Creating dynamic research reports using RMarkdown + {knitr}

EDH7916 | Spring 2020

Benjamin Skinner

In this lesson, we'll combine many of the pieces we've already covered — reading in data, cleaning data, making tables and figures — into a single RMarkdown document. We'll purposefully keep it simple at first by reusing some code we've seen before.

But first things first, you'll need to download a bit of software for our RMarkdown documents to properly compile as PDF documents. Specifically, you'll need LaTeX, a typesetting system best known for being able to nicely render mathematical notation but that is really useful for making reproducible documents. Depending on your operating system, you'll want one of these two systems (I'm assuming that if you use a Linux OS, you probably can get LaTeX installed if you haven't already).

- MacTeX (for MacOS)
- MiKTeX (for Windows)

These are large downloads (particularly for MacOS), but the LaTeX system is sufficiently useful in reproducible research that it's worth getting on your machine.

If you are unable to get LaTeX to install properly or cannot get the document to compile as PDF, you should be able to compile to HTML instead.

You will also need the R `knitr` and `rmarkdown` libraries. If you haven't already installed them, type

```r
install.packages(c("knitr","rmarkdown"))
```

into your R console.

What is RMarkdown?

From the RStudio website:

> R Markdown is a file format for making dynamic documents with R. An R Markdown document is written in markdown (an easy-to-write plain text format) and contains chunks of embedded R code…

In other words, an RMarkdown (hereafter *RMD*) document has two basic components:

- R code (in **code chunks**)
- Markdown text (most everything else outside of the code chunks)

RMD documents use the file ending, `*.Rmd`, which makes sense as they combine `R` code with `md` text. To *compile* an RMD file, meaning to

1. convert the plain Markdown text into formatted text
2. run R code, producing all output along the way
3. combine the Markdown text plus R output into a finished document

you will use the `rmarkdown::render()` function, which in turn uses the `knitr::knit()` function under the hood. It can be a bit confusing how all the pieces work together, but luckily, you can use RStudio's point-and-click interface to knit your documents.

Starting a new document

By default, RStudio will fill a new RMarkdown document with some example text that looks like this. You can turn this off in Rstudio's settings, but I think it's helpful to see the skeleton of an RMarkdown document (plus, it's not a big deal to just erase the parts you don't need).

~~~
---
title: "Document Title"
author: "Benjamin Skinner"
date: "3/30/2020"
output: pdf_document
---

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

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax
for authoring HTML, PDF, and MS Word documents. For more details on
using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that
includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to
prevent printing of the R code that generated the plot.
```

Our Document

Rather than using the example document given when opening a new file, we'll go through the example document `test_scores.Rmd` in your scripts folder (and linked above) piece by piece, starting with the YAML header.

YAML header

YAML, which stands for "YAML Ain't a Markup Language", is a common way to configure dynamic documents like RMD documents. The YAML header is this piece of code:

```
---
title: "Test scores from 1980-1985"
```
```

```
author: Benjamin Skinner
date: 30 March 2020
output: pdf_document
---
```

Notice the opening and closing three hyphens (`---`). This is how R knows that this section of code is special. The YAML can become complex, as you add document options, but for now we keep it simple:

- **title**: the document title (printed)
- **author**: the document author (printed)
- **date**: manually set date (printed)
    - leave **date** out of YAML and the date on which the document is compiled will be added automatically
    - set **date** to `""` (empty string) for no printed date
- **output**: document output
    - `pdf_output`: for PDF (uses LaTeX)
    - `html_output`: for web page output (open in browser)
    - `word_output`: for MSWord output (uses MSWord)

We're using `pdf_output` but you either change this setting or override it when compiling the final document.

NB The colon (`:`) is a special character in YAML. Notice that I don't necessarily have to use quotation marks for strings with spaces: I do for the title, but not for my name. That said, if your title includes a colon, you need to wrap the entire title string in double quotation marks (`"`) — otherwise the document won't compile.

NB If you download `test_scores.Rmd` from the link at the top of the page, there's a good chance that the YAML header has been stripped from the file (something to do with the way this site is build and served). If it's missing, just cut and paste the YAML above (including the opening and closing `---`) into the top of the `test_scores.Rmd` file.

Code chunks

In general, an RMD code chunk looks like a markdown code chunk. The key difference between the two is that while a plain markdown code chunk is purely about formatting, the RMD code chunk will try to run the code and print any output:

~~~
```r
## this is just a representation (nothing happens)
x <- rnorm(1000)
```
~~~

Markdown code chunk

~~~
```{r}
## this will produce 1000 random normal values and place in x
x <- rnorm(1000)
```
~~~

RMarkdown code chunk Notice the difference? It's subtle, but notice that the RMD chunk places braces around the **r** after the tick marks: `{r}`. In a normal markdown document, the braces won't mean anything. But in an RMD document, it's different between just printing the code and printing the output.

Code chunk options

~~~
```{r setup, echo=F, include=F, message=F, warning=F, error=F}
## ---------------------------
## libraries
## ---------------------------

library(knitr)
library(tidyverse)

## ---------------------------
## settings/macros
## ---------------------------

## NB:
## - echo (FALSE): don't repeat this code in output
## - include (FALSE): run code, but don't include output (unless a plot)
## - message (FALSE): don't output any messages
## - warning (FALSE): don't output any warnings
## - error (FALSE: don't output any errors
##
## We'll include these in the general knitr::opts_chunk() below, but
## need them here to silence unnecessary output before we can set the options

## set up knitr options
opts_chunk$set(error = FALSE,
               echo = FALSE,
               include = FALSE,
               message = FALSE,
               warning = FALSE,
               fig.path = "../figures/repo-", # where figures should be stored
               dpi = 300,                     # print quality (300 standard for print)
               out.width = "100%",            # figures should be as wide as margins
               comment = NA)                  # if code output, no comment char on LHS

## ---------------------------
## directory paths
## ---------------------------

## read in our data here, assuming we're in scripts like always
dat_dir <- file.path("..", "data")
sch_dir <- file.path(dat_dir, "sch_test")
~~~
```

In our first code chunk, notice how we still load our libraries and set our file paths. For the libraries, we need to load knitr with `library(knitr)` in addition to whichever libraries we need for our analysis.

In addition to our normal analysis setup, notice that we add knitr-specific options in two places.

First, we can set *code chunk-specific* options within the braces that start the code chunk. After `r`, the first word is the name of the chunk. While not strictly necessary to name your chunks, it can come in handy as your documents become more complex: if you get an error, it's much easier to find `data_input` chunk than `unnamed_chunk_38`.

There are a lot of options you can set for your chunks. Here we set the following:
```

- `echo=F` (FALSE): don't repeat this code in output
- `include=F` (FALSE): run code, but don't include output (unless a plot)
- `message=F` (FALSE): don't output any messages
- `warning=F` (FALSE): don't output any warnings
- `error=F` (FALSE): don't output any errors

In sum, this keeps our chunk from echoing the input code into our document and prevents any output. Basically, silence. Sometimes we want our code to echo; sometimes we want output. But since we are making a report, we generally want the underlying code to remain hidden. Readers of our report should only see the write up and any relevant tables and figures — but not all the hard coding we did to make them!

After this first chunk, we can save some typing by setting these options for the rest of the document using `knitr::opt_chunks$set()`. Notice that we include the same settings as above plus a few more:

- `fig.path`: path + prefix for all figures (put them in our `/figures` folder)
- `dpi` (dots per inch): the print quality of our figures; 300 dpi is a nice standard for print (72dpi is sufficient for most web output)
- `out.width`: our figures should fill the line width
- `comment`: if we return code output, don't prepend with `#` or anything — just the output.

There are other options we can use. We can also override these setting as necessary for individual code chunks (as you'll see below). The main idea with the set up code chunk is to get our document settings as close as possible to the way we generally want them.

Chunk to chunk

~~~
```{r input}
## ---------------------------
## input
## ---------------------------

df <- read_csv(file.path(sch_dir, "all_schools.csv"))

```

From 1980 to 1985, students at four schools took end of year exams in
three subjects --- math, reading, and science. While these tests did
not affect students' grades or promotion, they were meant to measure
what students had learned over the course of the school year. In each
year, only 9th grade students took the exam. This means that each year
of data represents a different cohort of 9th grade students. Because
test scores are standardized within subject area, student cohorts can
be compared across time. The table below shows average test scores for
each school in each year.

```{r table_all, include = T}
## ---------------------------
## make table of all scores
## ---------------------------

## use the kable() function in knitr to make nicer table
kable(df,
      digits = 0,
      col.names = c("", "Year", "Math", "Reading", "Science"))
```
~~~

Your coding environment carries from chunk to chunk, meaning that if you read in data in the `input` code chunk as `df`, then `df` will still be available to you in the next chunk `table_all` after writing some Markdown text. This means that you can still organize your RMD scripts like your R scripts (no need to do everything at once in a single huge chunk).

Make a nice(r) table with `knitr::kable()`

~~~
```{r table_all, include = T}
## ---------------------------
## make table of all scores
## ---------------------------

## use the kable() function in knitr to make nicer table
kable(df,
      digits = 0,
      col.names = c("", "Year", "Math", "Reading", "Science"))
~~~
```

Since we want to show all of our data (which isn't very big) and because the data frame `df` already is organized in the way we want to show the data (school by year with different columns for each test), we can just print out the data frame.

But to make it nicer, we use `kable()` which is part of knitr. Even without options, `kable()` will make a nicer table for us. We add `digits = 0` to make sure that we only show whole numbers and we change our column names to something nicer (leaving a blank `""` for the school name column, which is obvious). Notice that in the chunk braces we add `include = T` so that the output — our table — for just this one code chunk will be printed.

Inline code

You can also call R code inline, that is, outside of code chunks proper, inline with your Markdown text.

```
```{r table_averages, include = T}
## ---------------------------
## make table of averages
## ---------------------------

df_tab <- df %>%
    ## group by school
    group_by(school) %>%
    ## get average across years
    summarise(math_mean = mean(math),
              read_mean = mean(read),
              science_mean = mean(science))

## store variables to use in text below
hi_math_sch <- df_tab %>% filter(math_mean == max(math_mean)) %>% pull(school)
hi_math_scr <- df_tab %>% filter(math_mean == max(math_mean)) %>% pull(math_mean)

hi_read_sch <- df_tab %>% filter(read_mean == max(read_mean)) %>% pull(school)
hi_read_scr <- df_tab %>% filter(read_mean == max(read_mean)) %>% pull(read_mean)

hi_sci_sch <- df_tab %>% filter(science_mean == max(science_mean)) %>% pull(school)
hi_sci_scr <- df_tab %>% filter(science_mean == max(science_mean)) %>% pull(science_mean)
```

```
## use the kable() function in knitr to make nicer table
kable(df_tab,
      digits = 0,
      col.names = c("", "Math", "Reading", "Science"))
```

Across the six years of data, `r hi_math_sch` had the
highest average math score (`r hi_math_scr %>% round`);
`r hi_read_sch` had the highest average reading score
(`r hi_read_scr %>% round`); and
`r hi_sci_sch` had the highest average science score
(`r hi_sci_scr %>% round`). However, these six year averages cover a fair
amount of variation within schools across time. In the next sections, I'll
investigate this variation.
```

Inside the code chunk called `table_averages` we do three things:

1. use dplyr to munge our data to get averages
2. store names and scores for high test scores in distinct well named objects:
    - `hi_<test>_sch`: school name with highest average math/read/science score
    - `hi_<test>_scr`: highest average math/read/science score
3. make/print table using `kable()`

In the Markdown text below this code chunk, we call the values using the inline code method

```
`r `
```

that is, single back tick, an `r`, the code we want, then closing back tick. We also pipe the object value to `round` so that we don't return averages with extra and unnecessary decimal points. We could have simply run all the code inline (included what we did in step 2 of the code chunk above), but that would have made for extra messy code.

Being able to incorporate data-driven values directly in your test is very powerful. You can include all kinds of `ifelse()` logic to make complex dynamic documents. Be aware, however, that your document text still needs to make sense. It can be difficult enough writing one clear sentence; having to write a sentence that will remain coherent despite variable inputs can be *very tough*!

Figures

Finally, making figures is pretty much the same as making tables:

````
```{r fig_unadjusted, include = T}
## ---------------------------
## fig: unadjusted
## ---------------------------

## facet line graph, with one column so they stack
p <- ggplot(data = df_long,
            mapping = aes(x = year, y = score, colour = school)) +
    facet_wrap(~ test, ncol = 1, scales = "free_y") +
    geom_line()
p
```
````

Having reshaped our original data frame long (`df_long`), we make a figure just as we've done in the past. While it's not strictly necessary to store the figure in an object (`p`) that we then call, it works just fine.

Notice that again added `include = T` to the chunk brace. Because we added figure options to `opt_chunks$set()`
````

in the setup code chunk, this figure (as well as the next one) is sized so that it fills up the page width (with height determined as a ratio of that width) and printed at 300 dpi quality. If you look in the `figures` folder, you'll see the figure named `repo-fig_unadjusted`, which is the prefix we set above with the name of the code chunk.

### Text

Throughout our RMD file, we've include Markdown text. This text lives outside of the code chunks and is always printed in the final document. It follows normal Markdown text rules, but can have r code placed inline, as seen before.

## Writing an RMD document

Just as when you write a plain R script, your progress from initial RMD draft to final product will be iterative. While you can run R code from inside code chunks just as you've been all semester, you may find it useful to start your analyses in plain R files first and only add them to an RMD document later.

For big projects, such as dissertation, it also doesn't make much sense to put *everything* — data reading, cleaning, analysis, table/figure making — inside a single RMD document. You have to redo your entire workflow each time you compile! For large projects, it might make sense to do all the heavy lifting in separate R scripts — saving cleaned up data sets, tables, and figures along the way — and putting all the pre-establish pieces together at the end.

But for small projects, such a descriptive policy report, a single RMD document might suffice. In the next lesson, we'll work on cleaning up this document to make it even more professional looking.