MRC Biostatistics Unit

UNIVERSITY OF CAMBRIDGE

Reproducible Research

lessons learnt from software development

IEEE DSAA 2018, Kevin Kunzmann

02/10/2018

About me

- studied mathematics…
- got into statistics and programming …
- started working in biostatistics …
- realized how hard it is to make analyses **truely** reproducible …
- developed a reproducibility-fetish …
- but **not** a software developer!
- `kevin-kunzmann@mrc-bsu.cam.ac.uk` / @kevin_kunzmann

Outline

1. What is reproducible research and why do we need more of it?
2. Version control
3. Literate programming
4. Build automation
5. Containerization

- incremental approach adding ‘layers’ of reproducibility
- live demos introducing new techniques hands-on
- ultimate challange: get the sample analysis running on your system!
- code online at `https://github.com/kkmann/reproducibleresearch`

Assumed prerequisites

- Unix system
- some basic bash
- install docker / singularity:

```
$ sudo apt-get install docker-ce
$ sudo apt-get install singularity-container
```

- basic git

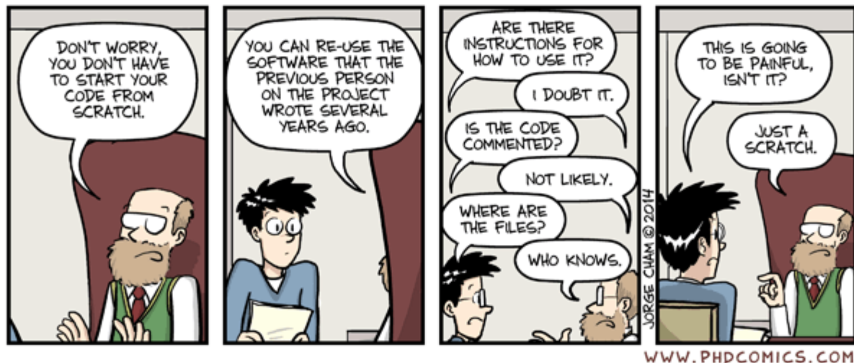
```
$ sudo apt-get install git
```

- if you want to follow the analysis example: some Python/R

Example analysis

- analysis example: tensorflow getting-started example (`https://www.tensorflow.org/tutorials/`)
- goal: classify 28-by-28 pixel images of the digits 0-9 from **MNIST** dataset of hand-drawn digits
- mainly built on python/tensorflow/keras, later combined with R
- just constructing a simple neural network model in **TensorFlow** and training it

Where are we?



- absolutely not restricted to academic projects ;)

What is reproducible research?

- small/medium analytic projects: end product often still a [pdf/html/docx] report
- **reproducible: same code, same data** $\rightsquigarrow$ same result
- **replicable: same code, new data** $\rightsquigarrow$ qualitatively same result
- replicability is hard and expensive, not our topic today, but ...
- ... reproducibility should be minimum standard!

Why is reproducibility so important?

1. increases **trust** in results:
   - (In the life sciences) amount of code required to produce results often longer than the actual paper
   - having the code available to reproduce the results in a paper will increase quality of peer review!
2. makes analyses **extensible**
3. increased **long-term efficiency**
   - often: person that has to reproduce your results will be you!
   - be gentle to you future self!
   - adopting a reproducible workflow can save you lots of work

Connection to software development?

- data analyses today is mostly software driven
- ‘customers’ usually not aware of complex software stack behind the reports
- (narrow) definition:
  analysis = software program turning data into report/figures
- many problems of software development apply:
  - testing (not covered today!)
  - agile development (versioning!)
  - documentation
  - dependency management/isolation
- tools can help with technical side of the issue, but:
  - not necessarily geared towards reproducible research
  - in some communities: not even known at all!

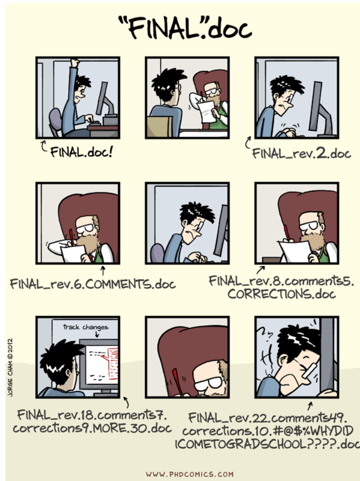
Some modest advice from a reproducibility-fetishist

- **reproducibility is a continuum**, small but simple improvements are good first steps!
- **cost/benefit** of measures must be taken into account!
- technical solutions can only complement: A fully containerized analysis without any documentation might be reproducible but is still practically useless for anyone not involved in the initial analysis

What we will look at today

1. What is reproducible research and why do we need it? ✓
2. Version control: keep track of changes to files over time, different variants of files, collaboration, (**git**, github.com)
3. Literate Programming: combine text and code, 'programming reports', (**jupyter**, **knitr**, **RMarkdown**, **pandoc**)
4. Build automation: Automate the 'build process' of your reports (**make**)
5. Containerization: Dependency management by packaging the entire computing system used for the analysis in an isolated container (**docker**, **singularity**)

We have all been there. . .



- never assume that something is finished and does not need to be revisited later

Why version control?

- analyses often initially exploratory: requirements and hypotheses evolve over time
- when multiple people are involved: diverging versions/variants
- version control allows to **keep track of changes** over time and between vararìants (branches)
- allows going back in time or developing different variants in parallel with the option of **merging** them together at a later point!

git (`https://git-scm.com/`)

- originally developed by **Linus Torvals** for managing the linux kernel development (2005)
- the name? most probably: pronouncible 3-letter combination not already in use by other unix commmand ...
- **free and open-source**
- widely used in industry and academia
- extremely powerful professional tool
- easy to get started with, **hard to master** (you can break things... completely)

git is not exactly beginner-friendly

Figure: xkcd git_2x

Why use git for research?

- git is designed with **distributed development** in mind (there is no central repository) $\rightsquigarrow$ perfect for academia!
- git **emphasizes 'branching'** (diverging versions from the main line of development), useful to try out new angles/features on an analysis
- acts as a **time-machine for your work**
- excellent community support + de-facto standard for web based services (github.com, gitlab.com)

git vs. github.com

- git is a command line tool
- github.com (or gitlab.com) are (commercial) code hosting platforms running git servers
- mostly free to use but beware of data protection laws when uploading data!
- github.com and the like make online collaboration extremely easy
- this workshop's materials are publicly available at `https://github.com/kkmann/reproducibleresearch`

git basics for today

- git quickly becomes complicated and deserves a workshop on its own!
- git survival package for today:

```
$ sudo apt-get install git

$ git clone xxx

$ cd xxx

$ git checkout jupyter
$ git checkout master
```

Version control

- probably one of the two most important tools for reproducible analyses (besides ‘make’, cf. later)
- git is the de-facto standard (SVN and Mercurial still used)
- git is the real stuff: very sophisticated, professional tool, steep learning curve, easy to break things
- git + ecosystem encourages collaboration
- Any form of professional version control encourages a clean and transparent workflow!
- good place to start learning are the tutorials at:
  - `https://try.github.io/`
  - `https://www.atlassian.com/git/tutorials`

Document your code!

Guido van Rossum

"Code is more often read than written."

Document your code

- one of the first things you (should) learn when programming: document your code!
- *“Code tells you how; Comments tell you why.”*, Jeff Atwood
  - good code should be fairly self-explaining
  - still need to document what you do and why!

Bad example

```
# set up sequential keras model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(), # input layer
  tf.keras.layers.Dense(512, activation=tf.nn.relu), # hidden layer
  tf.keras.layers.Dropout(0.2), # dropout layer
  tf.keras.layers.Dense(10, activation=tf.nn.softmax) # output layer
])
# compile the model, using adam optimizer and categorical crossentropy
# as loss function, monitor accuracy during training
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

- restating the obvious
- hard to maintain after changes
- what is this model for?

Better

```
# build neural network model in Keras for MNIST classification task
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

- don't state the obvious, be concise
- explain what and why, not how
- also: stick to code formatting guidelines!

Literate Programming

Donald Knuth, 1984

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming'."

Documenting data analysis code

- documentation even more important (explain the goal, choices)
- documentation often much longer than code $\rightsquigarrow$ **documentation-first approach**
- instead of embedding the comments in a code file, **embedd the code in a text document**!

1st candidate: Jupyter notebook

- `http://jupyter.org/`
- spin-off from IPython (around since 2001!)
- name composed of JUlia + PYThon + R: many languages (not just python) supported via kernels
- code is organized in ‘chunks’ (blocks)
- results are displayed right below corresponding code chunk
- interactive approach; code can be executed in arbitrary order
- code can be combined with markdown formatted text

Demo

Notebook formats and reproducible research

- great for early prototyping!
- limited formatting flexibility (cannot suppress unwanted chunks, no custom templating etc.)
- source file (.ipynb) not really human-readable
- encourage mistakes by sloppy execution order (code chunks can be executed in arbitrary order!)
- more in-depth critique: Joel Grus JupyterCon 2018 *“I don’t like notebooks”*

Enter markdown + pandoc

- markdown (.md files) :
  - `https://daringfireball.net/projects/markdown/`
  - John Gruber, 2004
  - extremely **simple markup language**
  - supports only most essential features (headings, lists, simple tables, hyperlinks, etc.)
  - **'least common denominator'** for pletora of different markup languages (html, LaTeX, . . .)
- pandoc
  - `https://pandoc.org/`
  - John MacFarlane, 2006
  - **converts between markup languages via markdown**, supports html, pdf, .ods, .docx, etc.
  - extensive theming possible via templates, very flexible
  - no support for literate programming (code chunks) out of the box!

RMarkdown + knitr + pandoc

1. **RMarkdown**: extends markdown to include code chunks
   `https://rmarkdown.rstudio.com/`
2. **knitr**: R package responsible for ‘knitting’ text, code and output in simple markdown file
   [https://yihui.name/knitr/](`https://yihui.name/knitr/`)
3. markdown file can then be rendered in almost arbitrary output formats (html, pdf, .ods, .docx) using pandoc

- knitr + RMarkdown support multiple languages (R, python, julia, SQL, bash, C++, Stan, etc.)

R Markdown basic structure

```
text

‘‘‘{[interpreter] [chunk-name] [, chunk options]}
[your code]
‘‘‘

more text
```

- look at this in more detail during next demo

RMarkdown + knitr + pandoc

- extremely **flexible** via custom pandoc templates
- RMarkdown still **human-readable source file**
- no notebook-like messed up order of execution (optionally available in RStudio though)
- clear ‘build process’ from .Rmd to .md via knitr and from .md to almost any output format
- allows combination of **multiple interpreters in one document**!
- excellent python + r interoperability via R package reticulate https://rstudio.github.io/reticulate/articles/introduction.html
- objects can be shared between R and python sessions in one document

Alternatives

- for even more control, use .Rnw (Sweave) files using LaTeXas markup language
- Sweave files can only be output as .pdf (or .ps)
- much more complex markup language (LaTeX)
- Pweave for python does the same thing

Demo

RMarkdown + knitr + pandoc

Build Automation

- by now, we have a single source file (.Rmd) for our analysis report
- **so far: built the report via RStudio's GUI**
- problem: not automatic, requires point-and-click user interaction
- problem: potential hidden stuff going on under the hood
- imagine big project with multiple interdependent reports, need to be processed in correct order!
- need to **completely automate build process**!

‘make’ reports reproducibly

- similar problem in software development: compile and link programs!
- tool of choice: *make*!
- software-dinosaur: around since 1976 (by Stuart Feldman)
- make executes ‘makefiles’ specifying recipies for how to ‘make’ files
- make keeps track of file dependecies and only rebuilds what is necessary - acts as cache!
- $\rightsquigarrow$ make is just as useful for automation of report builds

makefile structure

- a minimal makefile for out report:

```
report.pdf: mnist report.Rmd
    R -e "rmarkdown::render('report.Rmd', output_file = 'report.pdf')"
```

- **recipes** for files (report.pdf) with requirements (mnist folder, holds the data) and a bash command (knit the .Rmd file)
- **dependencies are monitored** for changes - report.pdf is only rebuilt when the content of the mnist folder changes or the RMarkdown source file
- dependency checking acts like **caching** during development
- with proper makefile: user just needs to call make in project folder - done.

Bonus: pandoc templates

- technically relatively easy ...
- ... but practically a bit tricky (especially for .ods and .docx outputs!)
- not really documented
- beyond the scope of this workshop
- best advice: look at respective pandoc default templates and go from there
  `https://github.com/jgm/pandoc-templates`
- can be tightly integrated with R
  `https://bookdown.org/yihui/rmarkdown/document-templates.html`

Demo

make

Wrap-up: make

- make might easily be *the* most important tool (and oldest) for reproducible research
- enables automation of the entire output generation
- can be used without literate programming to automate plot generation or non-output prerequisite operations
- essential for complex multi-layered projects (caching!)
- ‘makes’ the structure and sequence of the report generation transparent (what depends on what)

Are we there yet?

‘Reproducible’ vs. ‘portable’

- we have: fully automatic way to get from data to nice .pdf report (just call ‘make’)
- but: piled up huge stack of software dependencies along the way!
  1. base linux system with all its system libaries
  2. make
  3. R and some packages
  4. python and some packages
  5. pandoc
  6. LaTeX and some packages
  7. custom report template
- analysis might be reproducible (on my system) but **not portable** (to another system)

Dependency management

- Most programming languages have some sort of package manager (pip for python, built-in for R)
- reproducibility not necessarily primary design principle
- reproducibility tacked-on later (virtual environments, packrat + MRAN repository)
- **do not solve system-level dependencies**
- better than nothing but not really robust for complex analysis employing several different languages / software packages!

Fix #1: write specification manifest

- write a manifest with specification of the entire software, where to get it, and how to install it . . .
- nightmare to maintain up-to-date valid ‘protocol’
- error-prone (not really testable)
- future availability of required software and compatibility is hard to guarantee

Fix #2: virtual machines

- better: put everything in a virtual machine
- relatively easy and works fine
- but: not exactly the right concept: VMs are full blown systems capable of multiple task - we just need a minimal set-up to execute our analysis reliably
- VMs: large, ineffective, difficult to administrate
- lightweight alternative: *containerization*

Fix #3: Container

- disclaimer: I am far from being a container expert!
- good news: you don’t have to be either to use this stuff!
- technically wrong, but for our purposes: **container = lightweight VM**
- can be tuned to efficiency (only the stuff you need) or towards reusability
- **layer-wise construction** makes them effective to store
- no need to start from scratch! plenty of **base layers available for free**
- effectively provides a **portable computing environment** to execute our ‘make’ command in

Docker

- `https://www.docker.com/`
- very popular containerization software
- great community support
- easy to use (for our purposes)
- even works on Windows using the Windows Subsystem for Linux
- open source code
- comes with **free container hosting service** ‘dockerhub’ `https://hub.docker.com/`

Building a docker container

- building your own container is like cooking a curry
- you can start from scrach but you don't need to
- any publicly available container can be used as base layer
- *rocker project* maintains versioned images for R and Rstudio
  - *rocker/verse* container includes everything for using Rmarkdown + knitr + pandoc pipeline (incl. LaTeX)
    `https://hub.docker.com/r/rocker/verse/`
  - `https://www.rocker-project.org/`

The dockerfile

- basic example of a dockerfile building an image based on rocker/verse for a specific R version

```
FROM rocker/verse:3.5.1

MAINTAINER Kevin Kunzmann kevin.kunzmann@mrc-bsu.cam.ac.uk

RUN sudo apt-get update

RUN sudo apt-get install -y python3-pip python3-dev python3-tk
RUN sudo pip3 install -U pip
RUN sudo pip3 install numpy==1.14.3 matplotlib==2.2.2 tensorflow==1.8.0
```

Building and distributing a container image

```
docker build -t [imagename] .
```

```
docker push [imagename]
```

- that's it!
- docker container image for this tutorial available at: `https://hub.docker.com/r/kkmann/reproducibleresearchtutorial`

Running 'make' inside a container

```
docker pull kkmann/reproducibleresearchtutorial
docker run --name=dsaa2018 --rm -d -p 8787:8787;
  -e PASSWORD=dsaa2018;
  -v ${PWD}:/home/rstudio/DSAA;
  kkmann/reproducibleresearchtutorial
docker exec -it -w /home/rstudio/DSAA dsaa2018 make all
docker kill dsaa2018
```

- container images can also be identified by sha256 hash in case of updates (cf. demo)
- '-v' mounts host volumes in the container (keep data / analysis scource code out of container!)

Demo

Docker is not ideal for RR

- docker was never intended to be used for reproducibility!
- **requires root access**!
  - fine on your local machine, but some things need to be run on server/cloud
  - ideally, analysis runs in a cloud environment $\rightsquigarrow$ portability
- ideal container system should work with cloud / HPC environments on user level (no root!)

Solution: Singularity

- `https://www.sylabs.io/`
- new kid on the block (stable release 2.5.2: 2018)
- free, open-source, cross-platform
- designed for HPC (**no root access required!**) and reproducibility
- fully **compatible with docker!**
  - can pull and run docker containers out of the box!
  - similar command structure

Demo

Wrap-up: ideal structure

- git repository with analysis code
  - literate programming reports (.Rmd) + any required code files
  - top level make file with target ‘all’ executing all required steps in correct order
  - bash script to run make inside container
- container image with entire computing environment

- process to reproduce anywhere:
  1. clone git repository (at specified release tag!)
  2. run ‘make all’ inside the container (ideally via provided bash script)
- $\rightsquigarrow$ minimal dependencies: git, singularity-container or docker

Put it to the test

1. install singularity, Ubuntu package sources outdated, stable 2.6.0 must be installed manually

```
$ git clone https://github.com/sylabs/singularity.git
$ cd singularity
$ git fetch --all
$ git checkout 2.6.0
$ ./autogen.sh
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
$ cd ..
```

2. clone analysis code

```
$ git clone https://github.com/kkmann/reproducibleresearch
$ cd reproducibleresearch
```

3. execute the run script (needs to download image the first time!)

```
$ chmod u+x run_singularity.sh
$ ./run_singularity.sh
```