5: EDA, Big Jobs, Automation

`bit.ly/SISBID3`

Here, we'll talk about capturing exploratory data analysis (EDA), handling big data and big jobs, and project automation.

Regarding exploratory data analysis, we want to capture the whole process: what you're trying to do, what you're thinking about, what you're seeing, and what you're concluding and why. And we want to do so without getting in the way of the creative process.

Regarding long-running jobs, the problems are: (a) it's hard for someone to re-do all of that work, and (b) large-scale calculations tend to be organized in a system-dependent way, so even if time weren't a factor, it'd be harder to transfer the calculations to another system.

Exploratory data analysis

- what were you trying to do?
- what you’re thinking about?
- what did you observe?
- what did you conclude, and why?

We want to be able to capture the full outcome of exploratory data analysis.

But we don’t want to inhibit the creative flow. How to capture this stuff?

Avoid

- ”How did I create this plot?”
- ”Why did I decide to omit those six samples?”
- ”Where (on the web) did I find these data?”
- ”What was that interesting gene?”

I’ve said all of these things to myself.

Basic principles

Step 1: slow down and document.

Step 2: have sympathy for your future self.

Step 3: have a system.

I can't emphasize these things enough.

If you're not thinking about keeping track of things, you won't keep track of things.

One thing I like to do: write a set of comments describing my basic plan, and then fill in the code afterwards. It forces you to think things through, and then you'll have at least a rough sense of what you were doing, even if you don't take the time to write further comments.

Capturing EDA

- copy-and-paste from an R file
- grab code from the `.Rhistory` file
- Write an informal R Markdown file;
  make use of the "R Notebook" features.
- Write code for use with the KnitR function `spin()`

  Comments like `#' This will become text`
  Chunk options like so: `#+ chunk_label, echo=FALSE`

There are a number of techniques you can use to capture the EDA process.

You don't need to save all of the figures, but you do need to save the code and write down your motivation, observations, and conclusions.

I usually start out with a plain R file and then move to more formal R Markdown or AsciiDoc reports.

A file to spin()

```
#' This is a simple example of an R file for use with spin().

#' We'll start by setting the seed for the RNG.
set.seed(53079239)

#' We'll first simulate some data with x ~ N(mu=10, sig=5) and
#' y = 2x + e, where e ~ N(mu=0, sig=2)
x <- rnorm(100, 10, 5)
y <- 2*x + rnorm(100, 0, 2)

#' Here's a scatterplot of the data.
plot(x, y, pch=21, bg="slateblue", las=1)
```

Here's an example R file for use with `spin()`.

Activity

Try out `knitr::spin()`:

- Create an R script using
  - `#'` for text
  - `#+` for chunk options
- Compile the script to HTML using `knitr::spin()`
- Open the resulting HTML file.

A brief activity to try out `knitr::spin()`.

Big jobs

- You don’t want `knitr` running for a year.
- You don’t want to re-run things if you don’t have to.

It may not seem like “big jobs” are that big of a deal, but in my mind this is the only real difficulty in reproducible research.

Okay, there’s also the difficulty that some data can’t be generally distributed due to confidentiality issues.

But aside from subjects’ confidentiality, the main problem is how to manage and capture the really long-running computational analyses where even on the same system it can be a gargantuan effort to reproduce the results.

Biggish jobs in knitr

- Manual caching
- Built-in `cache=TRUE`
- Split the work into a separate subdirectory.

You can put big computations within an R Markdown file, but personally I don't want to wait more than a couple of minutes for it to compile. If it's going to take longer than that, I'll split things up.

And if you are going to have some large computations with knitr, you won't want to re-run all of them every time you make even the smallest change to the text!

That's where you want to cache some computations: save the results and just load the results rather than re-run the code. Unless the code changes, in which case you do want to re-run it (and any other code that may depend on the results).

Manual caching

~~~
```{r a_code_chunk}
file <- "cache/myfile.RData"

if(file.exists(file)) {
  load(file)
} else{

  ....

  save(object1, object2, object3, file=file)
}
```
~~~

This is the “by hand” approach. If the file doesn’t exist, run the relevant code and save the needed results to the file. If the file does exist, just load the file and skip the code.

If you want (or need) to re-run the code, you need to delete the file manually.

One issue: if you want the code to actually be shown, you need to repeat the code: in a chunk that is shown but isn’t run, and then in this chunk that is run but isn’t shown.

You need to be very careful about dependencies.

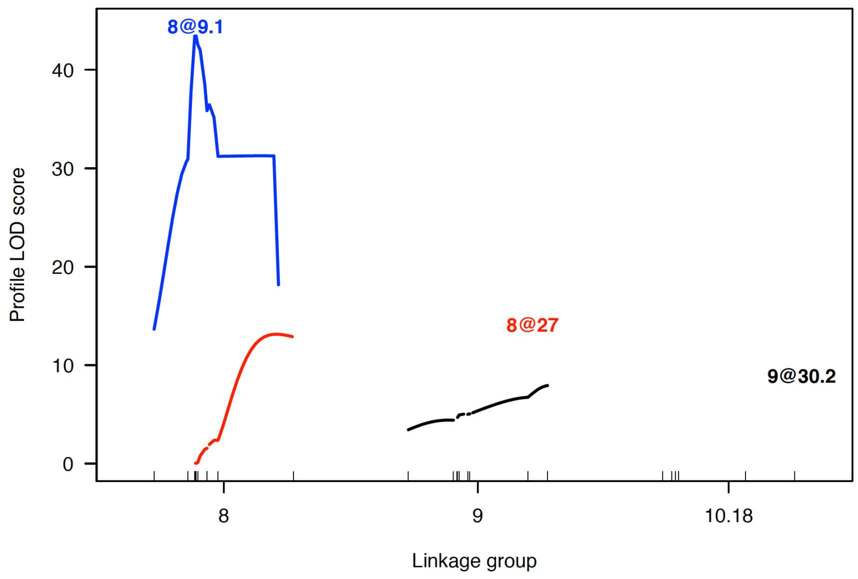
Chunk references

~~~
```{r not_shown, eval=FALSE}
code_here <- 0
```

```{r a_code_chunk, echo=FALSE}
file <- "cache/myfile.RData"

if(file.exists(file)) {
  load(file)
} else{
<<not_shown>>
  save(code_here, file=file)
}
```
~~~

Here's how I'd avoid repeated code: use chunk references.

The `<<not_shown>>` is replaced by the code from that chunk with that label.

A cache gone bad



This is the sort of thing that can happen with manual caching.

This is Fig. 11.14 from my book, A guide to QTL mapping with R/qtl.

I saw it immediately upon flipping through my first paper copy of the printed book.

I'd cached some results, but then changed the underlying software in a fundamental way and didn't update the cache.

Knitr's cache system

~~~
```{r chunk_name, cache=TRUE}
load("a_big_file.RData")
med <- apply(object, 2, median, na.rm=TRUE)
```
~~~

- Chunk is re-run if edited.
- Otherwise, objects from previous run are loaded.
- Don't cache things with side effects

  e.g., `options()`, `par()`

Knitr has a nice built-in system for caching.

A chunk with `cache=TRUE` will be run once and then all objects saved. In future runs, the code won't be run, but rather the cached objects will be loaded.

But some things shouldn't be cached. "Side effects" change the state of things; mostly, this is changing global variables. If these are placed in a cached chunk, the side effects won't be captured when the cache is loaded.

Cache dependencies

Manual dependencies

~~~
```{r chunkA, cache=TRUE}
Sys.sleep(2)
x <- 5
```

```{r chunkB, cache=TRUE, dependson="chunkA"}
Sys.sleep(2)
y <- x + 1
```

```{r chunkC, cache=TRUE, dependson="chunkB"}
Sys.sleep(2)
z <- y + 1
```
~~~

You can indicate dependencies among chunks. Here, if `chunkA` is re-run, the other two will be as well.

Cache dependencies

Automatic dependencies

~~~
```{r setup, include=FALSE}
opts_chunk$set(autodep = TRUE)
dep_auto()
```
~~~

There's also an automatic system for determining dependencies among chunks.

I've not used it.

Parallel computing

If your computer has multiple processors, use `library(parallel)` to make use of them.

- `detectCores()`
- `RNGkind("L'Ecuyer-CMRG")` and `mclapply` (Unix/Mac)
- `makeCluster`, `clustersetRNGStream`, `clusterApply`, and `stopCluster` (Windows)

R has a built-in package for performing parallel computations. A number of instances of R are invoked, calculations begun, and then the results brought back together.

The code can be a bit ugly, but it's not so bad once you get used to it.

See the links on the resources page for Karl's Tools for Reproducible Research course,
`http://kbroman.github.io/Tools4RR/pages/resources.html`

Systems for distributed computing

- At UW-Madison, we use HTCondor
- There are oddles of similar systems
- ”By hand”

For really big jobs, you’ll want to distribute the computations across multiple computers.

At UW-Madison, we use the locally-developed software HTCondor. This provides a way of distributing enormous numbers of jobs across a heterogeneous set of computers and collecting the results.

There exists other, similar systems for distributing and managing jobs across clusters of computers.

My own approach is more primitive: I have a Perl script that converts a template R script into a bunch of R input files (by replacing every instance of “`SUB`” in the template with a job-specific index). It also creates a script to set those running.

In either case, you’d write another R script to combine the results from the multiple jobs.

Simulations

- Computer simulations require RNG seeds (`.Random.seed` in R).
- Multiple parallel jobs need different seeds.
- Don't rely on the current seed, or on having it generated from the clock.
- Use something like `set.seed(91820205 + i)`
- An alternative is create a big batch of simulated data sets in advance.

RNG = Random number generator

Simulations split across multiple CPUs each need their own seed. In R, the seed is saved as `.Random.seed`; if you start all of the simulations from the same directory, they could all get exactly the same seed.

I tend to include a call to `set.seed` at the top of each R script, with the seed being some big number plus an index for the job.

You could, alternatively, generate all of the simulated data sets in advance. An advantage of this is that it'd be easier to reproduce the results later. Just be sure to save (and document) the code you used to generate the data.

Save everything

- RNG seeds
- input
- output
- version numbers, with `sessionInfo()`
- raw results
- script to combine results
- combined results
- `ReadMe` describing the point

This stuff (particularly code input & text output) doesn't take up much space. Compartmentalize it and save it.

Compartmentalize big jobs

- Separate directory for each batch of big computations.
- Collect the results in ‘.RData‘ or ‘.rds‘ files.
- KnitR-based documents for the analysis/use of those results.

My approach, for projects that involve big computations: compartmentalize those big computations into chunks, each in a separate directory to contain all of the materials and results.

I use GNU Make to handle the combination of those results as well as the compilation of any KnitR-based files that describe and carry out the further analyses.

This won’t capture the entire workflow, but it will indicate almost all of it, and the big jobs will be encapsulated as subdirectories, and the source of the major results will be indicated.

Potential problems

- Forgetting `save()` in your distributed jobs
- A bug in the `save()` command
- Keeping things synchronized
  - Have you re-run the big jobs when upstream data were revised?

These are common mistakes I make.

I forget the `save` command and so run a ton of computations and then get no results.

Or my `save` command has an error and so I run a ton of computations and then it dies on the last line of the script.

Or the upstream data changes and I fail to recognize that I need to re-run the big jobs that depend on it.

Big jobs summary

- Careful organization and modularization.
- Save everything.
- Document everything.
- Learn the basic skills for distributed computing.

Research with long-running computations are hard to make fully reproducible. Modularize the big jobs and document their purpose. And document the relationships among things.

Activity

Try out the knitr cache system.

- Create an RMarkdown document with multiple dependent chunks.
- Maybe add `Sys.sleep(5)` to slow things down, as if the jobs were taking a while.
- Compile the document.
- Edit a chunk.
- Re-compile the document and see what was actually run and what was taken from cache.

A brief activity to try out caching in RMarkdown

Automate the process (GNU Make)

```
R/analysis.html: R/analysis.Rmd Data/cleandata.csv
    cd R;R -e "rmarkdown::render('analysis.Rmd')"

Data/cleandata.csv: R/prepData.R RawData/rawdata.csv
    cd R;R CMD BATCH prepData.R

RawData/rawdata.csv: Python/xls2csv.py RawData/rawdata.xls
    Python/xls2csv.py RawData/rawdata.xls > RawData/rawdata.csv
```

GNU Make is an old (and rather quirky) tool for automating the process of building computer programs. But it's useful much more broadly, and I find it valuable for automating the full process of data file manipulation, data cleaning, and analysis.

In addition to automating a complex process, it also documents the process, including the dependencies among data files and scripts.

Automation with GNU Make

- `Make` is for more than just compiling software
- The essence of what we're trying to do
- Automates a workflow
- Documents the workflow
- Documents the dependencies among data files, code
- Re-runs only the necessary code, based on what has changed

People usually think of Make as a tool for automating the compilation of software, but it can be used much more generally.

To me, Make is the essential tool for reproducible research: automation plus the documentation of dependencies and workflows.

Fancier example

```
FIG_DIR = Figs

mypaper.pdf: mypaper.tex $(FIG_DIR)/fig1.pdf $(FIG_DIR)/fig2.pdf
    pdflatex mypaper

# One line for both figures
$(FIG_DIR)/%.pdf: R/%.R
    cd R;R CMD BATCH $(<F)

# Use "make clean" to remove the PDFs
clean:
    rm *.pdf Figs/*.pdf
```

As I said, you can get really fancy with GNU Make.

Use variables for directory names or compiler flags. (This example is not a good one.)

Use pattern rules and automatic variables to avoid repeating yourself. With `%`, we have one line covering both `fig1.pdf` and `fig2.pdf`. The `$(<F)` is the file part of the first dependency.

Look at the manual for Make and the many online tutorials, such as the one from Software Carpentry, or `http://kbroman.org/minimal_make`.

Installing Make

- On Macs, Make should be installed. Type `make --version` to check.
- On Windows, probably the easiest is to install Rtools, which includes Make.

  `cran.r-project.org/bin/windows/Rtools`

Installation of these sorts of command-line tools on Windows can be a bit difficult.

How do you use Make?

- If you name your make file `Makefile`, then just go into the directory containing that file and type `make`
- If you name your make file `something.else`, then type `make -f something.else`
- Actually, the commands above will build the first target listed in the make file. So I'll often include something like the following.

  ```
  all: target1 target2 target3
  ```

  Then typing `make all` (or just `make`, if `all` is listed first in the file) will build all of those things.
- To be build a specific target, type `make target`. For example, `make Figs/fig1.pdf`

Details on the use.