Scientific Communication, Collaboration and Design in Quantitative Research Using R

Daniel Hammarström

2025-03-13

Table of contents

Preface

> **Note**
>
> These course notes are under development. They will serve as the basis of the course *Scientific Communication, Collaboration and Design in Quantitative Research Using R* given as an elective in the PhD program Health and Welfare at the University of Inland Norway.

The purpose of this collection of workshops is to provide basic skills and understanding of scientific visualization, collaboration, and thinking. These concepts are often not taught explicitly in postgraduate courses. Instead, they are assumed to be known and understood or somehow magically infused in students through their respective scientific environments. Instead of relying on magic, professional skills should be taught and developed continuously, explicitly, and systematically.

The first workshop deals with data visualization which is a fundamental skill for any scientist. Data visualization, like writing, serves multiple purposes in day-to-day scientific practice. Making graphs before data collection can help the researcher better formulate the scientific problem or explain core concepts for oneself, committees or funding agencies. Visualizations will also be important for exploratory data analysis, for discovering problematic data formatting or important patterns related to the scientific problem. Lastly, visualizations will aid in communicating scientific results to other researchers and the public.

Two workshops are devoted to collaboration in data-intensive science. The first workshop deals with the basics of version control and the second workshop deals with structuring collected data into data packages for analysis and sharing. Version control is a way to keep track of changes in a set of files. It also provides opportunities for effective collaboration in complex projects. Using the R package system for data management and sharing is a way to ensure that data is well documented and downstream analyses are reproducible.

The last workshop deals with how we can develop scientific thinking using scientific models and data simulations. We will use graphical models to draw out assumptions and explore simulations as a way to test the robustness of our models and plan experiments.

1 Data visualization

The most apparent purpose of data visualizations in scientific contexts is to convey information not suited for text or tables but for graphical displays like comparisons between categories, relationships among variables, or trends over time (Gelman, Pasarica, and Dodhia 2002; Tufte 2001). We often think of the scientific graph as the end product of our labor. However, data visualization can be an effective tool for thinking about scientific problems and performing exploratory data analysis before preparing your manuscript. These steps will likely take up more time than polishing *Figure 1* of your upcoming paper.

Data visualization is thus a skill, like writing, with several purposes. You may write notes for yourself and write to communicate with others. Notes come in many forms, just as messages you write to others. Similarly, data visualizations can have different purposes, such as

- Checking that your data is what could be expected
- Diagnosing statistical models
- Exploring a relationship between many variables
- Showing that your measurements are reliable and valid
- Conveying the key message from your study

Traditionally, and in many software implementations, data visualization is a matter of choosing from a menu of charts such as box plots, scatter plots, or line graphs. These can be very effective. However, you would not restrict your writing to pre-made templates. As a professional scientist in a quantitative genre, you need a toolbox for data visualizations that do not limit you. By using a *grammar of graphics* you can be more creative in designing your visualizations.

1.0.1 Glamorous graphics

In a talk at RStudio conference in 2020, Will Chase presented principles related to glamorous graphics. In contrast to Tufte's principles (discussed above), glamorous graphics suggest adding ink to increase the attractiveness of graphics. (This might be a topic for a later workshop).

2 Version control and collaborative scientific coding

Reproducibility and transparency are aspects of scientific research practices that have great potential to affect the quality of scientific results. Quality in the sense that results, to a more considerable degree, will be available for scrutiny by researchers, funding agencies, and the public. As scientists, we have a moral obligation to be transparent and strive towards reproducibility, and to have these as goals in their own right will likely make us better scientists. Version control software allows researchers to automate the process of keeping a record of changes in a project. In creating this record, we also create transparency and allow for reproducibility. As a scientist, changing our mindset to focus on researching and communicating with transparency and reproducibility rather than one end product, such a scientific paper may change our thinking about the scientific process. Here, we will introduce version control as a tool in collaborative scientific writing.

Collaboration in complex projects is messy. Multiple files exist in multiple versions; changes are made in parallel with little to no control over what is lost or gained. It is likely a frequent experience that most scientific collaborations could use a more formal structure for working with data analysis or writing. Such formal structures are, however, difficult to establish without a common point of departure. This workshop aims to create a common understanding of version control, collaboration, transparency, and reproducibility.

2.1 Introduction to git and GitHub

Git is a version control system that you install on your local machine. It helps you create a record of changes that you make to files in a specific repository where git is initialized. The git software stores earlier versions of your repository content in a hidden directory called `.git`.

Git works great for working with plain text files, such as `.txt`, `.md`, `.qmd`, `.csv`. With such files git can provide you with an overview of changes or diffs that has occurred between two versions of the file. Other file types such as `.docx` or `.xlsx` are also tracked by git but changes are not human readable in the git system.

Git can be set up to communicate with an online repository. There are several alternatives for online hosting of version controlled repositories but GitHub seems to be a popular alternative in the (data) science community. GitHub offers online hosting and tools for collaboration.

To enable version control and collaboration you must therefore have git installed and an active GitHub account.

2.1.1 Accounts and git-to-GitHub integration

Git can be installed from https://git-scm.com. Git requires no additional account information or software. To allow for RStudio to act as a git client, you need to tell RStudio where your installation of git is located. This is done under *Tools* > *Global options* > *Git/SVN*.

Your local version history maintained by git can be controlled from your terminal window. However, to connect to the online services of github you will need to set git up with your github account.

Github has recently made changes to its authentication protocol, and seems to be in the process of making additional changes. GitHub CLI provides a command line interface to GitHub that makes it easy to authenticate. Install GitHub CLI here and run `gh auth login` in your terminal to set things up.

2.1.2 Local git edits

As already mentioned, git stores a record of changes to files in a given repository. A repository is a directory in which you have initialized git. We can do this in any directory using the command line by typing

```
git init
```

in our terminal. If you initialize a git (version control) project using RStudio it will run this command for you. You may check that you have initialized the version control system by typing

```
git status
```

You make decisions on what files to add to your version history and when to do so. A file, let us say `file-a.txt` is added to the staging area with the command

```
git add file-a.txt
```

The staging area is where files are kept until you commit a snapshot of them into the version history. We could also stage all files that have changes made to them by typing

```
git add -A
```

The next step is to commit changes. A commit is a snapshot of the repository or files at a given moment. We perform a commit when after staging files type

```
git commit -m "my commit message"
```

The `-m "my commit message"` adds a commit short commit message to the commit. This should be a description of the changes made to the repository. A longer commit message can be added by only initializing a commit using `git commit`. This command will bring up a dialog in your terminal making it possible to write a short message in the first line. After leaving one blank line a longer commit message can be added.

When a file has been commited to the version history it is marked as *unmodified* until you make changes to the file. Modified files can again be staged (e.g. `git add <file>`) and committed.

Files that were are tracked by mistake can be un-tracked without being removed using

```
git rm --cached <file>
```

This leaves the file in your local folder but removes it from the repository. Using `git rm <file>` removes the file from disc.

A local git "edit cycle" is shown in Figure 2.1.

```
library(ggplot2); library(ggtext)

ggplot(data.frame(x = c(0,1), y = c(0,1)), aes(x, y)) +

  scale_y_continuous(limits = c(0.3, 1)) +

  # Workspace
  geom_rect(aes(xmin = 0.01,
            xmax = 0.6,
            ymin = 0.35,
            ymax = 1),
            fill = "steelblue",
            alpha = 0.2) +

  geom_rect(aes(xmin = 0.02,
            xmax = 0.3,
            ymin = 0.35,
```

```
           ymax = 0.92),
           fill = "steelblue",
           alpha = 0.2) +

geom_rect(aes(xmin = 0.31,
           xmax = 0.58,
           ymin = 0.35,
           ymax = 0.92),
           fill = "steelblue",
           alpha = 0.2) +

  annotate("richtext",
           x = c(0.02, 0.32),
           hjust = 0,
           y = 0.89,
           label = c("*Untracked*", "*Tracked*"),
           fill = NA, label.color = NA) +


annotate("richtext",
           x = 0.02,
           y = 0.97,
           label = "**Workspace**",
           hjust = 0,
           size = 8,
           fill = NA, label.color = NA, # remove background and outline
          label.padding = grid::unit(rep(0, 4), "pt")) + # remove padding) +

# Stage area
geom_rect(aes(xmin = 0.61,
           xmax = 1,
           ymin = 0.35,
           ymax = 1),
           fill = "purple",
           alpha = 0.2) +



  annotate("richtext",
           x = 0.62,
           y = 0.97,
           label = "**Staging area**",
```

```
          hjust = 0,
          size = 8,
          fill = NA, label.color = NA, # remove background and outline
         label.padding = grid::unit(rep(0, 4), "pt")) + # remove padding) +



## Labels and arrows


annotate("segment", y = c(0.85, 0.85),
                 yend = c(0.85, 0.82),
                 x = c(0.1, 0.9),
                 xend = c(0.9, 0.9),
                 arrow = arrow(length = unit(c(0, 2.5), "mm"), type = "closed")) +



geom_label(aes(x = 0.62, y = 0.85 , label = "<file>"),
           hjust = 0) +


geom_label(aes(x = 0.02, y = 0.85 , label = "Add file `git add <file>`"),
           hjust = 0) +


 annotate("segment",
          y = c(0.8,0.75),
          yend = c(0.75, 0.75),
          x = c(0.9,0.9),
          xend = c(0.9, 0.48),
          arrow = arrow(length = unit(c(0, 2.5), "mm"), type = "closed")) +



     annotate("segment",
          y = c(0.75, 0.7),
          yend = c(0.7, 0.7),
          x = c(0.35, 0.35),
          xend = c(0.35, 0.4),
          arrow = arrow(length = unit(c(0, 2.5), "mm"), type = "closed")) +
```

```
geom_label(aes(x = 0.65, y = 0.8 , label = "Commit file `git commit -m 'msg'`"),
           hjust = 0) +


  geom_label(aes(x = 0.31, y = 0.75 , label = "Edit unmodified     "),
           hjust = 0) +




   annotate("segment",
         y = c(0.7, 0.7),
         yend = c(0.7, 0.67),
         x = c(0.5, 0.9),
         xend = c(0.9, 0.9),
         arrow = arrow(length = unit(c(0, 2.5), "mm"), type = "closed")) +


    geom_label(aes(x = 0.8, y = 0.7 , label = "<file>"),
           hjust = 0) +



    geom_label(aes(x = 0.40, y = 0.7 , label = "Stage modified `git add <file>`"),
           hjust = 0) +



    annotate("segment",
         y = c(0.65,0.65),
         yend = c(0.65, 0.62),
         x = c(0.8, 0.35),
         xend = c(0.35, 0.35),
         arrow = arrow(length = unit(c(0, 2.5), "mm"), type = "closed")) +

  geom_label(aes(x = 0.65, y = 0.65 , label = "Commit file `git commit -m 'msg'`"),
           hjust = 0) +
```

```
## Possible routs from unmodified

## Modify and commit
## Remove (untrack)
       annotate("segment",
          y = c(0.6, 0.6),
          yend = c(0.6, 0.54),
          x = c(0.4, 0.4),
          xend = c(0.70, 0.4),
          lty = 2,
          arrow = arrow(length = unit(c(2.5, 2.5), "mm"), type = "closed")) +
## File in staging area
       geom_label(aes(x = 0.75, y = 0.6 , label = "<file>"),
           hjust = 0.5) +



## Untrack file
     annotate("segment",
          y = c(0.5),
          yend = c(0.5),
          x = c(0.4),
          xend = c(0.1),
          lty = 2,
          arrow = arrow(length = unit(c(2.5), "mm"), type = "closed")) +

    geom_label(aes(x = 0.31, y = 0.60 , label = "Unmodified    "),
           hjust = 0) +




  ## Remove file (delete)
     annotate("segment",
          y = c(0.5),
          yend = c(0.33),
          x = c(0.4),
          xend = c(0.4),
          lty = 2,
          arrow = arrow(length = unit(c(2.5), "mm"), type = "closed")) +
```

```
    annotate("label",
           x = 0.31, y = 0.5 ,
           label = "Untrack file\n`git rm --cached <file>`",

           hjust = 0) +

## File in trash
      geom_label(aes(x = 0.4, y = 0.30 , label = "<file>"),
           hjust = 0.5) +


## File untracked
   geom_label(aes(x = 0.06, y = 0.5 , label = "<file>"),
           hjust = 0.5) +


  geom_label(aes(x = 0.31, y = 0.40 , label = "Delete file from disc\n`git rm <file>`",
           hjust = 0)) +



theme_void()
```

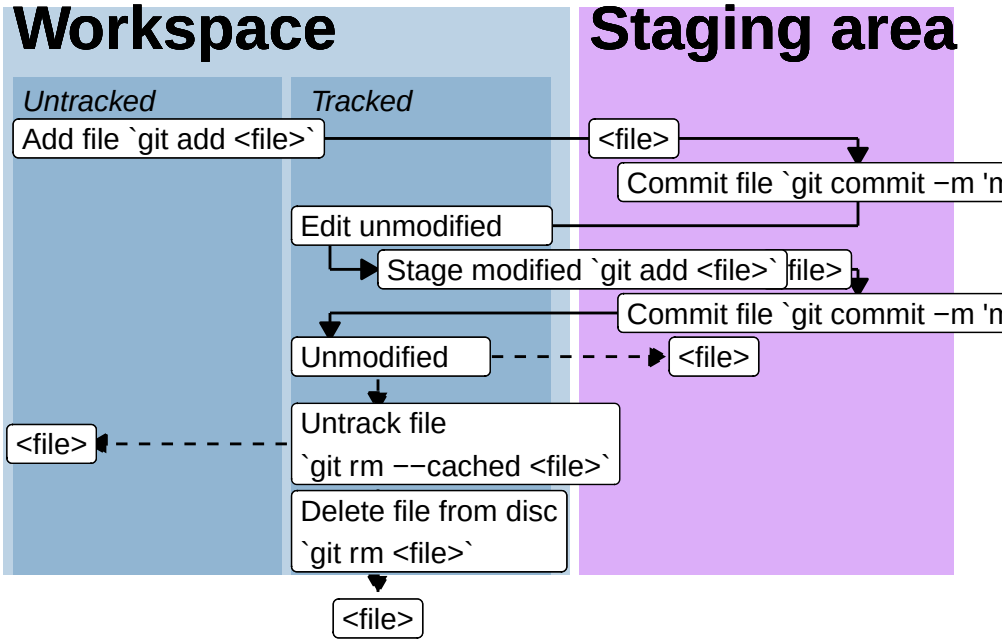

Figure 2.1: A cycle of local git edits. A files is added to the staging area by `git add <file>` and comitted to the *local repository* by `git commit`. The file is edited and is marked as modified after which it is staged (`git add <file>`). Again, staged modifications are commited and the file is once again marked as unmodified until edited. Files may be untracked from the repository or deleted from disc using `git rm` with or without the `--cached` option.

2.1.3 Local and online git usage

Your local git folder can be hocked up to a online repository. The online repository, or remote, is the repository in which you collaborate with others. The online repository can be set up as a starting point for your project or created from the command line in an already initialized git repository using GitHub CLI.

2.1.3.1 Starting with GitHub

After logging on to GitHub, select *New*, or go to github.com/new. This will bring you to a schema that will create your repository. You need to give it a name, describe it, decide if you want to initialize with a readme file (you want to do that) and choose a license. Once all this is done you will be able to copy the HTTPS address that you find under *Code* in your repository in GitHub. The HTTPS address is then entered in the *New Project Dialog* in RStudio after selecting *Version Control* under the different project types.

You have now initialized a remote repository and then cloned it to your computer as a local copy. You may now make changes to the project and add/commit these to you local version history.

2.1.3.2 Starting with a RStudio project

When starting up a new RStudio project without version control you have the opportunity to initialize git in the dialog box. If you decide not to do this, the next step should be to write `git init` in your terminal to initialize in your current project folder.

We can now create a remote repository using GitHub CLI with the command `gh repo create`. If this command is used without any subcommand you will be guided through the process on the command line. If you have your project running and wants to use it to create a repository on GitHub you should select *"Push an existing local directory to GitHub"*. You will be asked:

- The path to your local repository (defaults to `.`, the current directory)
- The name of the repository (defaults to your local folder name)
- If the repository should be private or public.
- A description of the repository (A short description of what the repository do/contain)
- If a remote should be added (Yes)
- What the remote should be called (Default to origin)

If successful, you will get a message telling you that the remote has been added. Your GitHub profile will now have a repository acting as the remote for your local repository.

2.1.3.3 Local to remote workflows

The online remote repository may have several purposes in the context of writing a scientific paper. It will act as the repository that you and any collaborators work together on. It can be used for showcasing code and computations and thereby supporting your paper. It can also act as a backup and reference for yourself.

The basic workflow (Figure 2.2) for updating and downloading from the remote repository can include `git push` to push all local commits to the remote. This will update the remote with any changes you have committed to the version history. When the remote has been changed you will be able to download these in two ways. `git pull` downloads all changes and directly overwrite your local files, updating them to their latest versions. This also include removal of files and addition of new files. If you want to have more control you can `git fetch` changes which updates the local repository, but not your working directory. Using `git merge` you can merge downloaded versions with your working directory. `git fetch` and `git merge` can be a way to continue working on files but updating your repository with the latest changes.

```
ggplot(data.frame(x = c(0,1), y = c(0,1)), aes(x, y)) +

  scale_y_continuous(limits = c(0.3, 1)) +

  # Local working directory / Repository
  geom_rect(aes(xmin = 0.01,
            xmax = 0.6,
            ymin = 0.35,
            ymax = 1),
            fill = "steelblue",
            alpha = 0.2) +

    geom_rect(aes(xmin = 0.02,
            xmax = 0.3,
            ymin = 0.35,
            ymax = 0.92),
            fill = "steelblue",
            alpha = 0.2) +

  geom_rect(aes(xmin = 0.31,
            xmax = 0.58,
            ymin = 0.35,
            ymax = 0.92),
            fill = "steelblue",
            alpha = 0.2) +

  # Remote repository
  geom_rect(aes(xmin = 0.7,
            xmax = 0.95,
            ymin = 0.35,
            ymax = 1),
            fill = "steelblue",
            alpha = 0.2) +


    annotate("richtext",
           x = c(0.02, 0.32),
           hjust = 0,
           y = 0.89,
           label = c("*Working directory*", "*Local repository*"),
           fill = NA, label.color = NA) +
```

```
annotate("richtext",
         x = c(0.02, 0.7),
         y = c(0.97, 0.97),
         label = c("**Workspace**","**Remote**"),
         hjust = 0,
         size = 8,
         fill = NA, label.color = NA) + # remove padding)


    # Git add, commit
# git push
# git pull
# git fetch / merge
     annotate("segment",
          y = c(0.8, 0.7, 0.6, 0.5, 0.5),
          yend = c(0.8, 0.7, 0.6, 0.5, 0.5),
          x = c(0.1, 0.4, 0.8, 0.8, 0.4),
          xend = c(0.5, 0.8, 0.12, 0.5, 0.12),
          arrow = arrow(length = unit(c(2.5), "mm"), type = "closed")) +

    annotate("label",
           x = c(0.1, 0.4, 0.8, 0.8, 0.35),
           y = c(0.8 ,0.7, 0.6, 0.5, 0.5),
           label = c("git add <file>\n git commit -m 'msg'",
                     "git push",
                     "git pull",
                     "git fetch",
                     "git merge"),

           hjust = 0) +

theme_void()
```

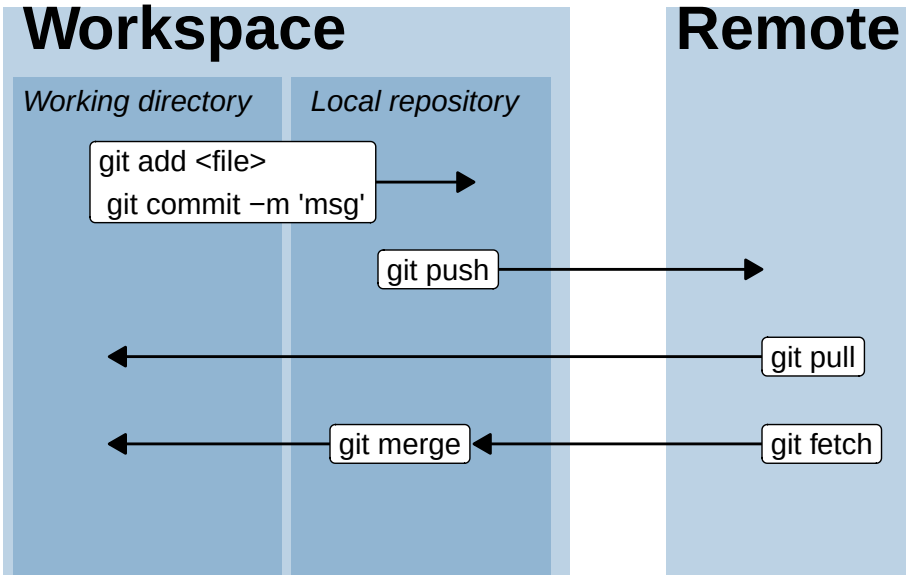

Figure 2.2: Local git commits can be pushed to a remote repository, this updates the remote with your local changes. If changes are made to the remote these can be pulled to the local reposotory and working directory. Git pull overwrites your local files in the working directory at the same time as updating the local version history. Git fetch downloads changes to your local repository without merging files. Git merge attempts to merge files in the working directory with the most recent changes from the remote repository.

2.1.4 Branches and pull requests

Sometimes changes to a repository can be expected to get big. Multiple files needs updating affecting multiple aspects of the repository. Instead of doing these as incremental steps in the main repository we could create a new branch. A branch starts at a specific state of the repository and adds new history without affecting the main branch. When changes have been made to the branch they can be incorporated into the main branch using merge or as a pull request.

2.1.5 Forks and pull requests

Forks are copies of an online repository from one user to another user where the receiving user gets the privileges to make changes in the repository.

2.1.6 Issues

Issues are an feature of GitHub which allows for creating notes on potential improvements, bugs etc. They can be accessed and edited using GitHub CLI using `gh issue` or in the GitHub web interface.

2.1.7 GitHub pages

GitHub pages is a service that allow for web hosting from your github repository.

2.2 A workflow for a scientific paper

A scientific paper (or a thesis) is reproducible if the data, code and documentation is available.

2.2.1 The repository

Organizing the repository is important for collaboration and communication

2.2.1.1 README

2.2.1.2 Folder structure

2.2.1.2.1 `data`

2.2.1.3 `R`

2.2.1.4 `derived-data`

2.2.1.5 `figures`

2.2.1.6 `docs`

2.2.1.7 `resources`

2.2.2 Structuring code in different scripts

2.3 Authoring quarto documents

Quarto

2.3.1 Code chunks

2.3.1.1 A style guide for scientific coding

We adhere to the tidyverse style guide. The following is a summary of the most important points with some modifications.

2.4 Publishing

3 R packages for data management and sharing

A basic principle in reproducible research is that the data and code used to generate results should be made available. Ideally, each component of a reproducible report (paper, thesis or similar) should be documented and organized to allow for independent execution of code that supports conclusions (Peng 2011). Publishers and journals have started to highlight reproducible scientific work, sometimes under the term open research[1]. Similarly, funding agencies are also taking steps to support or even demand open research practices, including reproducible research. Even though principles are well described and researchers have many incentives to publish reproducible open research, the fail to do so. Reproducible research is still not common practice. There are several reasons why this is the case, reasons such as lack of skills and knowledge on how to do it, insecurity or embarrassment in sharing behind-the-scenes work or fear of inappropriate use (Gomes et al. 2022). Such subjective concerns can be effectively addressed with a small to medium investment in time to learn skills and available solutions (Gomes et al. 2022), but also, more importantly by using reproducible practices throughout your workflow.

Sharing code and data in an efficient way may be difficult if you do not rely on software and workflows designed for this purpose. The R ecosystem has many advantages and an obvious one in this context is the possibility to formally combine data, code and documentation in a portable unit called a package. An R package can be shared trough CRAN or code repository systems such as GitHub. Alternatively, a package may be compressed and shared as a single file in a data repository such as DataverseNO. From a data-sharing perspective, an R package may include initial steps in data cleaning from raw unprocessed data to data used in statistical analyses. Such initial steps are part of the package source and may be documented as part of the package. Complex projects often involves several un-matched data sets as some variables are measured across multiple time-points others at a single time-point but across several domains or items. In such case, a single combined tabular data set is not possible to curate without loosing information or add complexity. In an R package you could combine several types of data sets that are possible to combine using analytic code for specific purposes. Additionally, "helper functions" can be included in the package to aid users to reproduce analyses or prepare data for novel analyses.

[1] Wiley marks publications with "open research badges" to indicate shared data, pre-registration and open material. The journal Biostatistics was an early adopter of marking papers with code, data and reproducible marks (Peng 2011).

3.1 R package basics

An R package is a structured

3.1.1 Incorporating data into R packages

3.1.2 Documenting data

3.2 Sharing packages

4 Scientific models and data simulations

Navarro argues that the theory comes before the statistical model (10.1177/1745691620974769). Theoretical formalism helps in defining study design and data analysis.

A theory can be simplified to allow for formalism…

A theory has consequences that can be tested using data and statistical models. The statistical model is a consequence of the theoretical model and …

Cox: The relation between theory and application in statistics

References

Gelman, Andrew, Cristian Pasarica, and Rahul Dodhia. 2002. "Let's Practice What We Preach." *The American Statistician* 56 (2): 121–30. https://doi.org/10.1198/000313002317572790.

Gomes, Dylan G. E., Patrice Pottier, Robert Crystal-Ornelas, Emma J. Hudgins, Vivienne Foroughirad, Luna L. Sánchez-Reyes, Rachel Turba, et al. 2022. "Why Don't We Share Data and Code? Perceived Barriers and Benefits to Public Archiving Practices." *Proceedings of the Royal Society B: Biological Sciences* 289 (1987): 20221113. https://doi.org/10.1098/rspb.2022.1113.

Peng, R. D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27. https://doi.org/10.1126/science.1213847.

Tufte, Edward R. 2001. *The Visual Display of Quantitative Information*. 2. edition. Cheshire, Conn.: Graphics Press.