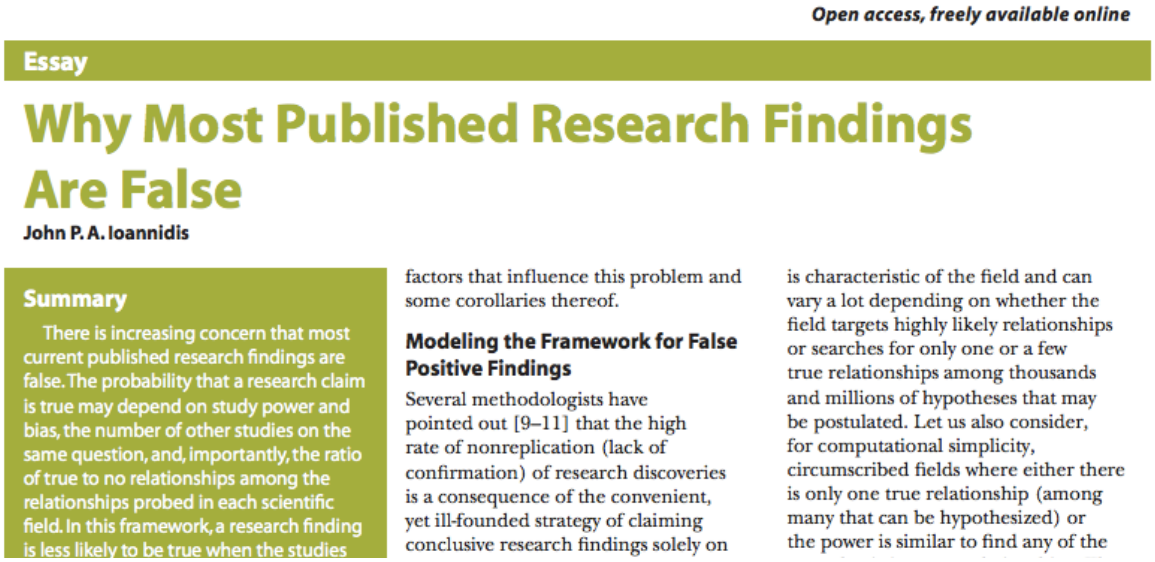
Software carpentry and reproducible data analysis

Shaun Mahony

What do we mean by a reproducibility crisis?



Open access, freely available online

Essay

Why Most Published Research Findings Are False

John P. A. Ioannidis

Summary

There is increasing concern that most current published research findings are false. The probability that a research claim is true may depend on study power and bias, the number of other studies on the same question, and, importantly, the ratio of true to no relationships among the relationships probed in each scientific field. In this framework, a research finding is less likely to be true when the studies

factors that influence this problem and some corollaries thereof.

Modeling the Framework for False Positive Findings

Several methodologists have pointed out [9–11] that the high rate of nonreplication (lack of confirmation) of research discoveries is a consequence of the convenient, yet ill-founded strategy of claiming conclusive research findings solely on

is characteristic of the field and can vary a lot depending on whether the field targets highly likely relationships or searches for only one or a few true relationships among thousands and millions of hypotheses that may be postulated. Let us also consider, for computational simplicity, circumscribed fields where either there is only one true relationship (among many that can be hypothesized) or the power is similar to find any of the

Ioannidis, et al. PLoS Medicine 2005, 2(8):e124

Reproducibility of a given scientific conclusion across different studies?

or

Reproducibility of <u>your</u> methods and results?

Questions researchers should ask themselves

- If this computer disappears, will my work be affected?
- Can I easily run this analysis on 1,000 more datasets?
- Will I be able to easily re-run this analysis in 5 years?
- Will someone else be able to easily run and understand this analysis without talking to me?

What makes research reproducible?

Knowing **what has been done** and **how has it been done**.

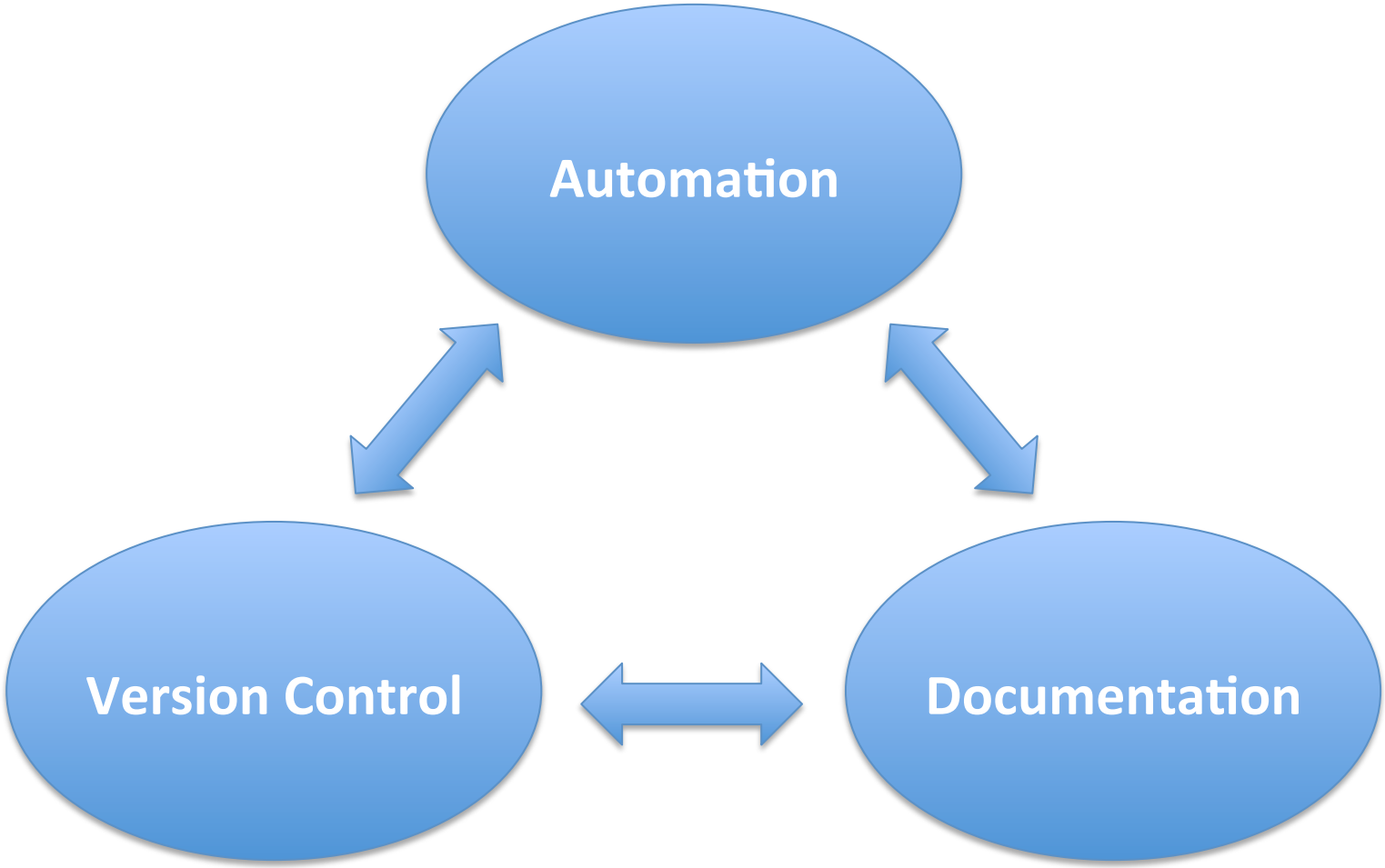
- Enumerate the “set pieces”: data, code, parameters, software versions.
- Explain the rationale behind making certain choices
- Ensure consistency in space and time: from computer to computer, or this year to next.

(Istvan Albert)

Decide to make your research reproducible

- **We do it because it helps us.**
- It shouldn't be an extra burden.
- We get better at those things that we practice
- If we do it all the time, no extra effort is needed.

(Istvan Albert)


Automation
Version Control
Documentation

Overview of today's session

Time	Subject
9:00 – 9:30	Best practices for scientific computing
9:30 – 10:30	Shell scripting basics
10:30 – 10:50	BREAK
10:50 – 12:00	Running scripts on clusters
12:00 – 1:30	LUNCH
1:30 – 3:00	Markdown & Git
3:00 – 3:20	BREAK
3:20 – 5:00	Git (continued) & Discussion

Required items

- Software carpentry website in browser
  - https://software-carpentry.org/lessons
- Command-line interface
- Git
- pandoc
- Account on Github
- Account on ACI-ICS

BEST PRACTICES FOR SCIENTIFIC COMPUTING

Wilson, *et al.* PLoS Biology, 2014 12(1):e1001745

Rule 1: Write Programs for People, not Computers

1.1: Keep it simple.

1.2: Make names consistent, distinctive, and meaningful.

1.3: Make code style and formatting consistent.

Rule 2: Let the Computer Do the Work

2.1: Make the computer repeat tasks.

2.2: Save recent commands in a file for re-use.

2.3: Use a build tool to automate workflows.

Rule 3: Make Incremental Changes

3.1: Small steps with frequent feedback

3.2: Use a version control system.

3.3: Version control EVERYTHING

(…except big data)

Rule 4: Don't Repeat Yourself (or Others)

4.1: Every piece of data must have a single authoritative representation

4.2: Modularize code rather than copying and pasting.

4.3: Re-use code instead of rewriting it.

Rule 5: Plan for Mistakes

5.1: Add assertions to programs to check their operation.

5.2: Use an off-the-shelf unit testing library.

5.3: Turn bugs into test cases.

5.4: Use a symbolic debugger.

Rule 6: Optimize Software Only After It Works Correctly

6.1: Use a profiler to identify bottlenecks.

6.2: Write code in the highest-level language possible.

Rule 7: Document Design and Purpose, Not Mechanics

7.1: Document interfaces and reasons not implementations.

7.2: Refactor code in preference to explaining how it works.

7.3: Embed the documentation for a piece of software in that software.

Rule 8: Collaborate

8.1: Use code reviews.

8.2 Use pair programming

8.3: Use an issue tracking tool.

SHELL SCRIPTING

http://swcarpentry.github.io/shell-novice

Why use a command-line interface?

- Run analyses **iteratively**.
- Run analyses **reproducibly**.
- Build powerful pipelines (combining tools).
- Use specialized analysis software.
- Access high-performance computing systems.

Basic shell commands

- **pwd** : display working directory
- **ls** : list files
- **cd** : change directory
- **cp** : copy file
- **mv** : move file
- **rm** : remove file
- **mkdir** : make directory
- **cat** : display contents of input
- **head** : display the first few lines of its input
- **tail** : display the last few lines of its input
- **sort** : sort its inputs
- **wc** : count lines, words, and characters in its inputs
- ***** : matches zero or more characters in a filename
- **>** : redirects a command's output to a file
- **|** : pipeline output of the first command as input to the second

Loops

- for filename in *.dat; do
  echo $filename;
  done

- for i in $(seq 1 0.5 10); do
  echo $i;
  done

- while read -r line; do
  echo “$line"
  done < "$filename"

- i="0”
  while [$i -lt 4]; do
  echo $i;
  i=$[$i+1]
  done

Numeric comparison

- -eq is equal to
  if ["$a" -eq "$b"]

- -ne is not equal to
- -gt is greater than
- -ge is greater than or equal to
- -lt is less than
- -le is less than or equal to

- < is less than (within double parentheses)
  (("$a" < "$b"))
- <= is less than or equal to (within double parentheses)
- > is greater than (within double parentheses)
- >= is greater than or equal to (within double parentheses)

String comparison

- = is equal to

  if ["$a" = "$b"]
- == is equal to
- != is not equal to
- < is less than, in ASCII alphabetical order
- > is greater than, in ASCII alphabetical order
- -z string is null, that is, has zero length

Named argument passing example

```
#!/bin/bash

usage() { echo "Usage: $0 [-s <45|90>] [-p <string>]"; exit 0;}

while getopts ":s:p:" o; do
    case "${o}" in
        s)
            s=${OPTARG}
            ((s == 45 || s == 90)) || usage
            ;;
        p)
            p=${OPTARG}
            ;;
        *)
            usage
            ;;
    esac
done
shift "$((OPTIND-1))"

if [ -z "${s}" ] || [ -z "${p}" ]; then
    usage
fi
echo "s = ${s}"
echo "p = ${p}"
```

Using the aci-b cluster

#Logging in

ssh –X <username>@aci-b.aci.ics.psu.edu

#Submitting jobs to the open queue

qsub –A open -l nodes=1:ppn=1 –l walltime=0:05:00 script.sh

#Checking job status

qstat

#Deleting a job

qdel <job id>

#Available software modules

module avail

#Load software modules

module load r/3.3

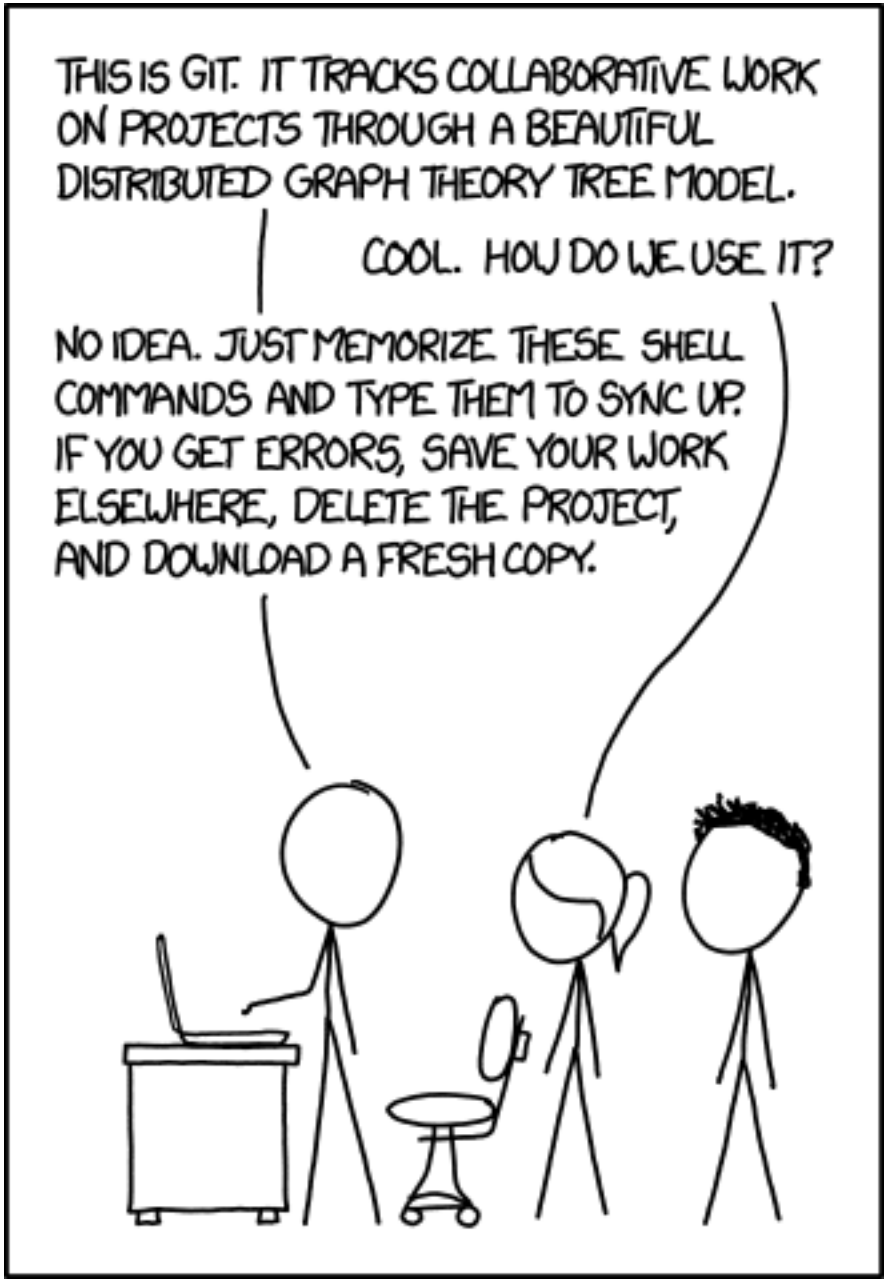
Example cluster script header

```
#PBS -l walltime=1:00:00
#PBS -l nodes=1:ppn=1
#PBS -j oe
#PBS -A open
#PBS -l mem=1gb

module load r/3.3
module load python/2.7.8
```

VERSION CONTROL WITH GIT

http://swcarpentry.github.io/git-novice


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Git commands

$ git init [project-name]

Creates a new local repository with the specified name

$ git clone [url]

Downloads a project and its entire version history

$ git status

Lists all new or modified files to be commited

$ git add [file]

Snapshots the file in preparation for versioning

$ git reset [file]

Unstages the file, but preserve its contents

$ git diff

Shows file differences not yet staged

$ git diff –staged

Shows file differences between staging and the last file version

$ git commit -m ”[message]”

Records file snapshots permanently in version history

$ git push [alias] [branch]

Uploads all local branch commits to GitHub

$ git pull

Downloads bookmark history and incorporates changes

https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

DISCUSSION & WRAP-UP

Discussion points

- What did we cover today that was new to you?
- How can we put all the pieces together?
- How will you put all of this into practice?