside a "code chunk."

In-line code for display only is just for referring to specific R commands as you are writing. Markdown will place these pieces of code in a different, computery font with a grey background. But this code WILL NOT be evaluated or run.

To write in-line code, use single, forward-sloping quotes `` ` `` (on the same key as the tilde). Then if you write about the `lm()` function, or the `ggplot2` package, it will appear in this different font and have a grey background.

# In-line Code that is Evaluated

If you use the single, forward-sloping quotes, the code is displayed but not run. Alternatively, if you type `r` prior to any code within the quotes, markdown will evaluate the code and display the output in the text.

For example, try typing the following into your markdown document:

```
The current date and time are `r Sys.time()`
```

`Sys.time()` is R's internal clock, so this syntax should display the current date and time as regular text, not as computer code.

This feature is great for filling in details about the data into your text automatically.

# Code Chunks

A code chuck contains several lines of code, on separate lines from the text. It gets **run by R**, and the output appears in the document as well.

Start a code chunk like this:

```
```{r nameofthischunk}
```

First, type three forward single-quotes. Then within curly braces, the letter `r` — this tells markdown that this is R code, then a distinct name you give this code chunk. Naming the chunks is optional, but will help you isolate errors if there are any.

End a code chunk by typing **three more forward single-quotes** on a new line:

```
```
```

# Code Chunks

You can type as many lines of R code as you want inside one code chunk. But, best practice is to only write a few lines at a time in one code chunk.

The reason is that you are trying to bring a reader along and explain your code. It's easier to explain a few lines at a time.

Also, If there's multiple outputs from one chunk, they will **all be displayed** after the chunk. Keeping the chunk small helps us keep track of which output goes with which code.

Take a moment to try out some code chunks in your markdown document.

# Code Chunk Options

You can write options inside the curly braces, separated by commas, to change the behavior of a code chunk.

Here are some options you can use:

`echo=FALSE` — don't display the code

`eval=FALSE` — don't display the results

`warning=FALSE, message=FALSE, error=FALSE` — don't display warnings, messages, or errors (I always use these options for the code chunk that loads the packages I need)

Other options are listed here: https://yihui.name/knitr/options/

# Code Chunk Options

A new .Rmd file has this code chunk at the top of the document:

````
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
````

This code chunk contains global options to be applied to all subsequent chunks.

By default, it sets `echo=TRUE`, which tells all the chunks to display the code in the document in addition to the output. If this were instead `echo=FALSE`, the output would get displayed but the R code that generates the output would not be displayed.

You can set other global options here if you want them applied to all code chunks. Just write `knitr::opts_chunk$set(`*option*`)` in this chunk.

# Caching

One extremely useful code chunk option is caching the output of the code.

Caching means that R **saves the output of the chunk** so that the next time the document compiles, it can load the saved results instead of running the code again.

To cache the output of a code chunk, use the option `cache=TRUE`.

This option saves A LOT OF TIME when using commands that take a while to run, such as loading a big dataset or running a complicated model. But don't use this option for every chunk, as it can cause problems with the keeping results accurate as code changes.

# Tables and Figures (from code)

A big reason why markdown makes sense for R is the ability to cleanly display tables and figures you produce with R code.

To convert tables from R output to nice looking HTML, use the `kable()` function. Just place the code that creates the table inside of `kable()`.

For figures, such as **ggplot** graphics, no extra function is needed to display. But use the `fig.width` and `fig.height` options on the code chunk to control the size of the figure in the output.

For example:

````
```{r plot, fig.width=6, fig.height=8}
ggplot(mtcars, aes(x=wgt, y=mpg)) + geom_point()
```
````