Git, GitHub and Reproducible Research

Alexander E. Zarebski

October 20, 2022

Contents

1 Introduction

Welcome to *Git, GitHub and Reproducible Research*! In this tutorial we will look at ways git and GitHub can help you with your computational research, including how make it more reproducible. This is good for science as a whole as it means others can build upon your work, but it is also good for you to help you keep track of your projects. You will learn the fundamentals of git and gain practical experience in the use of GitHub. For those interested in learning more about version control, there is also a list of additional resources and some homework problems at the end to test your understanding.

Please read through the background material first then follow along with the worked example. You do not need any special software installed (everything is done in the browser), however the worked example does make use of R and ggplot2. Installing RStudio is probably the easiest way to utilise these software. If you find an error in this document, or would like to suggest an improvement, please make a note of this in the issue tracker on GitHub.

1.1 Disclaimer

These materials makes use of proprietary products and a simplified workflow in order to emphasise the main concepts and to save on installation and configuration time. Some references will be given at the end to direct you to free and open source solutions and more sustainable workflows. The core ideas we will see are largely the same regardless of the software you use, so understanding this material should make it much easier to migrate to more sophisticated methods in the future, should the need for them arise.

2 Background material

This section provides some basic information about version control, git, and GitHub. These are complicated programs so we will only see some of the basic features today; it should be enough to get you started should you want to learn more though.

2.1 Version control systems (VCSs)

A *version control system* (VCS) is a system to help manage the writing and maintenance of things such as software, documents, and websites. These systems were developed to manage large software projects but are useful at many levels. For example, version control can help you avoid the following situation: `my-analysis.R`, `my-analysis-final.R`, `my-analysis-final-again.R`, and so on. Try coming back to that 3 months later when a reviewer asks you to re-run something with a slight modification! Similar to the way in which an interactive debugger allows you to step through the evaluation of your program,

version control allows you to step through the process in which the code was written. You can think of it as an extreme version of undo/redo.

In addition to helping you organise your files, a version control system (VCS) and its associated tooling can help the scientific community by making your research reproducible and open. For your computational research to be considered *reproducible,* it needs to be described in such a way that others can replicate your results. For it to be *open*, the materials (code, data and sufficient documentation to use it) need to be available for others. Simply dumping all of your code into something like GitHub is not sufficient for your research to be considered reproducible.

2.2 Git

Created in 2005 by Linus Torvalds[1] to help with the development of the Linux kernel, git has become a fundamental tool for software development. In the 2021 Stackoverflow Developer Survey over 90% of respondents used git; it is nearly synonymous with version control. If you intend to collaborate in the writing of substantial amounts of code, taking the time to learn how to use git is a good idea. This can be made easier by learning to use a GUI interface too.

Working with git will be much easier once you are familiar with some of the terminology (there is a lot). Unless you are familiar with git already, you should at least skim these before continuing.

Repository

A *repository* (a.k.a. a *repo*) is a directory (a.k.a. folder) containing your files and the history of the edits (see commit) you have made to these files. You can have a (local) repository that only lives on your machine, but they are often shared on a platform such as GitHub.

Commit

An edit to a file that you have recorded as part of the history of edits is called a *commit*. It is both a noun and a verb, you commit an edit and the repository contains all of your commits. This can be thought of as a stronger version of saving a file. Each commit gets a unique identifier (called a *hash*). Sometimes we use "commit" to refer to the state of all the code after an edit. If you want to go back to the version of your code from three weeks ago, you just checkout the commit from that date and all the files will return to the state they were in at that time.

Clone

When you make a copy of a repository you are *cloning* that repository. The resulting copy is referred to as a *clone*. Typically this will mean you have downloaded a copy

[1] Legend has it, he named git after himself.

from a platform such as GitHub. Note that downloading a ZIP file with the contents of a repository is not necessarily the same as cloning it, when you download the ZIP file you'll get the files, but you may not get the history of commits associated with them.

Pull

Suppose you cloned a repository a while ago and you want to get a copy of all the commits that have been made to the original repository since then. To get these commits you *pull* them, which is a fancy way of saying updating your files. This is sometimes referred to as *fetching*. There are some subtle differences that you probably won't ever need.

Push

If you have committed some changes to your clone of a repository and want the original repository to have these changes as well, you *push* these changes. This is a fancy way of saying use your edits to update the original files.

Branch

A *branch* is similar to a clone in that it is a copy of a repository. This provides a more sophisticated way for people to work on their own version of code, without messing up the main copy. This is not particularly important unless you are collaborating with others on a project. It provides a safe way to experiment with your code without risking messing up your "good copy".

Merge

If someone has made some useful changes on their branch the owner of the repository may decide to include their commits in the main copy. This process of including the changes on someone's branch is called *merging* the changes. As with branches, this is likely only to be relevant if you are collaborating with others on your code.

Fork

When you make a copy of a repository that sits on your GitHub account. This is similar to, (but distinct from) cloning and making a branch. Forking a repository is an important part of contributing code to other peoples' projects when using GitHub, however the details of this are beyond the scope of this tutorial.

Pull request (a.k.a. PR)

A pull request is a way to request that the owner of a repository accepts the changes you are proposing. As with forking, the details are of this are beyond the scope of this tutorial.

Client

A *git client* is a program that "simplifies" the use of git by hiding some of the details and giving you GUI to click buttons on rather than typing mysterious text at the command line. We will use the GitHub website in this tutorial because it handles most of this for us and avoids any complicated installation. If you are going to be making substantial use of git, I would recommend getting a client and learning how to use it. GitHub Desktop and Sourcetree are both free options and are developed by major companies so should have excellent documentation and be easy to use[2].

If you are already using RStudio, you might be interested in the features it provides for version control. There is a tutorial on using RStudio as a GitHub client, but it requires some command line experience and for both RStudio and git to be installed on your machine.

2.3 GitHub

What is GitHub?

GitHub, Inc. is a subsidiary of Microsoft. Their website provides freemium hosting of git repositories. In addition to hosting the repositories, it offers additional tools to assist with software development. We will use GitHub in this tutorial to avoid you needing to install anything on your machine. If you are going to use git extensively, it would be wise to learn how to do this from the command line or some other program. Please don't mistake git and GitHub, one is a piece of software, the other is a platform that uses this software.

Setting up a GitHub account

To register an account you will need an email address that can be used for verification. A Gmail account makes this particularly easy. Please follow the following steps to register an account:

1. Visit `https://github.com/` and click **Sign Up**.
2. Fill in the forms to create an account. **ProTip:** Choose your username wisely!
3. Verify that account by entering the access code GitHub sends to the email address you registered with.
4. Verify that you can summon the **Command Palette** with `crtl k` for Windows and Linux and `command k` on a mac.
5. The appearance and accessibility settings can be reached by searching for them in the command palette.

You will need an account to complete the worked example below.

[2] I use magit, which is nice if you already use emacs, but otherwise may be a bit weird.

2.4 Zenodo

Zenodo is an open access archive operated by CERN which allows researchers to archive research materials with a DOI[3] which makes them easier to find and cite. This is a more permanent form of storage than GitHub. It is easy to archive a particular commit of a repository which is good practice if you want to refer to a particular version of some code in a paper. GitHub has tooling for this build in.

3 Worked example

Now that we have an understanding of version control and its associated tooling, we can see an example of how this enables us to do more reproducible research. Suppose you wanted to ensure that the analysis leading to Figure 1 was reproducible. In this worked example we will work through the process of setting up a repository and uploading the relevant files. A copy of the resulting repository is available here.

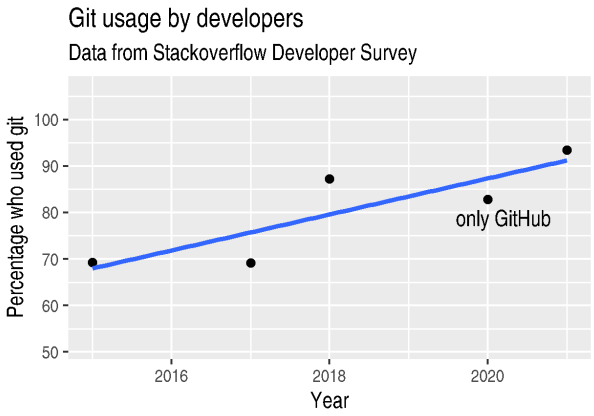

Figure 1: The percentage of developers who use git has increased since 2015. Data from the Stackoverflow Developer Survey is shown as black points and a least squares regression is shown as a blue line, the trend is not significant at 95%.

We will start by looking at the code and data used in this analysis, then consider how one might organise the files. Once we have the code in a sensible state we can put it in a GitHub repository and add some finishing touches such as a license, some documentation, and some additional information about package versions. We then consider the situation in which changes need to be made by a collaborator.

[3] A *Digital Object Identifier* (DOI) is a unique and persistent way to refer to a document online, for example `https://doi.org/10.1371/journal.pcbi.1009805`.

3.1 Code and data

Start by making a new directory called `git-usage` which will hold all our files. The data and the code that generated this figure are included below. This data should be saved in a called `stackoverflow-git-data.csv`.

Git Usage

```
year,percentage
2015,69.3
2017,69.2
2018,87.2
2020,82.8
2021,93.43
```

We then need a script to carry out the analysis. Save the following code in a file called `make-plot.R`

```
library(ggplot2)

sods_data <- read.csv("stackoverflow-git-data.csv")

g <- ggplot(
  data = sods_data,
  mapping = aes(x = year, y = percentage)) +
  geom_point() +
  geom_smooth(method = "lm") +
  geom_text(
    aes(x = 2020, y = 82.8, label = "only GitHub"),
    nudge_x = 0.2,
    nudge_y = -4) +
  labs(
    x = "Year",
    y = "Percentage who used git",
    title = "Git usage has increased",
    subtitle = "Data from Stackoverflow Developer Survey")

ggsave(filename = "git-usage.png",
       plot = g,
       height = 7.4,
       width = 10.5,
       units = "cm")

sink(file = "regression-summary.txt")
summary(lm(percentage ~ year, data = sods_data))
sink()
```

Once we have run the `make-plot.R` script, the directory should contain four files and have a structure like the following.

Directory contents

```
git-usage
+-- git-usage.png
+-- make-plot.R
+-- regression-summary.txt
+-- stackoverflow-git-data.csv
```

In the next section we will go through cleaning this up so it is easier for people (including yourself in the future) to make sense of this.

3.2 Organising the data and code

As a first step we will use directories to impose a sensible structure to our files. Organising files in this way is useful as it makes it far easier for someone to understand what each file is needed for. Follow the following steps (starting from within `git-usage`,) to organise your code more appropriately:

1. Make a directory called `src` and move `make-plot.R` there.
2. Make a directory called `data` and move `stackoverflow-git-data.csv` there.
3. Make a directory called `out` which we will write results to.
4. Fix the call to `read.csv` in `make-plot.R` so it can find the CSV since it now lives in the `data` directory.
5. Fix the calls to `ggsave` and `sink` so they write their output to the `out` directory.

Once you have done this, the R script should look like the following.

```
sods_data <- read.csv("data/stackoverflow-git-data.csv")

...

ggsave(filename = "out/git-usage.png",
       plot = g,
       height = 7.4,
       width = 10.5,
       units = "cm")

sink(file = "out/regression-summary.txt")
summary(lm(percentage ~ year, data = sods_data))
sink()
```

Once you have run the code (with `git-usage` as your working directory), the directory structure should look like the following. Note how the output files now appear in the `out` directory. If you are running the script from an R REPL, remember you can use `setwd` to specify the working directory.

Git Usage

```
git-usage
+-- data
|   +-- stackoverflow-git-data.csv
+-- out
|   +-- git-usage.png
|   +-- regression-summary.txt
+-- src
    +-- make-plot.R
```

3.3 Uploading to GitHub

Now that our code is in a reasonable state, we can upload it to GitHub. If you do not already have a GitHub account, please follow the instructions above, which describe how to make one. Once you have a GitHub account, you can follow the following steps to upload these files:

1. Visit `https://github.com/` and create a new repository by clicking **New**, you will need to pick a name for the repository (I called mine `git-usage`.) The default settings provided by GitHub are fine. Click **Create repository**.

2. We now need to commit our files and push them to the remote repository. However, since we are doing this through GitHub, it is all combined into a single step. Click **Add file** and then **creating a new file** to start the process of adding the `src/make-plot.R` file.

   a) Ensure the name of the file is `git-usage/src/make-plot.R` (be careful that you have the *path* with the \ correct.)

   b) Copy-and-paste the code in `make-plot.R` into the text box provided.

   c) Click **Commit new file** button.

   If you are struggling to make a new directory in GitHub, see the next section.

3. Repeat this process with `data/stackoverflow-git-data.csv` and the output TXT file. In the case of the PNG image, `git-usage.png`, you will need to use **Upload file** instead of **Create new file**.

3.4 Making directories via GitHub

You cannot add an empty directory to a GitHub directory. If you want a new directory to be added, you need to commit a file to it. One convention for this is to make an empty file in the desired directory (often this will be a file called `.gitkeep`).

3.5 Adding a license

A license specifies what people can do with your code. If you aren't sure what license suits your needs, you might find `https://choosealicense.com/` has some helpful information. Most of the time, I will opt for the MIT license.

There are two ways you might add a license. The manual method is to copy and paste the license text into a file called `LICENSE` to your repository, filling in `[year]` and `[fullname]` as appropriate. Alternatively, you can **Add file** and **Create new file** and specify that the file will be called "LICENSE" and it will offer you some templates to choose from. It will auto-fill the details of your name and the year.

3.6 Adding a README

When you encounter a repository online it can be difficult to understand what its purpose is and how to use it. "README" is the name given to a file that contains this sort of information. Typically these will be written in markdown (similar to RMarkdown). Add a file called `README.md` to your repository with text similar to the following.

````
This repository contains an analysis of git usage through
   time.

To run this analysis use the following command:

```
Rscript src/make-plot.R
```

The input data is in `data` and the results are in `out`.
````

3.7 Recording the session information

Software gets updated, and sometimes these updates cause things to break. Where possible, it is very good practise to include details of the versions of software you have used. When working with R the `sessionInfo` command makes this simple. Try adding the following to the end of the `make-plot.R` script.

```
sink(file = "out/package-versions.txt")
sessionInfo()
sink()
```

The next time that you run this script, it will write a description of the version of R you used and the versions of all the loaded packages to the file `out/package-versions.txt`. Try running the script again to make sure this additional file was generated and contains something similar to the following.

```
Session information
R version 4.1.2 (2021-11-01)
Platform: x86_64-pc-linux-gnu (64-bit)
```

```
Running under: Ubuntu 20.04.3 LTS

Matrix products: default
BLAS:   /usr/local/lib/R/lib/libRblas.so
LAPACK: /usr/local/lib/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
 [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
 [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] ggplot2_3.3.5

loaded via a namespace (and not attached):
 [1] magrittr_2.0.1   splines_4.1.2    tidyselect_1.1.1 munsell_0.5.0
 [5] colorspace_2.0-2 lattice_0.20-45  R6_2.5.1         rlang_0.4.12
 [9] fansi_0.5.0      dplyr_1.0.7      tools_4.1.2      grid_4.1.2
[13] gtable_0.3.0     nlme_3.1-153     mgcv_1.8-38      utf8_1.2.2
[17] withr_2.4.3      ellipsis_0.3.2   digest_0.6.29    tibble_3.1.6
[21] lifecycle_1.0.1  crayon_1.4.2     Matrix_1.3-4     farver_2.1.0
[25] purrr_0.3.4      vctrs_0.3.8      glue_1.6.0       labeling_0.4.2
[29] compiler_4.1.2   pillar_1.6.4     generics_0.1.1   scales_1.1.1
[33] pkgconfig_2.0.3
```

Once you are happy that this has worked, we need to commit these changes. First by editing the script, and second, add the `package-versions.txt` file.

3.8 Branching and merging

Suppose that after doing all of this one of your collaborators wants to adjust the figure. We will now go through the steps involved with doing this using branches.

Branching to make changes

Figure 2 is a modification of Figure 1 with the desired changes.

To avoid making changes to the main copy of the code we will work on a branch, and then when we are happy with the changes we will merge them. To start with, create a new branch by clicking on the drop-down menu labelled "main" as shown in Figure 3. I called it "edit-plot", but you can use anything other than "main" (because that is the default branch name used by GitHub).

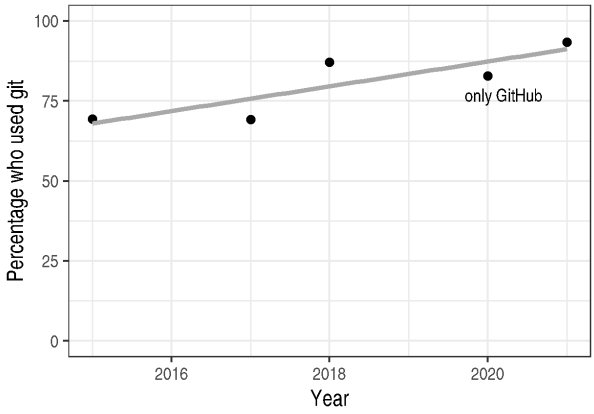

Figure 2: The percentage of developers who use git has increased since 2015. Data from the Stackoverflow Developer Survey is shown as black points and a least squares regression is shown as a grey line, the trend is not significant at 95%.

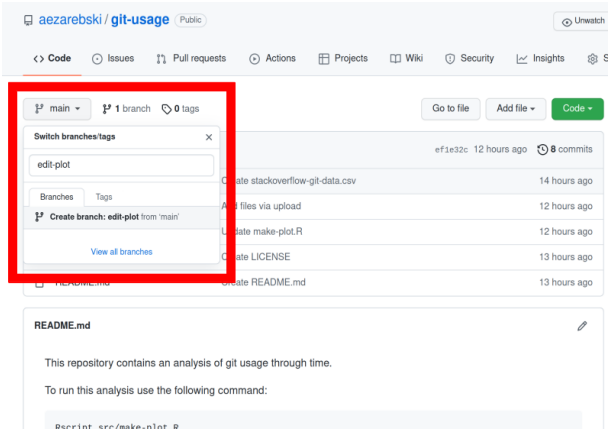

Figure 3: Create a new branch using the drop-down menu.

Make desired edits to the code and output

Making sure that you are on your branch — if you're not sure, click on the **branch** button to double check — edit the `make-plot.R` script so that it has the following

```
g <- ggplot(
  data = sods_data,
  mapping = aes(x = year, y = percentage)) +
  geom_point() +
  geom_smooth(method = "lm", colour = "darkgrey") +
  geom_text(
    aes(x = 2020, y = 82.8, label = "only GitHub"),
    size = 3,
    nudge_x = 0.2,
    nudge_y = -6) +
  labs(
    x = "Year",
    y = "Percentage who used git") +
  ylim(c(0,100)) +
  theme_bw()
```

Once you have made the changes and re-run that script the figure in `git-usage.png` will have changed — it should look like Figure 2 now. Ordinarily, you would update the figure in the same way that you update code, by committing the changes. However, this is tricky to do via the GitHub website for image files, so instead, delete the file and upload the modified one. At this point it might be interesting to move between the `main` branch and your new branch to see how the files change between the two.

One motivation for branches is that you can make exploratory changes without risking messing up your code on the main branch. If you have a collaborator that wanted to try something, they could do so on a separate branch and then, if you like their edits, you can merge them into `main` as we are about to do now.

Merge the changes

To merge your changes via the website, go back to the main page of the repository and you should see a new button, like the one shown in Figure 4, inviting you to compare the changes on this branch, i.e., to inspect if you consider this work worthy of inclusion.

Inspect the differences between the branches and if you are happy with them create a pull request by clicking the button as shown in Figure 5.

Once you have created the pull request, the next step is to merge that branch into the `main` branch. To do this you just need to click the button shown in Figure 6.

Once a branch has been merged it will hang around until you delete it. Since having old branches around can lead to confusion, it is sensible to delete them afterwards. As shown in Figure 7 there is a button to achieve this.

At this point you should only have a single branch left and it should have the modifications to the figure. Congratulations on a reproducible analysis!

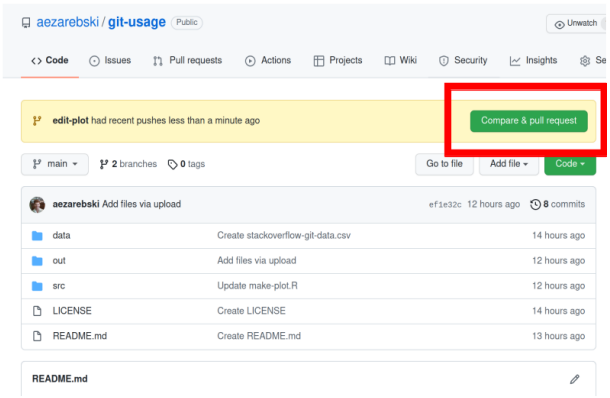

Figure 4: A button appears to invite you to compare branches.

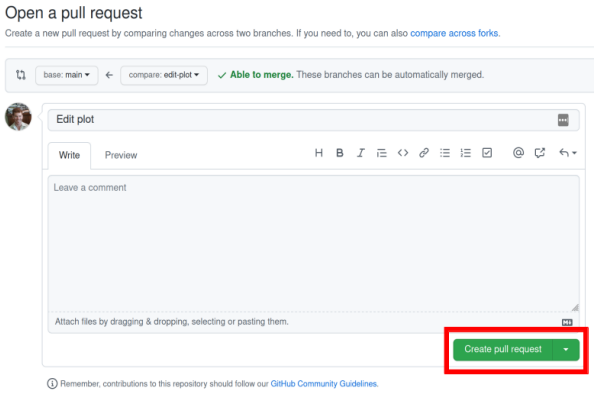

Figure 5: If you are happy with the content of a branch, you can create a pull request.

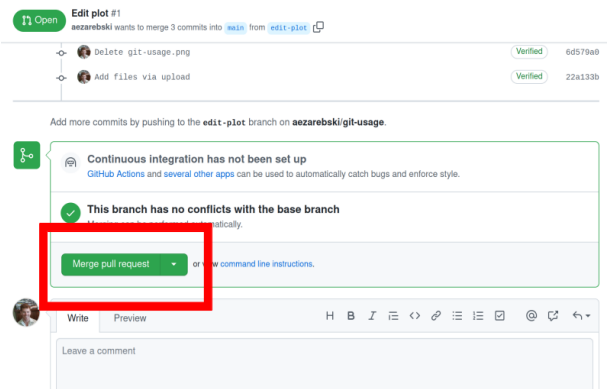

Figure 6: If you accept a pull request you can merge the changes with the Merge pull request button.

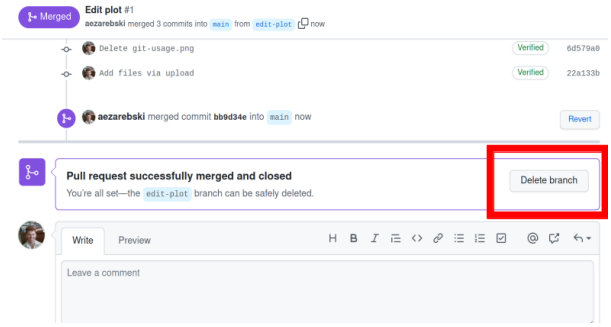

Figure 7: Deleting a branch after it has been merged keeps the repository tidy.

4 Next steps and alternative solutions

4.1 Help! I just want to download the files

If you want to download the files from GitHub and do not want any of the associated git functionality, you can download a ZIP file that contains the contents of a repository. Figure 8 shows the menu for downloading a ZIP file containing the contents of a repository. If you want to use any of git's features though you should clone the repository instead.

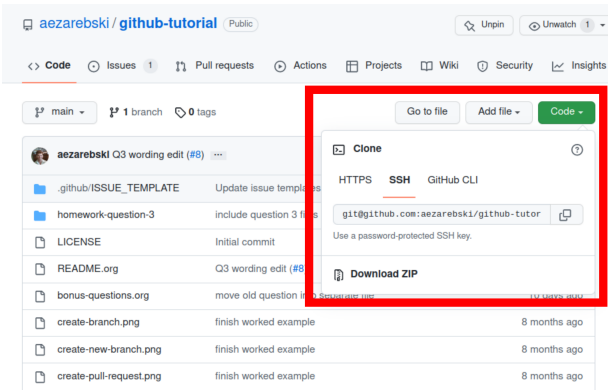

Figure 8: You can download a ZIP file containing the contents of a repository from GitHub.

4.2 What if I am using Python, or some other language?

We have used the R programming language in this tutorial, but how we use GitHub is language agnostic (for the most part). Most of the time, organising your data and source

code in this way is a good idea. Different programming languages record their package versions in different ways. Recall in this section that we generated a file containing the package versions. For the Python language, there is the pip function. Running `pip freeze` at the command line will print out the package versions. You can pipe this information to a text file (which is conventionally called `requirements.txt`) with the following command:

```
pip freeze > requirements.txt
```

If you want to use multiple Python packages, it's a good idea to invest time into learning how to set up a virtual environment. A virtual environment is an isolated group of packages used for a project. If you want understand how to do this, see the relevant python documentation.

4.3 Upload to Zenodo

The Zenodo FAQs contain information about how to archive a GitHub repository if you want a more permanent form of storage. Ideally, one would archive the commit used to generate the contents of a manuscript so it has a DOI and reference both the archive and the *live* version of the code on GitHub in the manuscript.

4.4 Learn more about git

- Pro Git by Scott Chacon and Ben Straub is a free book that is the ultimate guide but is a bit technical at times.
- Atlassian/Bitbucket has excellent tutorials.
- Learn Git Branching is a game revolving around explaining git.
- GitHub Learning Lab has some introductory material on the use of git and GitHub.
- Stackoverflow questions will often have answers to your questions.
- Inside the Hidden Git Folder - Computerphile gives a bit of a behind the scenes tour of how git works.

4.5 Learn more about GitHub

There are lots of features in GitHub that haven't been covered but may be worth looking into:

- the issue tracker,
- the wiki,
- VSCode integration,

- GitHub Pages[4],
- and GitHub Actions.

4.6 Alternative solutions

Git

Git has the greatest market share but there are alternatives such as Subversion, Mercurial, CVS and Darcs. Given that the overwhelming majority of people use git, your time is probably best spent learning git. As mentioned above, if you are going to be using git a fair bit, it is probably worth learning how to use a client as well.

GitHub

While git dominates the market as the choice of version control system, there are many viable alternatives platforms to GitHub which may be more suitable for your needs:

- Bitbucket
- GitLab
- SourceForge

Zenodo

There are good general purpose alternatives to Zenodo such as figshare and Dryad. Institutions and journals often have a favoured provider, but they are reasonably interchangeable. There are also numerous alternatives that are more field specific, such as GISAID for genomic data.

Trusting a plain text file to describe my packages

If you need additional assurance that your work will be reproducible, it may not be sufficient to use `sessionInfo` (or `pip freeze` for Python) to record the versions of the packages that you use. There are ways to capture a lot more information about the environment you are working in so it can be recreated by others. Unfortunately, these methods can be complex and require a lot of configuration. Some popular examples include

- Packrat which helps manage R packages,
- docker for a general purpose solution which is probably a bit overboard,
- and Nix for a general purpose solution that is definitely overboard.

[4] GitHub Pages offers free hosting of static websites. You may have accessed this tutorial via a GitHub Pages site.

5 Homework

Please ensure that in answering these questions, you use a format that is easy to read and supports hyperlinks. The ability to include chunks of code (or mono-spaced fonts) may be useful. You want to display the output of the `tree` command. I would recommend using something like RMarkdown.

5.1 Question 1

Explain (in 50–150 words) how the git, GitHub, and Zenodo complement each other and their respective roles. Describe the value of one of the additional features of GitHub not covered in this tutorial (in 50–100 words).

5.2 Question 2

Explain (in 100–150 words) the function of version control in reproducible research. Give an example (in 50–100 words) where version control alone does not suffice to make a piece of work reproducible.

5.3 Question 3

This question will test your ability to organise the artefacts of a computational project. Download the scripts and data files using the links below and run them. Organise the files you have downloaded and the results of running them in an appropriately structured GitHub repository. Give a brief overview of the decisions you made along the way (in 100–200 words). Once you are happy with this, download a ZIP file for this repository and include it as part of your submission.

- Data description
- Melbourne rainfall data
- Oxford rainfall data
- First R script
- Second R script

If you cannot download these files directly, they should also be available here.

5.4 Question 4

This repository contains an attempt at visualising two datasets. Unfortunately, a bug was introduced somewhere during that attempt. The attempt consisted of the following steps:

1. Plotting the data in `iris.csv` using `make-fig-a.R`.

2. Plotting the data in `mtcars-renamed.csv` using `make-fig-b.R`
3. Realising that there is duplicated code and refactoring it: `reshape-data-function.R`.
4. Tweaking the figures to make them clearer.
5. And finally, realising that something is wrong with `fig-a.png`!

Referring to the commit history of the repository, answer the following questions (in 150–250 words **total**):

1. What bug was introduced, and how would you fix it?
2. How did you find the bug? How would you do this if the bug was subtle and there were hundreds of files and thousands of commits?
3. Could you have prevented this bug by doing something differently?

5.5 Question 5

Read the editorial Ten Simple Rules for Reproducible Computational Research and (in 150–250 words **total**) give a brief explanation of how git and GitHub would or would not be relevant to each rule.