



# Intro to Programming with R for Political Scientists

## Session 1: RStudio and Version Control

Markus Freitag

Geschwister Scholl Institute of Political Science, LMU



July 12, 2021

# Intro

## Why this course?

- Intro to R with a focus on programming ( $\leftrightarrow$  standard pol sci quant training).
- A gentle introduction, but with some detail on how R works as a programming language (perhaps a bit different to other intro courses).
  - Set some good programming/workflow habits to make your life easier later on.


*Provide you with some basic practical skills necessary to get serious about quantitative research.*

- There are tons of awesome & free materials from the best R Gurus of the world out there.
- Take this course as a humble starting point to see through the thicket.

**Note:** Text will be highlighted in **yellow** or **turquoise**. Links will appear **purple**.

# Hi!

## Who am I?

- ~~Research Fellow (pre doc)~~. Soon to be Junior Data Scientist in pharma. I like stats, programming, and causal inference. I also like cooking and (board) gaming.
- Web: 

## Who are you?

Let's do a break-out icebreaker session.

This will also be your team for the problem sets.

# What are we going to cover?

1. **Intro + R-Studio, and Git(Hub)**
2. Base R & Tidyverse Basics
3. Data Wrangling
4. Data Viz
5. Writing Functions
6. A complete scientific workflow with R

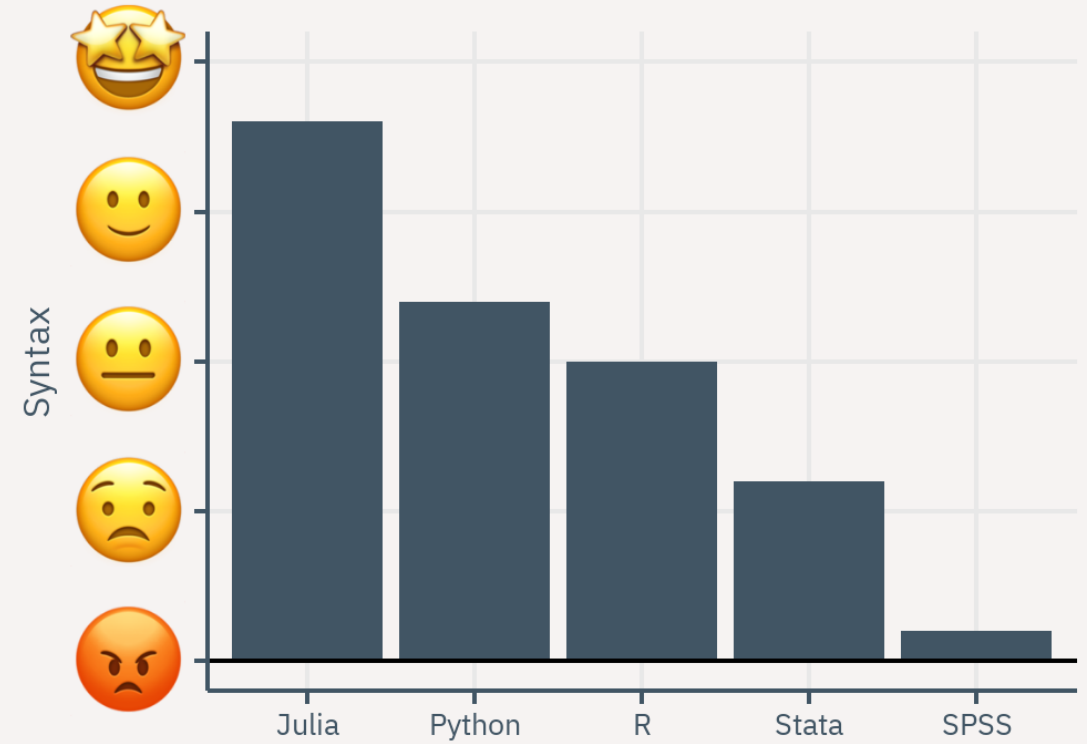
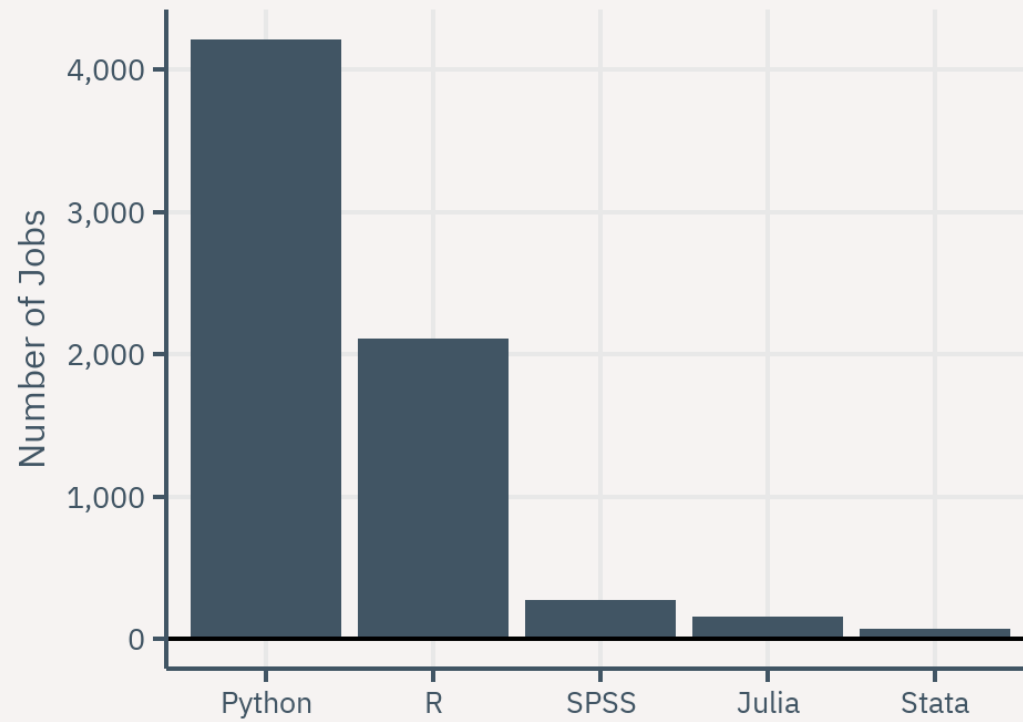
# Why R?

At the present day (and surely years to come), R is arguably the best programming language for academics:

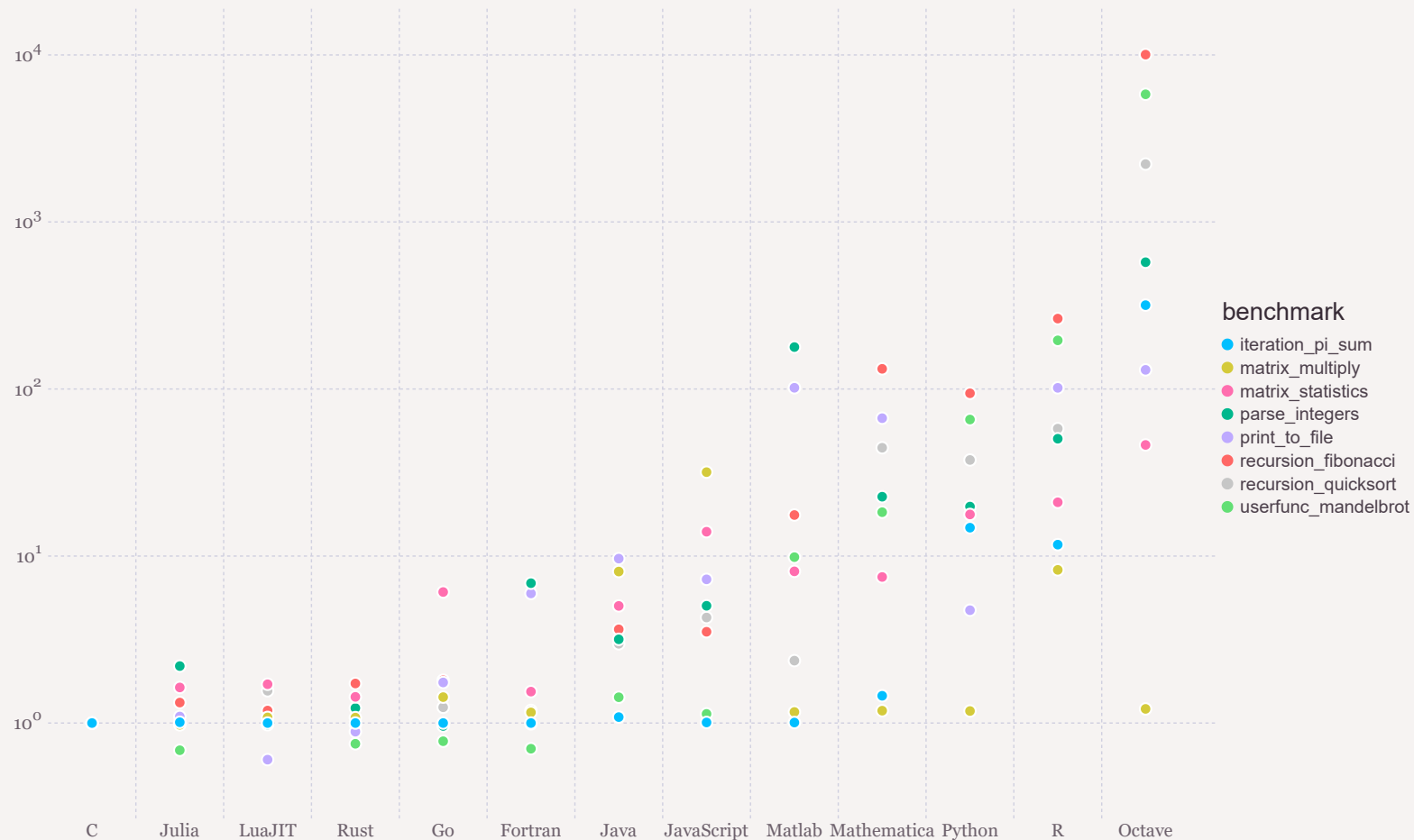
- R is from statisticians for statisticians.
- Most active (academic) development community for **statistical** computing/programming.
- Excellent IDEs; Good integration of other languages and workflow components.
- Best for data viz.

# Why R?

DS Jobs on Indeed.com, 01/05/2021



# R versus other langs/software

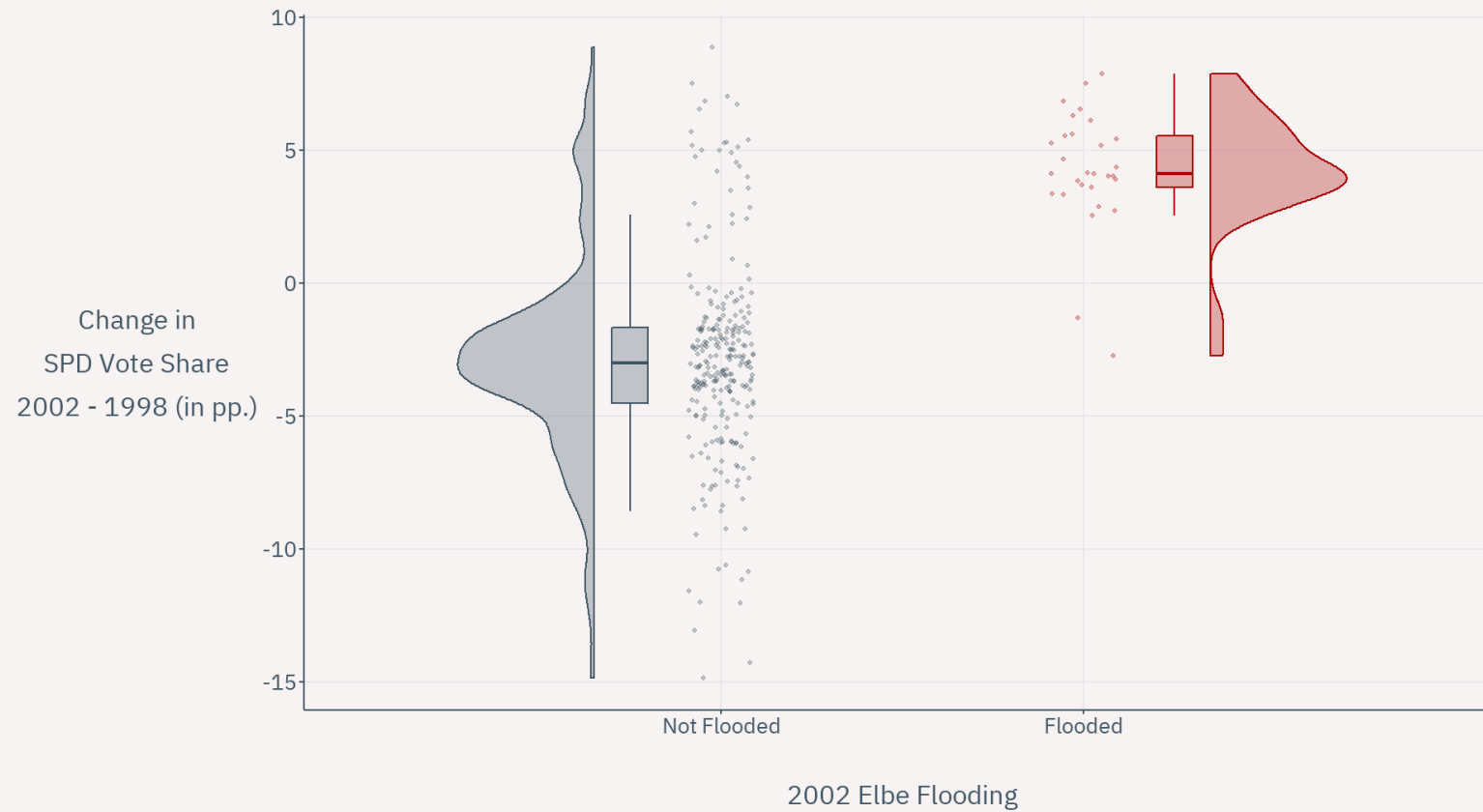


Source: **Julia Micro-Benchmarks.**



# Showcase

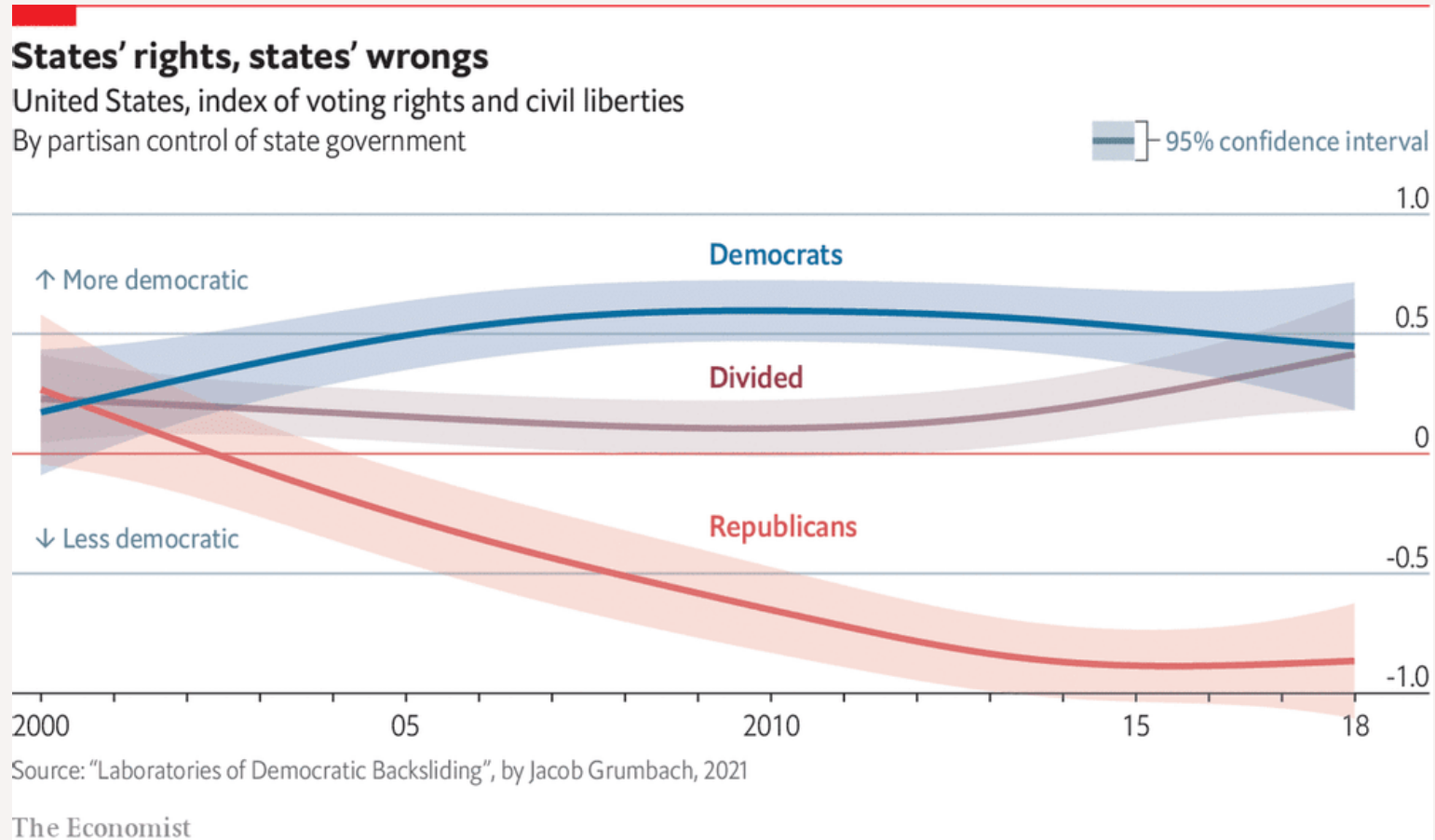
U can make pretty graphs...



Data from **Haimueller/Bechtel 2011.**

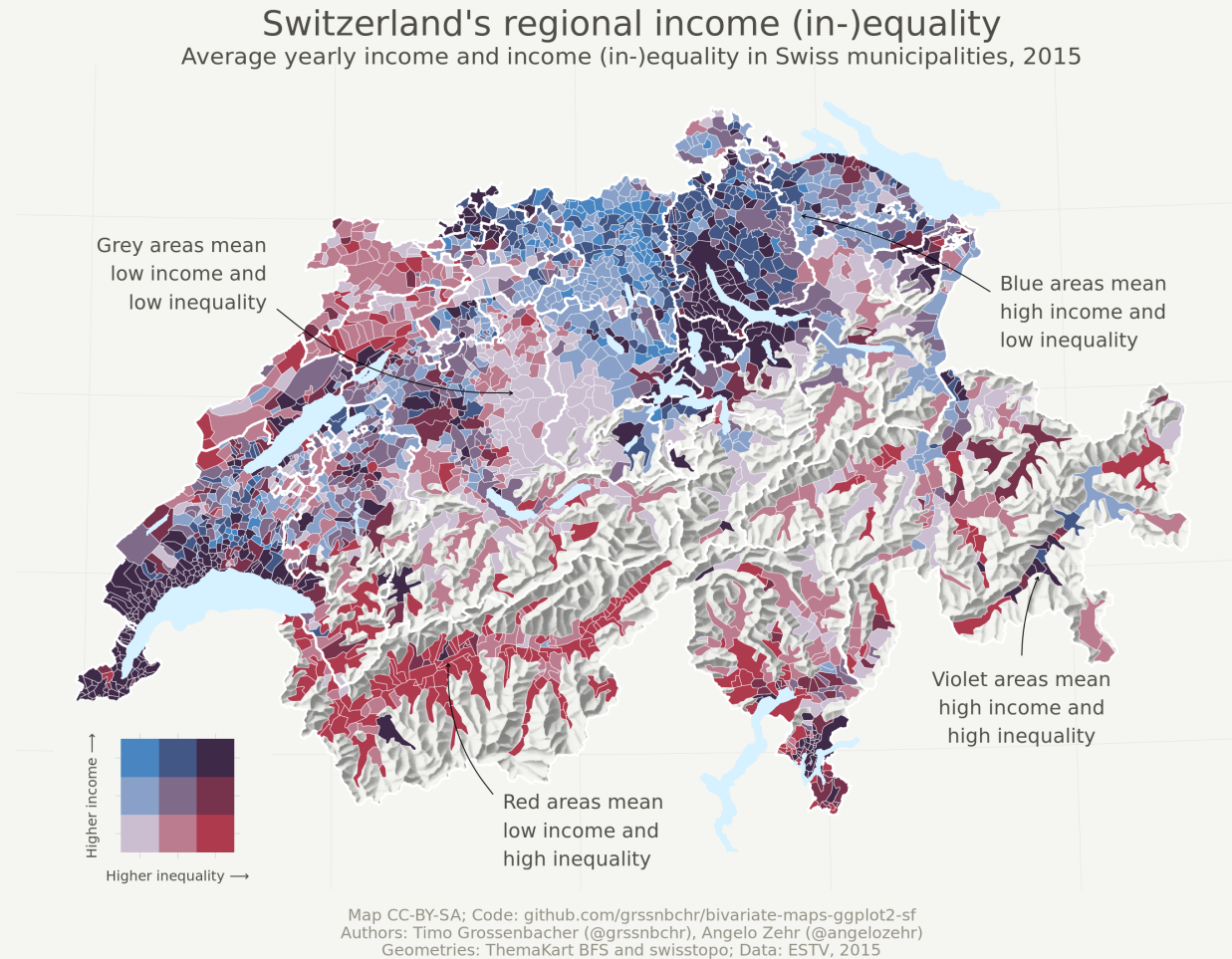
# Showcase

U can make pretty **graphs**...



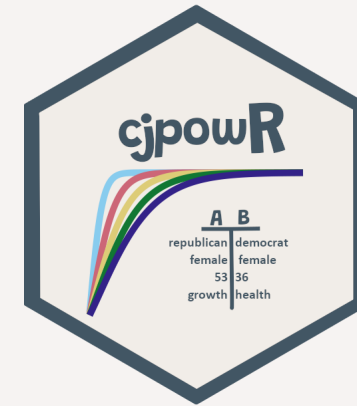
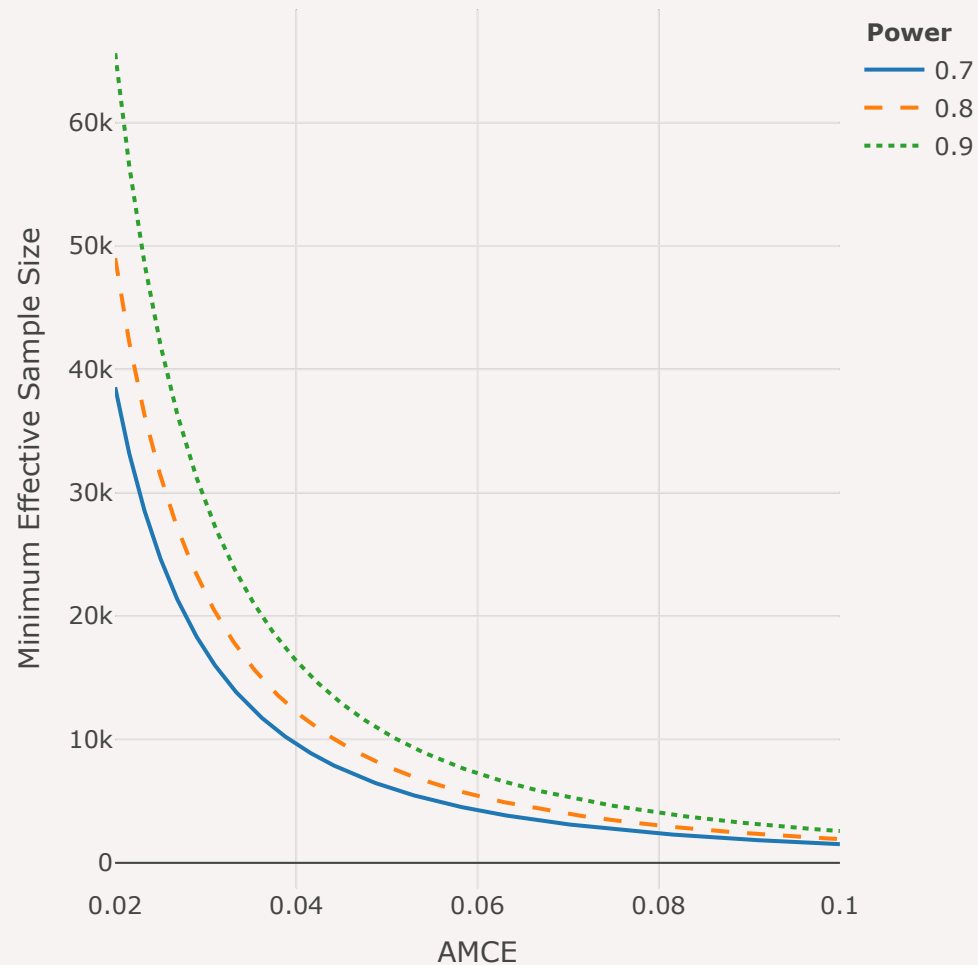
# Showcase

Or **maps**...



# Showcase

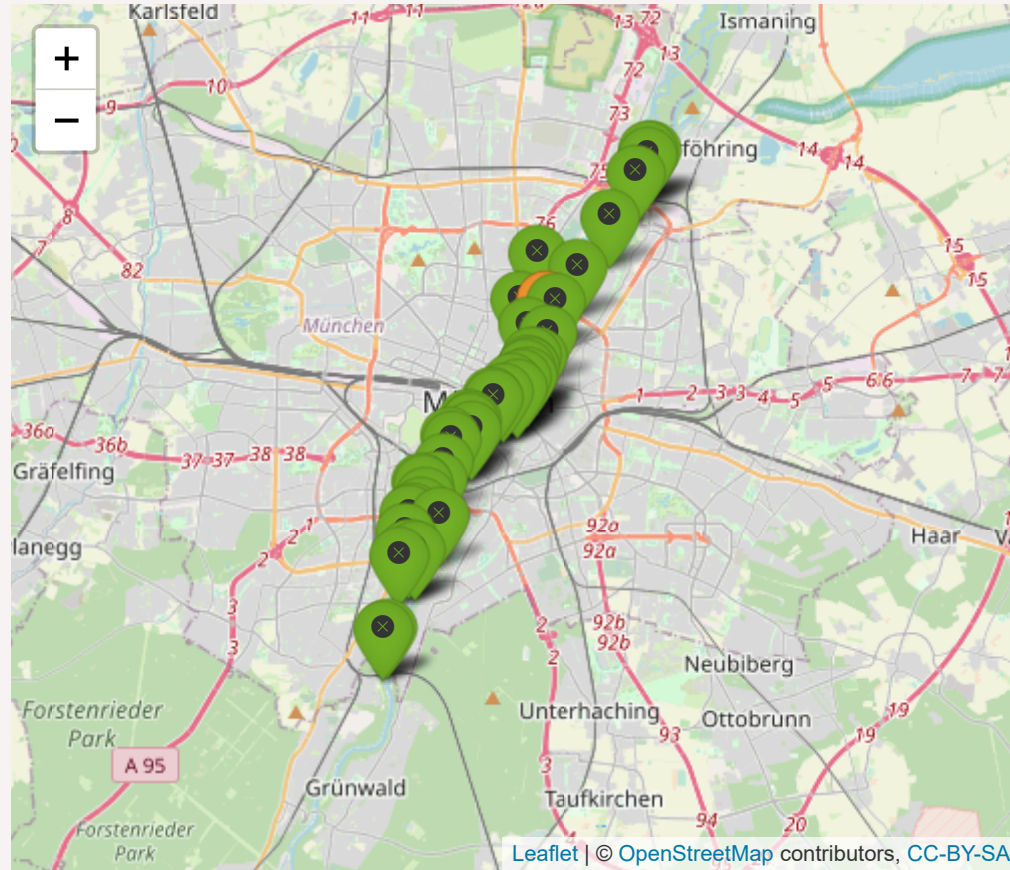
Or interactive graphs...



If you are interested in **power for**  
**(conjoint/factorial) survey experiments...**

# Showcase

Or or interactive maps...



Map shows **points of interest** at the southern part of the Isar in Munich (GSI in orange). Let summer come!

# Showcase

- You can easily combine R code and text in so-called Rmarkdown files to produce reproducible documents in various output formats (.html, .pdf, etc.).
- This presentation is a (hopefully good) example. You can check out the source code [here](#). Feel free to take and adapt.
- We will learn a little more about this in just a few slides (and a little more next week)!

# R-Studio & Git(Hub)

# Installation

Steps you should have done already:

1. Installed **R**.
2. Installed **R-Studio**.
3. Created a **GitHub** account. Take some care with the username if you want to keep this account throughout your career. You can't change it afterwards.
4. Installed and set up **Git** (and optionally a desktop client).



# R-Studio

R-Studio is an **IDE** (integrated development environment) for the R language:

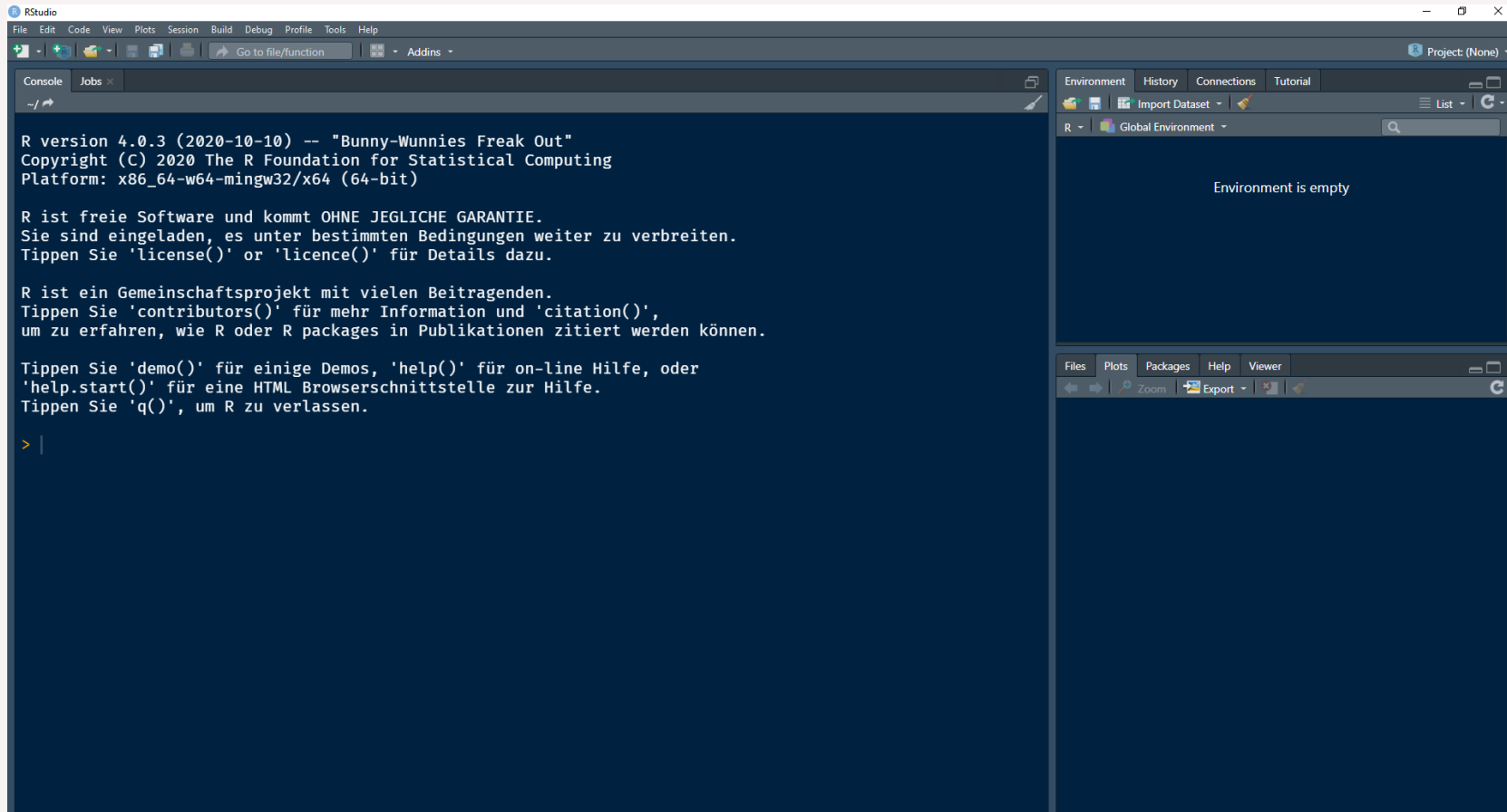
- comes with a console, code editor, tools for plotting, history, debugging, and workspace
- open source
- pretty accessible/easy to use

Alternatives:

- Visual Studio Code (nice, if you work with multiple langs; my fav IDE)
- Alternatives are a bit more clunky for package development, shiny platforms, and some other R-specific stuff
- If your main language is R, you can't go wrong with RStudio

# R-Studio

Let's take a tour...



# A Small Detour: Two Types of "Scripts"

In this course, we will use two types of scripts to write our code:

1. Classic R-Scripts (`.R`): simple text file; comments are usually done like so: `# A comment.`
2. **Rmarkdown** files (`.rmd`): combines code and free text (+ figures and formulae)
  - Can be "knitted" to, e.g., .pdf, .html and Word
  - Makes your documents (e.g. a paper or a thesis) fully reproducible
  - Nice for problem sets (hence, we will use it for this right from the beginning)

# Rmarkdown

An Rmarkdown file consists of mainly three things:

1. **YAML header** (Yet Another Markdown Language). Specifies meta info (e.g., author, date, document format, etc.):

```
---  
title: "Untitled"  
author: Markus  
output: html_document  
---
```

2. **Code chunks** surrounded by `````. You can execute each chunk individually.
3. **Plain text** formatted via Markdown, a markup language with very **straightforward syntax**.

Problem sets will come as pre-formatted `.rmd` files such that you can start working on exercises directly.

# R projects

- To keep our sanity when coding (and to produce something reproducible in the end), we want to keep all our data, analysis scripts, outputs, etc. together.

## Three approaches:

### Bad

Creating a folder in the explorer, dropping all files into it, and manually setting the working directory in the R script using, e.g., `setwd("C:/Users/XYZ/New Folder")`.

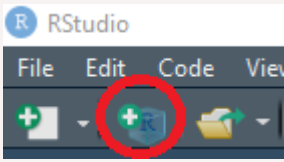
Just don't use `setwd()`. Ever.<sup>1</sup>

Why? Reproducibility.

[1] Yes, I am looking at you, Stata user, who loves to set working directories via `cd`.

# R projects

## Ok

Clicking , creating a local "R-Project".

This creates a folder with an `.Rproj` file.

Whenever you open an R-project, a fresh instance of R starts, and **the current working directory is set to the project directory**. You can then work with file paths relative to the project directory: E.g. `"Figures/somepicture.png"`.

**TIPP:** Future You should also check out the `here` package to make relative file paths more robust.

## Good

Creating a project and using version control.

This is where Git(Hub) comes in...

# Why Git(Hub)?

Having multiple scripts inside your project called, e.g., `thesis_analysis_final_01_revised_2.R` is a nightmare.

- Using Git(Hub) improves your workflow:
  - Helps with keeping track of the changes you make.
  - Makes (code) collaboration with other researchers easy.
  - Helps to make your research/code projects accessible/reproducible/open source.

# Why Git(Hub)?

What's that thing called "Git"?

- It's a version control system:

**Git  $\approx$  MsOffice track changes and restore features + dropbox/google drive version history**

- But it's better, especially for any kind of projects that involve code.
- You have to "commit" (approx. an additional save) actively, but that's a good thing!



# Why Git(Hub)?



# GitHub

GitHub to the rescue:

- Built on top of Git
- A kind of online cloud service that makes working with Git easier
- Again, no automated sync (but that's good!)
- Instead of in some folder, your project lives in a remote **repository**.

**The remote repository is your upstream storage.**

→ You can **clone** it from GitHub to create a local copy.

→ You can **fork** some repo (including those of other users); i.e., create a repo copy under "your repositories". You can then **clone** this forked repo to get a local copy.

# Your first repo

1. Click **here** and create a new Git(Hub) repo. Call it "test", set it to **private** , and initialize with a readme file.
2. In R-Studio, navigate to `File > New Project > Version Control > Git`.
3. Paste the Repository URL, chose a name and project path, and **clone** the thing.
4. You will be asked to provide a personal access token. Generate it **here** using some name and check the "repo box".
5. In the files tab, open README.md. Also, click on the Git tab.

Do this **now** !

# 4 Operations You Need to Know

## 1. **Stage** ("add")

- Tells Git which files u want to make changes (edits, deletes, etc.) to in the repo (simplification); in RStudio this boils down to "selecting" files/changes to files by checking them.

## 2. **Commit**

- Git's way to "save" the changes you staged.

## 3. **Pull**

- "downloads" all new changes/new commits from GitHub

## 4. **Push** (to origin)

- "uploads" all commits to GitHub; to the origin, i.e., your upstream remote repo.

# Commit

Make some changes to the `README.md` file **save**, stage, commit and push.

To establish best practice, give your commit a meaningful name:



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAANDS	2 HOURS AGO

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

- In general, commit whenever you think you made a meaningful change
- Push a bit less often than you commit

# Collaborate with Git(Hub)

- Using version control really comes to shine when collaborating.
- BUT: You are always collaborating with your future self. Therefore, always use version control.
- To invite someone to a repo on GitHub, go to the repo `settings > manage access`.
- Your collaborator can then clone the repo and contribute commits, push them, etc.

# When Things Go Sideways: Merge Conflicts

1. Go to your new repo on **GitHub**. Edit the README.md manually in line, say, **3**.
2. Commit some changes in the **same** line locally in R-Studio.
3. Pull ( **don't push** ).
4. Git:



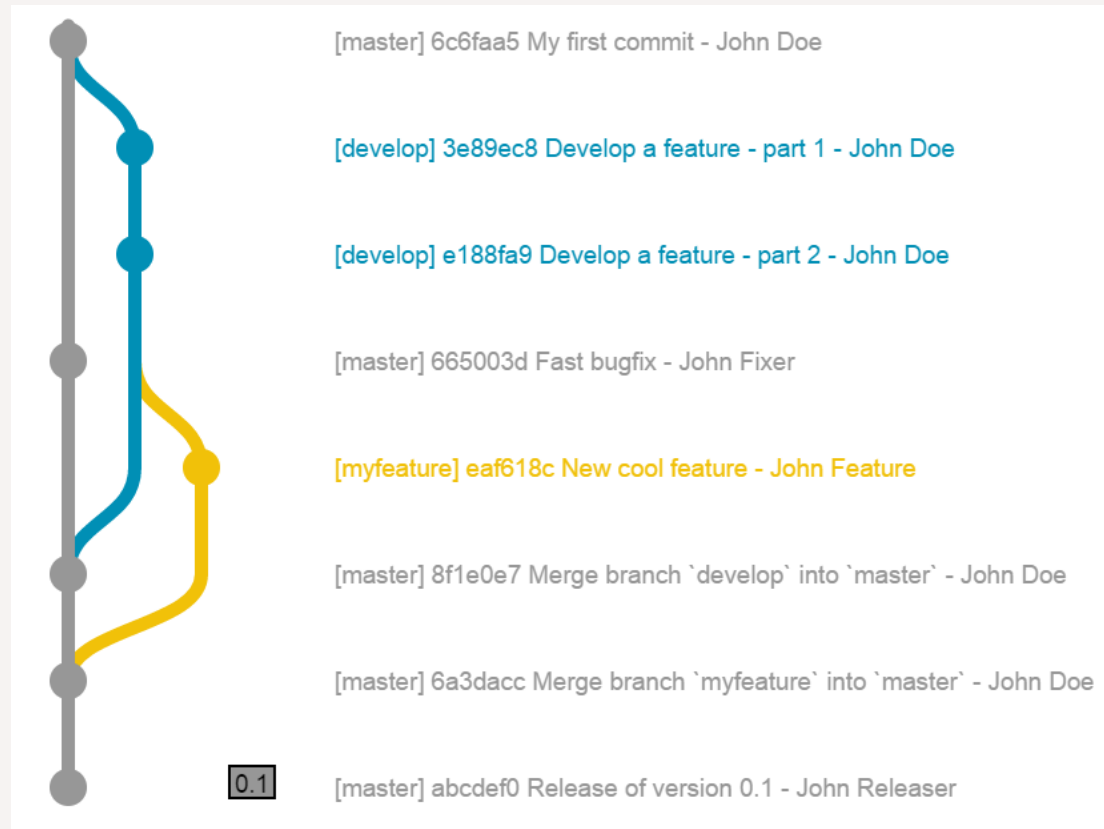
# When Things Go Sideways: Merge Conflicts

What do you do now?

- Well, you (maybe in exchange with your collaborator) decide!
- Solve the conflict manually, add, commit, push.



# Branches

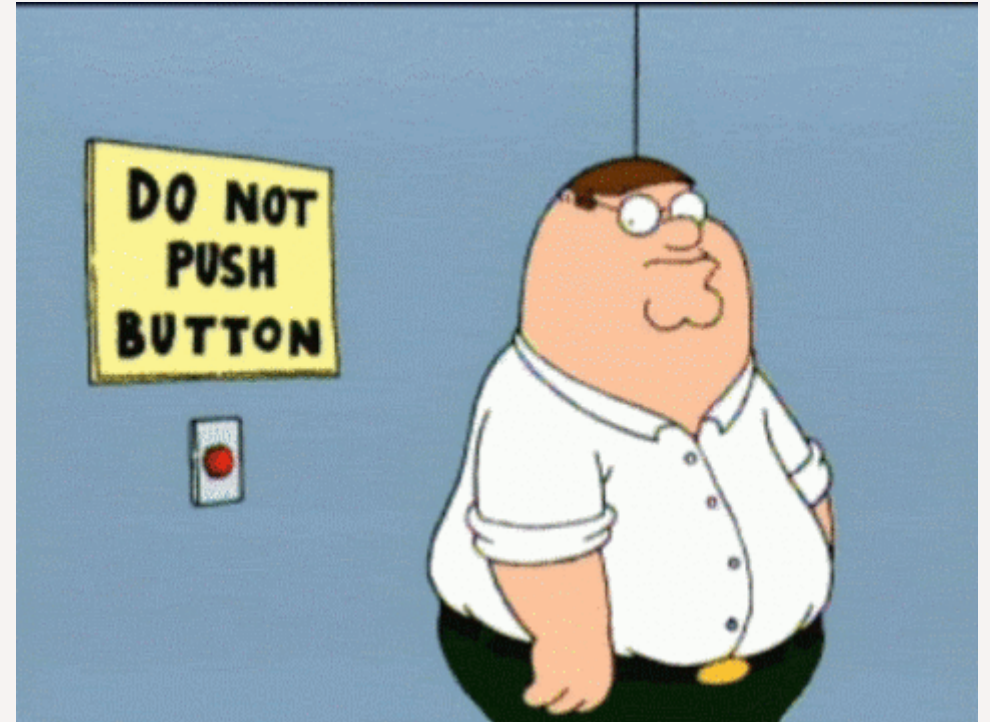


# Branches

- **Branches** allow you to develop/test some ideas without touching the main version of your code.
- Useful for larger projects. You get a full copy of the repo, and you can commit/pull/push all you want.
- If your ideas turned out not to work. Just delete the branch.
- In R-Studio, they can be created **via the little purple branch icon in the Git tab.**
- If you want to integrate the feature/idea into the main branch, issue a **pull request** (the easiest way is via GitHub).

# When Things Go Really, Really Wrong

- **DON'T PUSH.**
- If you did not, just clone a fresh instance of the repo.
- If you did, you can revert to an older version (that may be easier in some cases but more work in others).



# Workflow Summary

1. Create a repo or fork some existing repo
2. Invite collaborators
3. Clone it & create an R project
4. Edit code or make other changes
5. Stage, commit (with a message), pull (esp. if u work with others to avoid conflicts right away), push
6. Rinse and repeat steps 4-5.

# Workflow Summary

Yes, the order really is stage, commit, pull, push.

**Always commit first.**

Pulling **after** committing does **not** overwrite your work. If you committed, what you pull gets compared to what you committed. If things conflict, it's up to you to resolve. If things work fine, great!

If you work alone, you don't really need to pull if you do not change stuff on the remote repo manually.

# Further Steps

- Fork the course repo and clone it via R-Studio (that's perhaps a bit hacky but easy and does the job).
- You will need this to access the course materials/problem sets conveniently.
- As the fork is your own copy of the repo, feel free to commit and push all you want.
- You could also use the **shell** to clone the repo or, alternatively, GitHub Desktop.
- For beginners, using a Git GUI (like GitHub Desktop) and R-Studio will cover well over 90% of the use cases.

See, e.g., **here** for further reading.

**Next Up: Base R & the Tidyverse**