# Get going with git

Mark Jones

2023-11-16

# Table of contents

# Introduction

The topic of version control is quite broad and ranging. Additionally, the git, github, github cli and github pages etc version control technologies can be daunting for newcomers to set up and apply. However, once you have things installed (and configured) and you have a basic grip on how it all fits together at a high level, then you can go a long way with some introductory knowledge.

I know, it is true that a little knowledge can be a dangerous thing, but it probably isn't quite as bad as complete ignorance.

This text, **Get going with git**, aims to provide introductory knowledge that is sufficient to get you up and running with git as a means for version control. The primary focus is on the core sets of functionality.

The text can be browsed online or downloaded as a word document for your reference and will be used as a companion for a training workshop.

# Preamble (don't skip)

## Background reading

There is no point reinventing the wheel, Jenny Bryant motivates the use of version control in the following paper: Excuse Me, Do You Have a Moment to Talk About Version Control?

## Objectives

> **Note**
>
> There is no getting around it. You are likely[1] going to find some[2] of this text hard and/or frustrating and/or/also painful.
> Your knowledge will progress in a stepwise fashion[3] through setup, core concepts and collaboration. Each of these will give you pain, but once you have practice them, they will become an intuitive, integrated and a comforting safety net.

The approach I have taken is purposefully (and unapologetically) applied and a fairly low level perspective on git. The rationale stems from experience of this being the best way to build a clear and coherent understanding. Once you have the fundamentals practiced, it is much easier to pick up any git client of your choosing (from the long list of clients) safe in the knowledge that you have some insight into what is going on under the hood and that if things do go horribly wrong (as they invariably do) you have some chance of being able to fix them.

The goal here is for you to gain understanding into the fundamental aspects of revision control and `git` in particular. Once you have that base, extending from there will be much easier for you.

The pace of the workshop will be based on the abilities of the audience but there is an expectation that the audience practices the skills in their own time. The workshop is likely to be run over a few sessions but the exact length is yet to be determined. It is mainly targeted at analysts that are new to revision control but with some effort is accessible to anyone.

[3]Actually with very high probability. Sorry.
[3]Probably a lot.
[3]Monotonically, I am sure. Ideally strictly positively.

## Command line interfaces

A command line interface (CLI) is a piece of software, supplied with the operating system, that you use to interact with your operating system via your keyboard rather than a mouse. Software that implements such a text interface is often called a command-line interpreter, command processor or a shell. You enter in commands as text and the system will do something, e.g. create a file, delete a file. Each of these commands is a piece of software that performs some function. `git` is one such command that facilitates access to a wide range of functions.

People outside of software development and (some) analysts find the command line esoteric and redundant. They are wrong on both counts. At one stage, the CLI was the only way to interact with a computer. The approach provides an extremely useful set of tools that pay dividends if you make the effort to learn even a small set of them.

Nearly universally, if I use the word *terminal* or *shell* I am referring to the operating system command line interface. Windows refers to its CLI as the *command prompt*, and in macOS they use *terminal*. For the purposes here, you can treat all these terms as synonomous.

If you do not know how to operate your operating system CLI then you need to address that to make the most out of this text.

### Why command line?

Because every other approach fundamentally rests on the git functionality that is exposed through the command line. While other approaches might abstract away from some of gits complexity, they also make some of the workings opaque. By learning the commandline functions, you focus on what you actually need and when you use another approach, you will understand what it is trying to achieve.

### Windows

You can do these tutorials to familiarise yourselves:

- Old, but with applicable coverage of the basics - Windows Command Line Tutorials
- LinkedIn course - Learning Windows Terminal

Please do at least one of the above.

### macOS

- How To Use Terminal On Your Mac
- What Is the Mac Terminal?
- Absolute BEGINNER Guide to the Mac OS Terminal

### Extra credit

- Learn the command line

> **Note**
>
> The reason I haven't mentioned Linux is simply because for Linux oriented people, it is reasonable to assume that this level of competence has already been met.

## Operating system management - environment variables etc

You will need to have some minimal technical competence in navigating and driving your computer. For example, you need to know how to set an environment variable under your operating system of choice. If you do not know how to set an environment variable, then you will need to figure it out otherwise you won't be able to use the tools that are going to be introduced here.

### Windows

You can do these tutorials to familiarise yourselves:

- Environment Variables : Windows 10
- How to Set Environment Variables in Windows 11

you should be able to set a user variable even if you do not have admin priviledges.

### macOS

- How to Set Environment Variables in MacOS
- PATH Variable (Mac)
- Use environment variables in Terminal on Mac

## Operating system management

Create a directory on your machine where we will store all the files for this workshop.

Simply go to the your `Documents` directory and create a sub-dir called `get-going-with-git` (you can call it what you want, but this is going to be the location of our scratchpad and temp work).

Throughout this text, if I say go to your local workshop directory, this is the location I want you to go to.

## GitHub

Follow part 1 of the instructions provided by Getting started with your GitHub account to create and configure your account.

> ⚠ Warning
>
> The part on **configuring 2-factor authentication** is absolutely mandatory, the rest of the 2-factor content can be skimmed. See Configuring two-factor authentication.

To use the USyd GitHub Enterprise Server, you will need a unikey. If you have a unikey, you should have access. If you go here you can confirm that you can login, but note that you will not require access to the Enterprise Server to complete this course.

## OneDrive

It would be beneficial if you can install the desktop application for OneDrive so that you have file system integration (i.e. so that you can see your OneDrive through your file explorer app).

You may need to go to IT Support to get them to install OneDrive for you, but it may already be installed.

# Part I

# Part 1 - Logistical elements

In this part we will get everything set up. Prepare yourselves, this could be the most frustrating part.

# 1 What is (this thing) called revision/version control?

## 1.1 The big picture

Before we do any setup, let's start with a big-picture. Version control is any practice that allows you to track and manage the evolution of a project.

Remember filenames like `document_v3.docx`, `document_v3_20230101.docx`, `document_v3_20230101_ts.docx`? That is a manual, informal and unstructure approached to version control. You can tighten up the filenaming standards, but this approach is still very limited. For example, the approach is case by case, simultaneous changes and merges are tedious, can involve double handling and are error prone.

A software implementation of revision control is a mechanism to understand a project/file history without having to used $N^q$ distinct filenames to identify the different versions. Git is one of many revision control systems. In data science, IT and tech fields, it is the de-facto standard. If you ever aspire to working in a technical role in one of those fields, you would need to learn git.

Irrespective, arguably, if you are collaborating, you should probably be using git.

When you *get going with git* you will need to think in terms of repositories, not to be confused with suppositories. Repositories are where you project code is held.

git is by design a distributed system, but do not worry about that right now. Basically, there are two types of repository - *local* and *remote*. The local ones are those that reside on your own machine. The remotes ones are off in the void somewhere, but they can be linked and interact via `git` like this:

Remote repositories are hosted by service providers, the most common being GitHub, GitLab and Bitbucket. We only deal with GitHub here. GitHub comes in a few varieties:

- GitHub Enterprise is hosted by the company called GitHub, see github.com. It is a commercial platform, but parts of it are made freely available.
- GitHub Enterprise Server is self-hosted; this is what USyd provides via https://github.sydney.edu.au/

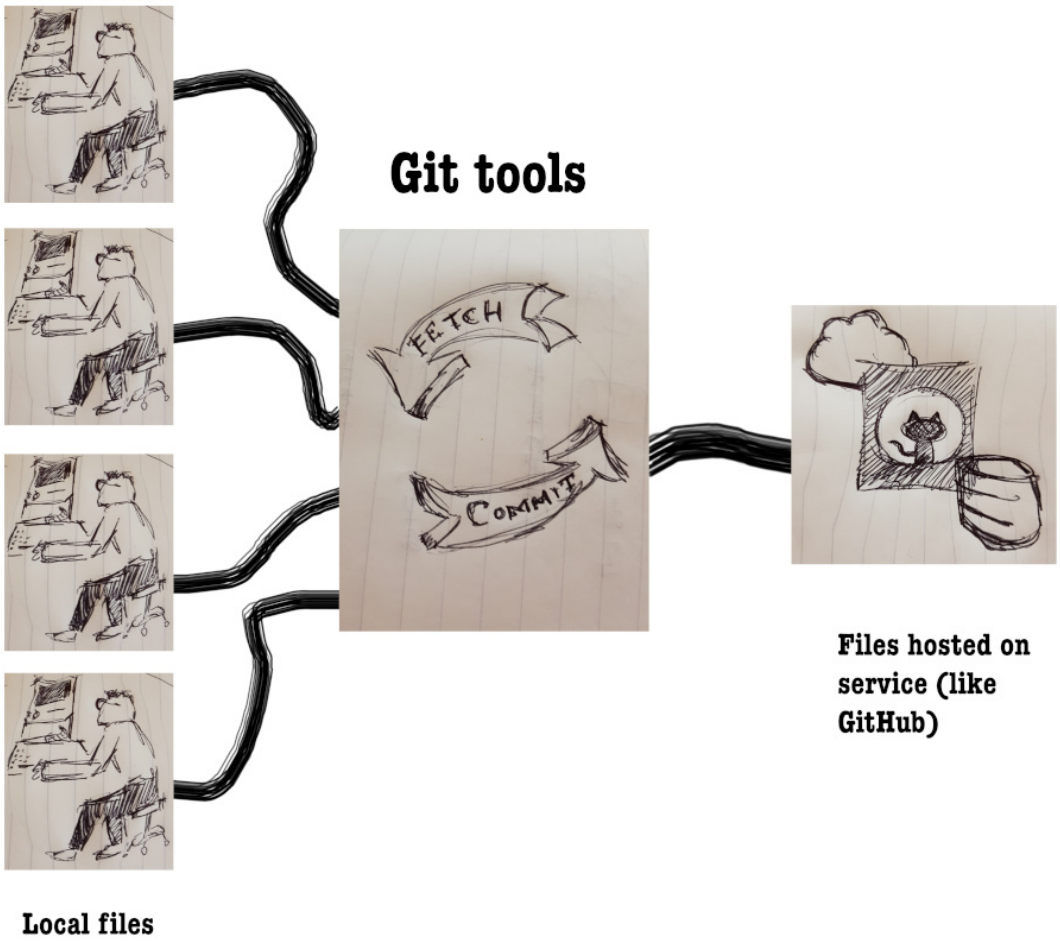


Figure 1.1: Big picture

In a nutshell, git provides a set of commands that allow you to manage the files that are retained in these local and remote repositories.

Again, I cannot sum it up better than Jenny, so please take the time to read it.

Excuse Me, Do You Have a Moment to Talk About Version Control?

# 2 Git install

Inevitably there are some installation tasks that we need to take care of before we proceed.

> ⚠ Warning
>
> The following steps can be a bit of a pain. Don't be disheartened, it will get better.

## 2.1 RStudio

Aside - is your instance of RStudio up to date? If not, update it. Ditto for R. Keep them both updated.

## 2.2 Install git

I am going to break this down into Mac and Windows because they are the two systems that most people use and the installation is somewhat different for each. If you are using Linux, you probably have no need to be reading this.

### 2.2.1 Mac OSX

First, do you have git installed already? Launch the `terminal` app (see the pre-requisites on the landing page if you do not know how to do this). In the terminal, type:

```
which git
```

which should show the location of the version of git in use. For me, its:

```
## /opt/homebrew/bin/git
```

If you have homebrew (see Section 2.2.2 below) installed, you can just type:

```
brew install git
```

and git will be installed, otherwise, follow the instructions below and then come back here.

Once git is installed run the `which git` command again and then run `git --version` which is shown (along with the output) below:

```
git --version
## git version 2.42.0
```

If you got here, then you have git installed. You can close down terminal, open it up again and then run the `git --version` command again to make certain that everything is ok.

### 2.2.2 Homebrew

In the previous section, you can see that the path output from the `which git` command includes `homebrew`. For macOS, `homebrew` is a package manager. This basically just lets you install and manage packages (applications) on your mac.

To use homebrew, you need to install it first. To do that, go here, then follow the instructions, which amount to going to the terminal and running the commands listed below.

Please go and read the landing page for homebrew before you proceed any further.

The first command ensures that pre-requisites are met, see here:

```
 xcode-select --install
```

if this has already been done you will get an error, or be asked to run Software Update. Generally, you can just move on to the next command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install
```

`curl` is a tool for transferring data from a url. It is usually stored under `/usr/bin` but if you are having issues then exporting the following might assist:

```
export HOMEBREW_FORCE_BREWED_CURL=1
```

which basically tells homebrew to use its own version of curl. After the above is complete, homebrew should be installed. Now you can run

```
brew install git
```

to install git for you.

For reference, here are a minimal set of commands for using homebrew (additional information can be found in the homebrew man pages).

Basic information on homebrew:

```
# Display the version of Homebrew.
$ brew --version
# Print Help Information
$ brew help
# Print Help Info for a brew command
$ brew help <sub-command>
# Check system for potential problems.
$ brew doctor
```

Keep your homebrew applications up to date:

```
# Fetch latest version of homebrew and formula
$ brew update
# Show formulae with an updated version available
$ brew outdated
# Upgrade all outdated and unpinned brews
$ brew upgrade
# Upgrade only the specified brew
$ brew upgrade <formula>
# Prevent the specified formulae from being upgraded
$ brew pin <formula>
# Allow the specified formulae to be upgraded.
$ brew unpin <formula>
```

The core commands for managing commandline applications are:

```
# List all the installed formulae.
$ brew list
# Display all locally available formulae for brewing.
$ brew search
# Perform a substring search of formulae names for brewing.
$ brew search <text>
# Display information about the formula.
$ brew info <formula>
# Install the formula.
$ brew install <formula>
# Uninstall the formula.
$ brew uninstall <formula>
```

```
# Remove older versions of installed formulae.
$ brew cleanup
```

Homebrew casks allow you to install GUI applications. Unless you are an advanced user, you will rarely need to use these, but for completeness:

```
# Tap the Cask repository from GitHub
$ brew tap homebrew/cask
# List all the installed casks .
$ brew cask list
# Search all known casks based on the substring text.
$ brew search <text>
# Install the given cask.
$ brew cask install <cask>
# Reinstalls the given Cask
$ brew cask reinstall <cask>
# Uninstall the given cask.
$ brew cask uninstall <cask>
```

### 2.2.3 Windows

The official site for the git windows binary download is https://git-scm.com/download/win.

If you haven't done all the install steps, download the 64-bit standalone installer, run it, agree to the conditions and license, choose the default location.

> **Note**
>
> There might be some merit in re-installing git just to make sure you have the latest version and that you select the correct setup.

Ensure that the following install components are chosen:

- windows explorer integration
- large file support

and accept any other defaults.

With the exception of the following, for any of the other prompts, just accept the defaults.

1. You will need to nominate a text file editor for editing commit messages and so on. Unless, you know what you are doing, I would advise just select the Windows Notepad application, you can reconfigure this later if you want to.

2. You should select to override the default branch name as `main`. The reason to do this is so that git aligns with GitHub (which uses `main` as its default branch).
3. For adjusting the PATH environment variable, ensure that you select `Git from the command line and also from 3rd-party software` which is the default.
4. Ensure that line ending conversion is set to `Checkout as-is, commit as-is`.
5. For the terminal emulator, select `Use Windows default console window`. This has some limitations but it is ok for an introduction. That said, if you are comfortable with the `bash` variant, do not let me stand in your way.
6. Ensure that `Git Credential Manager Core` is selected when prompted.

We will run through this install for someone in the group.

To keep git up to date, you will need to go to the above site and download and reinstall git.

Open the command prompt and type:

```
git --version
## git version 2.42.0
```

# 3 Git setup

## 3.1 Configuration for git

Per the sentiment of Fred Basset, you are now up but not quite running.


Figure 3.1: Fred Basset

One of the first things we need to do is to set a username and email address:

```
git config --global user.name "Fred"
git config --global user.email "fred.basset@comic-land.com"
```

You can list your configuration with

```
git config --global --list
```

The config is usually retained in a file in your home directory. It will be under `~/.git/config` or simply `~/.gitconfig`. If you look at this file, you will probably see that it is broken up into sections corresponding to `user`, `global` and so on. Don't worry overly about that right now, it is just something to be aware of.

> **Note**
>
> The tilde is just an abbreviation for the path to your home directory. If your home directory is `/Users/mark` then:
> `~/Documents = /Users/mark/Documents == TRUE`

We will get into the why later, but basically any interaction you have with git will be tied to your username and email address. This has obvious benefits if we want to be able to figure out who has done what, when and why.

Another tweak we will make is to the default branch name. Git always used to call this `master` by default. GitHub, in its infinite wisdom, appeared to make some judgement on the connotations of that default name and so changed it to `main`. This piece of config just makes Git consistent with GitHub.

```
git config --global init.defaultBranch "main"
```

# 4 GitHub setup

## 4.1 GitHub account

As noted in the pre-requisites for using this knowledge base, you will need to have GitHub account.

The two ways to interact with GitHub are HTTPS and SSH. They are both secure protocols with the goal of protecting your credentials and your work. SSH can be a little harder to set up but has some advantages over HTTPS It is also less commonly used and can be problematic when firewalls are involved, so we will stick to HTTPS.

A pre-requisite is to set up a Personal access token.

### 4.1.1 Personal access token

GitHub introduced personal access tokens a short while ago. Personal access tokens are basically a password with some bells and whistles.

1. Login to your GitHub account.
2. Open Creating a personal access token (classic) in a new tab in your browser and follow the instructions.
3. Set the expiry to at least several months into the future.

### 4.1.2 Create a private repo

This will just help us (you) with the subsequent steps. From your GitHub account, go up to the top right corner and select New -> Repository You should get a page that looks like this:

You can call it whatever you want (temp is fine because we will delete it at some point in the future) but make sure to (1) select the **Private** radio button and (2) initialise a **readme.md** file.

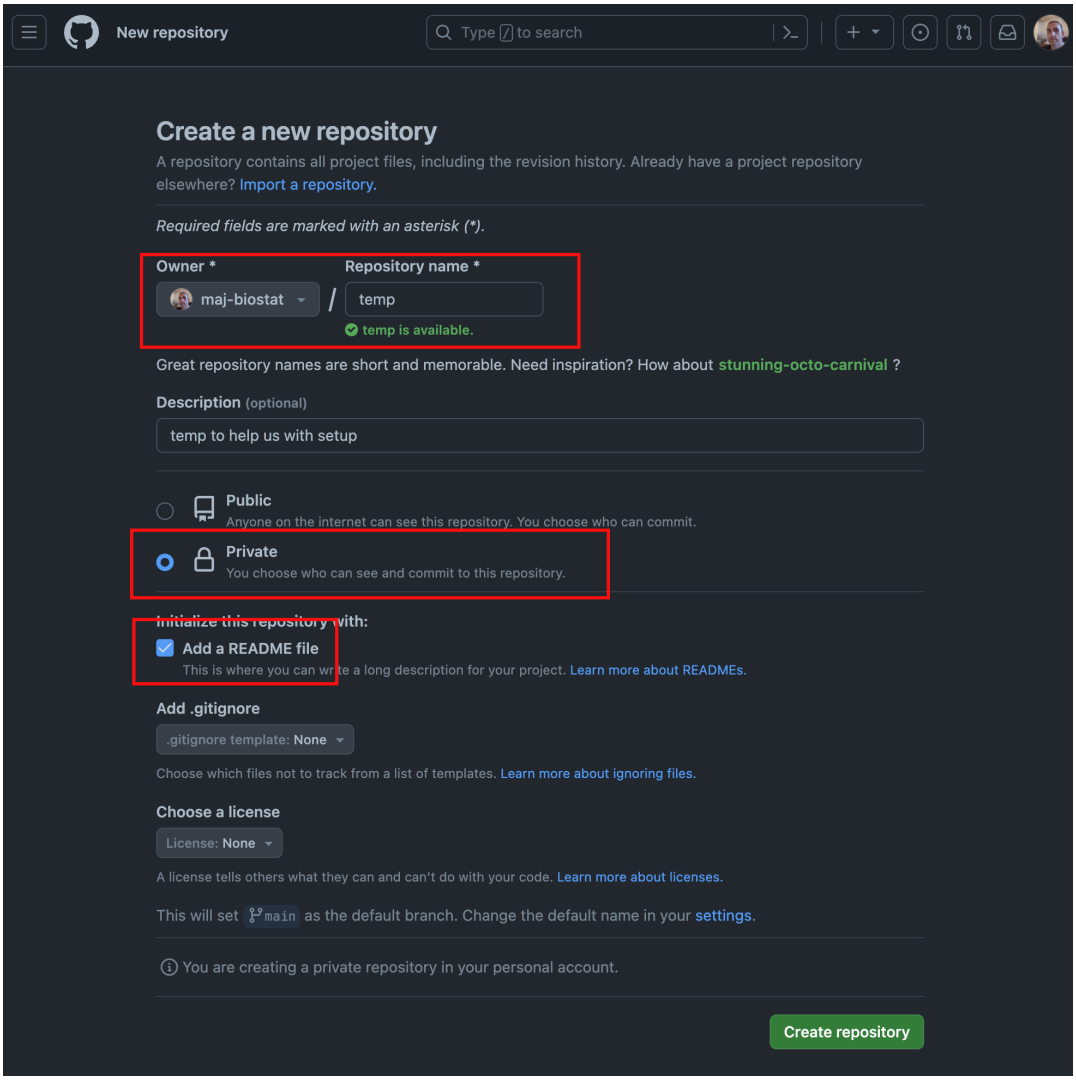


Figure 4.1: Create private repo

## 4.2 Git Credential manager

While SSH uses keys (along with a key agent which will normally already be on your machine) HTTPS relies on you to have some way to retain your credentials. That is, if you want to avoid having to enter your user name and token for every transaction you have with your GitHub repository, which is something you want to avoid. There is another alternative and that is to keep your credentials in a plain text file, but again that is strongly discouraged.

Enter the Git Credential Manager (GCM).

The GCM is a platform agnostic credential manager (in English, that translates loosely to a *password manager*). Once it's installed and configured, the Git Credential Manager is called by git and your interactions with GitHub become much more seamless.

The next time you clone an HTTPS URL that requires authentication, Git will prompt you to log in using a browser window. You may first be asked to authorize an OAuth app. If your account or organization requires two-factor auth, you'll also need to complete the 2FA challenge.

Once you've authenticated successfully, your credentials are stored in the macOS keychain (or Windows equivalent) and will be used every time you clone an HTTPS URL. From there on in, git should not require you to type your credentials in the command line again (unless you change your credentials).

### 4.2.1 GCM install

For Windows users it can be installed by selecting this option during the installation wizard, see Section 2.2.3, step 6. To display the current version, one of the following should help depending on which version of git you have:

```
git credential-manager version
```

```
git credential-manager-core --version
```

For macOS, use `homebrew` again, specifically:

```
brew install --cask git-credential-manager
## ==> Downloading https://formulae.brew.sh/api/cask.jws.json
## ######################################################################################
## ==> Downloading https://github.com/git-ecosystem/git-credential-manager/releases/downlo
## ==> Downloading from https://objects.githubusercontent.com/github-production-release-as
## ######################################################################################
```

```
## ==> Installing Cask git-credential-manager
## ==> Running installer for git-credential-manager with sudo; the password may be necessa
## Password:
## installer: Package name is Git Credential Manager
## installer: Installing at base path /
## installer: The install was successful.
##   git-credential-manager was successfully installed!
```

and then type:

```
git-credential-manager --version
```

to confirm it was installed and is accessible.

### 4.2.2 GCM demo

Below I demo the process by cloning a private repository from my GitHub account. You can use the `temp` repo that you just created, the URL will be something like `https://github.com/<your username>/temp.git` where you need to replace `<your username>`.

```
192-168-1-100:tmp mark$ git clone https://github.com/maj-biostat/wisca_2.git
Cloning into 'wisca_2'...
info: please complete authentication in your browser...
```

at this point the following window is launch by GCM:

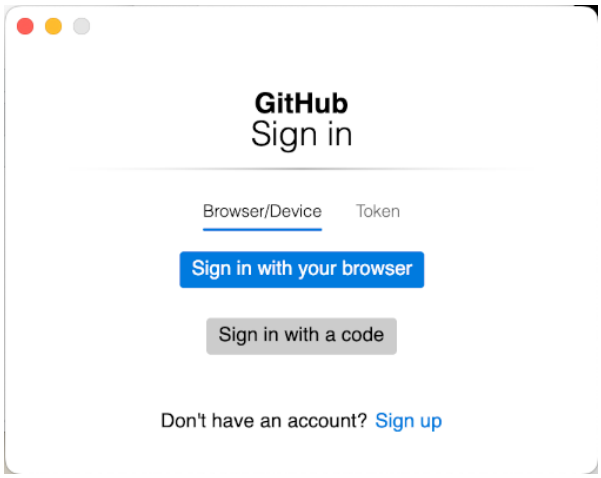


Figure 4.2: GCM

selecting `Sign in with your browser` the following will launch in your default browser (Chrome, Safari, etc)

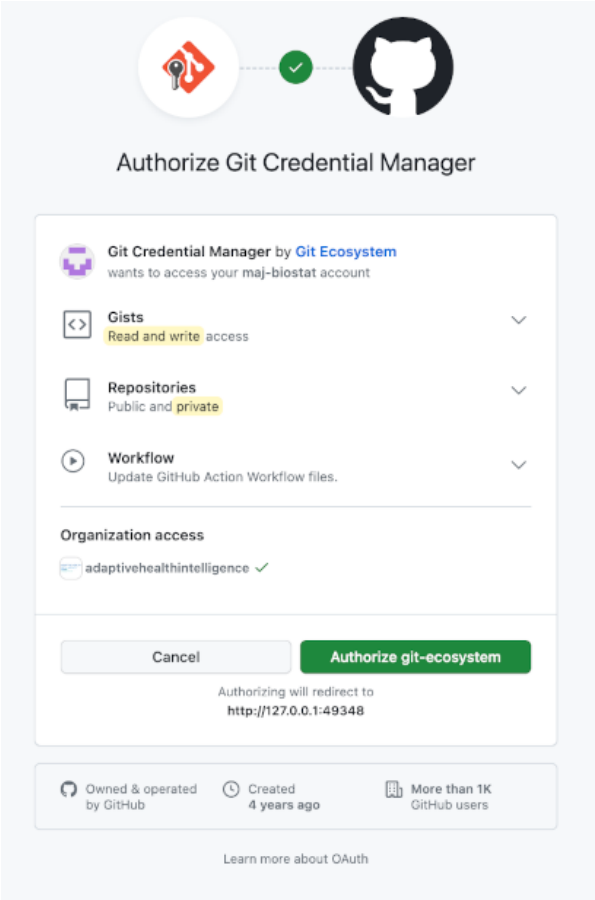


Figure 4.3: Sign in with browser

selecting `Authorize git-ecosystem` will result in

at which point you use the 2-factor authenticator tool (e.g. google authenticator, microsoft authenticator etc) to respond with an authentication code.

Looking back at the terminal, the following output can be observed, which details the repository being cloned.

```
remote: Enumerating objects: 297, done.
remote: Counting objects: 100% (297/297), done.
remote: Compressing objects: 100% (156/156), done.
remote: Total 297 (delta 148), reused 284 (delta 137), pack-reused 0
Receiving objects: 100% (297/297), 7.85 MiB | 2.13 MiB/s, done.
```

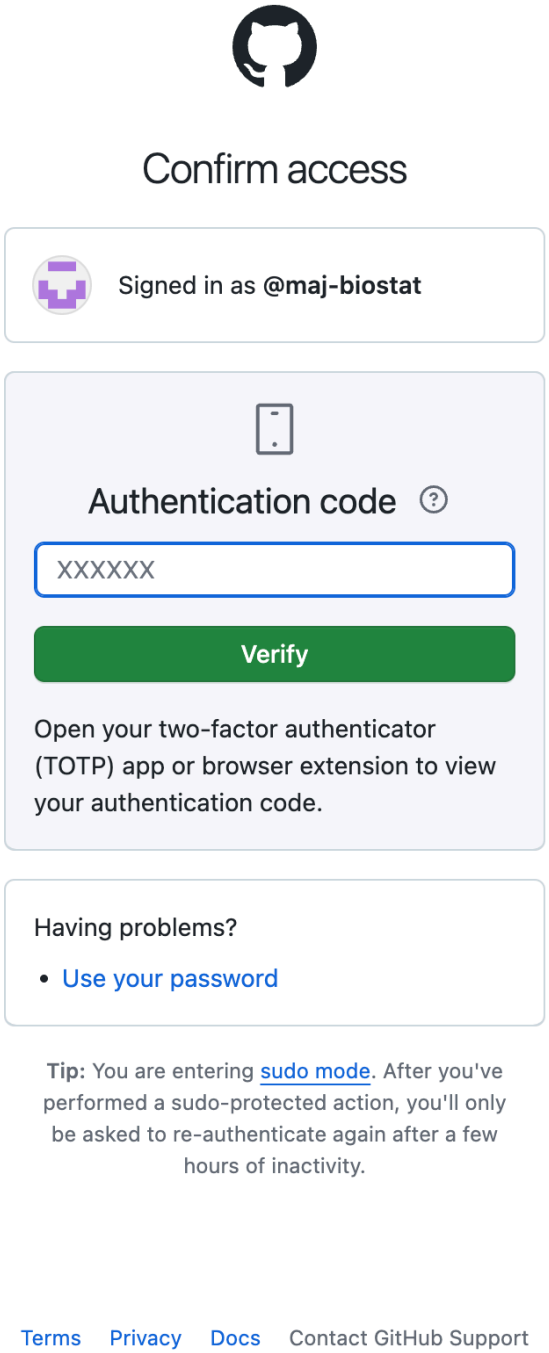


Figure 4.4: Sign in with browser

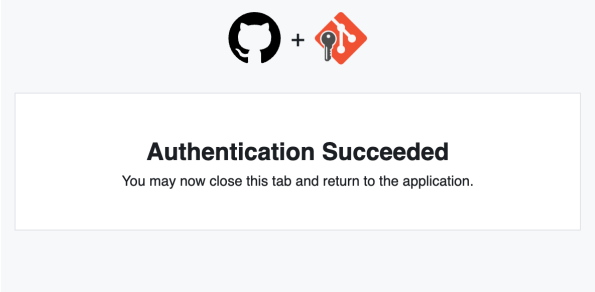


Figure 4.5: Auth success

```
Resolving deltas: 100% (148/148), done.
```

Finally, you will receive an email of this sort to the email account that you have linked through GitHub.

```
Hey maj-biostat!

A first-party GitHub OAuth application (Git Credential Manager) with gist, repo, and workf
Visit https://github.com/settings/connections/applications/0120e057bd645470c1ed for more i

To see this and other security events for your account, visit https://github.com/settings/

If you run into problems, please contact support by visiting https://github.com/contact

Thanks,
The GitHub Team
```

On repeating this process a second time, all the authentication works in the background and there will be no need to go through various authentication handshakes again.

The same process applies irrespective of whether you are using GitHub.com or the USyd GitHub Enterprise Server. However, it is adviseable to get this working in GitHub first and then work on getting it to work in the USyd GitHub Enterprise Server.

The transition from the old authentication approach has (so far) proved completely seemless for macOS. It will be interesting to see what happens for the Windows platform.

### 4.2.3 GCM configuration (advanced only)

You can view the current credential manager by running the following commands:

```
git config --local credential.helper
git config --global credential.helper
# /usr/local/share/gcm-core/git-credential-manager
git config --system credential.helper
```

> **Note**
>
> Of the local, global and system, the first one checks the local repository config, the second is your ~/.gitconfig, and the third is based on where git is installed. Note that only one of the `git config` commands returns a credential helper in the above example.

In some circumstances you may need to reconfigure things. If you have to start from scratch, the following may be useful:

```
git config --local --unset credential.helper
git config --global --unset credential.helper
git config --system --unset credential.helper
```

For windows uses check the contents of the credential manager. I believe that this can be accessed via Control Panel » All Control Panel Items » Credential Manager or by simply typing Credential Manager in the Windows task bar. Under generic credentials you should see the git entries.

## 4.3 GitHub CLI

For now, we can probably leave GitHub CLI setup, because I don't think we will get to use it. It will be used in a more advanced introduction.

In the day to day grind, having to deal with GitHub through its Web interface can be a little cumbersome. You can avoid having to regularly interact with GitHub through the browser by using the GitHub CLI. This tooling allows you to review, create and manage your repositories from the comfort of your commandline. You can think of it as an extension of git that allows you to invoke the GitHub specific functionality.

The extremely terse `gh` CLI manual can be found here.

For Windows users, you can pick up the latest Signed MSI executables from the release page.

For macOS, use `homebrew`:

```
brew install gh
## ==> Downloading https://formulae.brew.sh/api/formula.jws.json
```

```
## ##############################################################################################
## ==> Downloading https://formulae.brew.sh/api/cask.jws.json
## ##############################################################################################
## ==> Downloading https://ghcr.io/v2/homebrew/core/gh/manifests/2.37.0
## Already downloaded: /Users/mark/Library/Caches/Homebrew/downloads/331c0b76fd34aa97342ef
## ==> Fetching gh
## ==> Downloading https://ghcr.io/v2/homebrew/core/gh/blobs/sha256:a8c21e08d77963c2d12102
## Already downloaded: /Users/mark/Library/Caches/Homebrew/downloads/d0e6a3f8f7a4b138b3648
## ==> Pouring gh--2.37.0.arm64_ventura.bottle.tar.gz
## ==> Caveats
## Bash completion has been installed to:
##   /opt/homebrew/etc/bash_completion.d
## ==> Summary
##   /opt/homebrew/Cellar/gh/2.37.0: 191 files, 44.2MB
## ==> Running `brew cleanup gh`...
## Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
## Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
```

### 4.3.1 GitHub CLI authentication

In order to make use of `gh` we need to go through another round of authentication setup. To do this, go to the terminal and run:

```
gh auth login
## ? What account do you want to log into? GitHub Enterprise Server
## ? GHE hostname: github.sydney.edu.au
## ? What is your preferred protocol for Git operations? HTTPS
## ? Authenticate Git with your GitHub credentials? Yes
## ? How would you like to authenticate GitHub CLI? Login with a web browser
```

For additional information, see gh auth –help.

In order to use `gh` with github.com directly you need to authenticate for that platform too. Repeat the above, but now the responses look like this:

```
gh auth login
## ? What account do you want to log into? GitHub.com
## ? What is your preferred protocol for Git operations? HTTPS
## ? Authenticate Git with your GitHub credentials? Yes
## ? How would you like to authenticate GitHub CLI? Login with a web browser
```

You are nearly set. You can verify that what you have configured worked via:

```
gh auth status
## github.sydney.edu.au
##     Logged in to github.sydney.edu.au as mjon7053 (keyring)
##     Git operations for github.sydney.edu.au configured to use https protocol.
##     Token: gho_************************************
##     Token scopes: gist, read:org, repo, workflow
##
## github.com
##     Logged in to github.com as maj-biostat (keyring)
##     Git operations for github.com configured to use https protocol.
##     Token: gho_************************************
##     Token scopes: gist, read:org, repo, workflow
```

However, for `gh` to work with the desired host you need to set an environment variable to tell `gh` which platform to use. On macOS, you can set this up easily with the following entries in the `.profile` shell initialisation script (or `.bash_profile` for those inclined).

```
gh-ent() {
  export GH_HOST=github.sydney.edu.au
}

gh-std() {
  export GH_HOST=github.com
}
```

On Windows, I have no idea how you are supposed to do the above in an easy manner. You may just have to resort to running

```
set GH_HOST=github.sydney.edu.au
```

or

```
set GH_HOST=github.com
```

each time you want to switch.

Now (on macOS) when you want to interrogate github.com repositories we can use the following commands.

> **Note**
>
> Do not worry about the meaning of the commands, this is just to establish that we have configured things correctly.

```
gh-std
gh repo list

## Showing 30 of 185 repositories in @maj-biostat
##
## maj-biostat/misc-notes                          info for manjaro/arch linux setup
## maj-biostat/wisca_2                             Revised approach to antibiogram
## maj-biostat/motc.run
## maj-biostat/motc.sim                            Simulation for motivate c trial
## maj-biostat/motc.stan                           Stan models for motc
## maj-biostat/quarto_demos_basic                  Demo using Quarto to render to word docu
## maj-biostat/BayesDRM                            Dose response models in stan
## maj-biostat/motc.modproto
```

and for the USyd Enterprise GitHub Server, use:

```
gh-ent
gh repo list

## Showing 12 of 12 repositories in @mjon7053
##
## mjon7053/motc-mgt            Monitoring statistics for Motivate-C study
## mjon7053/fluvid.analyses     Analyses for fluvid coadministration study (COVID19 + FLU)
## mjon7053/motc.sap
## mjon7053/motc-sim-report     Motivate-C simulation report
## mjon7053/roadmap-notes       Notes relating to the ROADMAP project.
## mjon7053/mjon7053.github.io
```

# Part II

# Part 2 - Fundamentals

Now we will make a start with git. Initially we will focus on using git within the confines of your local machine. Yes, that's right, we won't be using GitHub until the very end of this part. The point of this is to give you a chance to get to grips with the basic ideas. After the main concepts are bedded in, we will move to thinking about GitHub, which is a whole new beast.

We will talk about;

- Repositories, concepts and how to set them up and configure.
- How to interact with repositories on a basic level.
- How to compare and assess the present state of our repository
- Branches, what they are and what the are used for as well as setting them up
- Merging, what it is and how to do it

# 5 Repositories

## 5.1 Git repositories

A repository is the most basic component of git. It is where the history is retained in terms of commits (it's not correct but for the time being you can think of these as versions or snapshots of the files/repository).

Repositories can be public or private, can involve single people or multiple collaborators and can be stored locally (on your personal computer), be located/stored in the cloud (in the cloud as in hosted by a service provider like GitHub), or on a physical server (a basic file server will suffice in most cases), or just on your local machine.

Using the functions provided through git, you can create and configure repositories, add or remove files, review history of the files in the repository and much more.

### 5.1.1 Initialisation

Let's initialise a new repository. Run the following on your machine:

```
# Change dir to the local workshop directory,
# e.g. cd ~/Documents/get-going-with-git
mkdir first-repo

cd first-repo

git init
## Initialized empty Git repository in /Users/mark/Documents/project/misc-stats/first-repo
git status
## On branch main
##
## No commits yet
##
## nothing to commit (create/copy files and use "git add" to track)
```

> **Note**
>
> The repository is implemented by a hidden directory called `.git` that exists within the project directory and contains all the data on the changes that have been made to the files in the project. You should never touch this directory nor its contents.

If you received the output detailed above (or something very similar to it) then congratulations, you initialised a git repository.

If you have configured your file explorer to show hidden files, you will notice that the `first-repo` directory now contains a `.git` sub-directory. If you are running Windows, then depending on what version you have, you may be able to run `ls -la` or `dir /a` to list all files (i.e. including hidden files) within a directory.

Again, the `.git` directory contains everything related to the repository.

> **Warning**
>
> **If you delete the `.git` directory, you will erase all of your commits. So, do not delete it unless you are absolutely certain this is what you want.**

> **Note**
>
> 1. You can also create a repository from a pre-existing directory that has already got an established file structure and files. The process is exactly the same, just change to the directory that you want to add to version control, and run `git init`.
> 2. When you create a new project in Rstudio, you can select to initialise a new git repository. Underneath the covers, RStudio is simply invoking `git init`.

### 5.1.2 Repository structures

Before we start adding files to the new repository, you need to be aware of a few concepts.

There are three main structures within the context of a repository:

1. Working directory
2. Staging area
3. Commit history

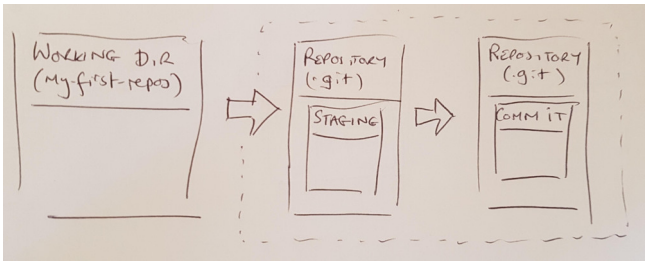


Figure 5.1: Repository structures

#### 5.1.2.1 Working directory

Is the usual files and sub-directories within your project directory. You introduce, update, rename, delete files and direcotries in this area and they form your project (an analysis, a set of documentation etc). When you first create a file or directory within the working directory, it is not yet under version control. Such files are referred to as **untracked files**.

#### 5.1.2.2 Staging area

Is a special space to where you **stage** (aka list or identify) the files that are to be committed as a new version under version control.

#### 5.1.2.3 Commit history

After staging files, they are **committed** to the repository. Once committed, files (and directories) are under version control and are referred to as **tracked files**. A commit is simply a version, but you could also think of it as a transaction with the repository if it helps your mental model. Changes to committed files are monitored and new updates to files can be committed to the repository as work on the project progresses. Every time you commit files, the **commit history** is saved.

# 6 Stage, commit and history

## 6.1 Adding files to projects

Let's go through the motions of introducing files for a project. Open a text editor, enter the following contents (or download the readme.md file from the applicable section under Chapter 14) and save the file as `readme.md` in the `first-repo` directory.

```
# first-repo

A demo markdown file for the git workshop.
```

Ditto for the following and save the file as `hello.R` in the `first-repo` directory.

```
cat("Enter a string please: ");
a <- readLines("stdin",n=1);
cat("You entered")

str(a);
cat( "\n" )
cat(a, file = "log.txt")
```

Run the R script from the terminal by entering this text:

```
Rscript hello.R
```

Now from the terminal in the `first-repo` directory:

```
git status
## On branch main
##
## No commits yet
##
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##  hello.R
```

```
##   readme.md
##   log.txt
##
## nothing added to commit but untracked files present (use "git add" to track)
```

We see that there are three untracked files, two of which we will ultimately want to store in the git repository. In contrast to the newly initialised repository as shown in Section 5.1.2 we now have the following:

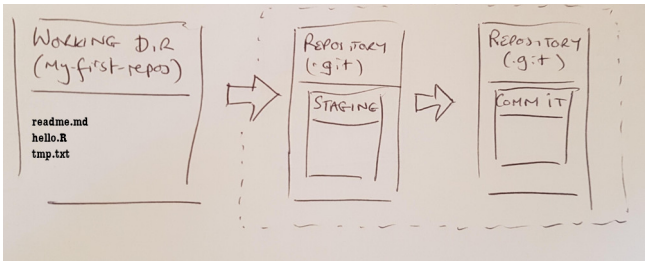


Figure 6.1: Add file to working directory

The above is idealised. You may encounter the situation where you have many files, a number of which you have no intention of tracking under version control. You can ignore these files by creating a `.gitignore` file, which tells git which files it should ignore.

Create a new text file with the following content and save the file as `.gitignore`. This is a special configuration filename that git recognises. The `.` at the front of `gitignore` is important.

If you missed the `.` or mistakenly added a `.txt` extension then the `.gitignore` functionality will not work.

```
# .gitignore file contains files
# that the repository will ignore
log.txt
*.txt
```

If you list (`ls -la`) the files in the `first-repo` directory, you should see the following (or something very similar):

```
192-168-1-100:first-repo mark$ ls -la
total 32
drwxr-xr-x  7 mark  staff  224  7 Nov 10:15 .
drwxr-xr-x  8 mark  staff  256  7 Nov 10:12 ..
drwxr-xr-x  9 mark  staff  288  7 Nov 10:15 .git
```

```
-rw-r--r--@ 1 mark  staff   81  7 Nov 10:15 .gitignore
-rw-r--r--@ 1 mark  staff  126  7 Nov 10:13 hello.R
-rw-r--r--  1 mark  staff    4  7 Nov 10:14 log.txt
-rw-r--r--@ 1 mark  staff   59  7 Nov 10:13 readme.md
```

Run `git status` again and note that the `log.txt` file no longer registers with git.

```
git status
## On branch main
##
## No commits yet
##
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##  .gitignore
##  hello.R
##  readme.md
##
## nothing added to commit but untracked files present (use "git add" to track)
```

## 6.2 Commit process

Now we want to add the new file to the repository. The steps are simple:

1. Add the file (or files) that we want to include in the repository to the staging area
2. Commit the staged files

### 6.2.1 Staging

To stage files (put them into the staging area) run the commands:

```
git add hello.R readme.md .gitignore
```

Note that we added the `.gitignore` file as well as the `hello.R` and `readme.md` files at the same time. We did not add the `log.txt` file. Now run

```
git status
## On branch main
##
## No commits yet
```

```
##
## Changes to be committed:
##   (use "git rm --cached <file>..." to unstage)
##  new file:   .gitignore
##  new file:   hello.R
##  new file:   readme.md
```

We can see that no commits have occurred but that we have staged the files that we want to add to the repository.

What happens if we accidentally add a file that we did not want to add (the `-f` says we want to add a file that is included in the `.gitignore` list)?

```
git add -f log.txt
```

if you run `git status` you will see that `log.txt` is also staged. To remove `log.txt` from the staged area:

```
git reset log.txt
```

The picture now looks like this.

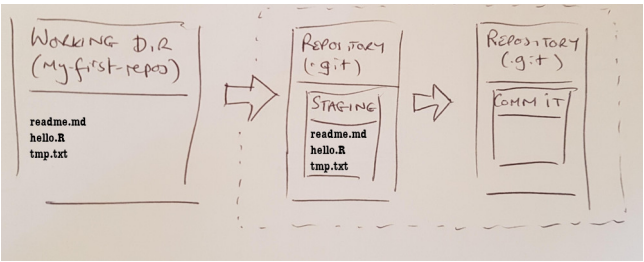


Figure 6.2: Add file to staging area

### 6.2.2 Commit

To commit the files that have been staged:

```
git commit -m "First commit"
## [main (root-commit) 728d107] First commit
##  3 files changed, 15 insertions(+)
##  create mode 100644 .gitignore
##  create mode 100644 hello.R
##  create mode 100644 readme.md
```

the `-m` flag is necessary or you will be prompted to provide a message. When you make a commit, you need to provide a message that describes the nature of the changes.

What is the weird stuff that is output prior to the commit message ([main (root-commit) 728d107]) in the commit history? It is a unique hash code that identifies this specific version of the project. Note, you will have a different hash code (and that is fine).

When we run `git status` we see that the repository is up to date with the working area files. We also see that the files have been removed from the staging area.

```
git status
## On branch main
## nothing to commit, working tree clean
```

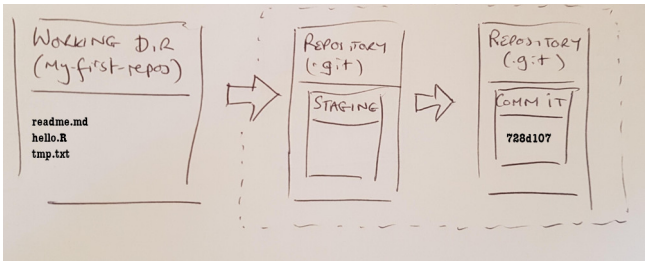


Figure 6.3: Commit files to history

## 6.3 Exercises

**Exercise 6.1.** Create a new R script in the working directory, it can contain anything you like. If you are lost, just use:

```
library(survival)
print("My script")
```

and save it as `myscript.R`.

**Exercise 6.2.** Add the new script to the staging area by following the approach introduced in Section 6.2.1, ensuring that you review the status.

**Exercise 6.3.** Commit the staged files to the repository by following Section 6.2.2 making sure that you record a message for your commit.

**Exercise 6.4.** Edit the `readme.md` (use notepad or rstudio) file adding a new line with some arbitrary text. Stage the file and commit.

## 6.4 Tracking commit history

One of the most notable features of revision control is that you can review your project file history. The simplest way to do this is with `git log` which will report all of the commits in reverse chronological order. You can see

- who made the commits
- when they were made and why (the commit messages)
- the hash code associated with project version at each commit
    - note that the full hash is reported whereas previous a truncated version is shown

The commit followed by `(HEAD -> main)` shows what part of the history our working directory currently reflects.

Here is an example (your repository will look different but that is ok)

```
git log

## commit 327170a6bc4d39463c4cfbc0f257420496642cb5 (HEAD -> main)
## Author: Mark <mark.jones1@sydney.edu.au>
## Date:   Tue Nov 7 10:25:23 2023 +0800
##
##     Minor edit
##
## commit b078716e80498c2fa7abfb8ae27b204b2dc603d8
## Author: Mark <mark.jones1@sydney.edu.au>
## Date:   Tue Nov 7 10:24:58 2023 +0800
##
##     New file
##
## commit 0cd2d52e989059a61315525a3488e06d22cd04a5
## Author: Mark <mark.jones1@sydney.edu.au>
## Date:   Tue Nov 7 10:23:23 2023 +0800
##
##     First commit
```

You can format the logs in a variety of ways. For a more condensed view you can use the `--oneline` flag:

```
git log --oneline
## 327170a (HEAD -> main) Minor edit
## b078716 New file
## 0cd2d52 First commit
```

If you want the commit history for the last n commits, or between specific dates, or by author or even via searching for a specific string in the message you can run the following

```
git log -n 2
git log --after="2013-11-01" --before="2023-10-15"
git log --author="Mark\|Fred"
git log --grep="first" -i
```

Try them.

The first restricts to the last two commits, the second returns commits between mid Oct and the start of Nov, the second returns commits made by Mark or Fred and the third returns any commits where the word first was included in the message text (ignoring case).

The log command is powerful and it lets you see who updated the files, when they made the update and why they did it. Obviously, this has less utility when you are working on a repository in isolation but it still does have value (especially to your future self). For example, you might simply want to review when specific changes were made to the files or you might want to pick up some update that has been removed from the code and reintroduce it.

When you are working on a repository in collaboration (see later) the value of the logs increases many fold as a way to be able to understand the evolution of the project and to work out who you need to contact if you think a problem has been introduced.

A particularly useful log command is, which gives you a visual depiction of the repository structure. Here is an example from a small repo that we will look at later.

```
git log --oneline --graph --all

*   a904f44 (HEAD -> main, origin/main) Merge branch 'feat001'
|\
| *   9c949d0 (feat001) Resolved merge conflict by retaining majbiostat implementation.
| |\
| |/
|/|
* | 14b1c35 Csum implementation
* | cc35ec0 Readme for landing page
| * c615c0c Implementation of cumsum alias
|/
* 8e9007a Initiall commit
```

To establish what files were included in any given commit, you can use `git show`:

```
git show --name-only 0cd2d52
## Author: Mark <mark.jones1@sydney.edu.au>
## Date:   Tue Nov 7 10:23:23 2023 +0800
##
##     First commit
##
## .gitignore
## hello.R
## readme.md
```

# 7 Reviewing differences

## 7.1 Comparisons with the working directory

Git allows you to compare different versions of files that exist in the repository. In its vanilla form, the difference functionality compares the differences in a file (or files) in the working directory to the repository version.

Update the contents of the `hello.R` script to match what follows.

```
cat(paste0("What is your name?\n"));
nme <- readLines("stdin",n=1);
cat(paste0("Hi ",nme, " enter a string please: \n"));
a <- readLines("stdin",n=1);
cat("You entered the following string: ")
cat(paste0(a, "\n"));
cat(a, file = "log.txt")
```

Similarly, edit the `readme.md` file as below.

```
# first-repo

A demo markdown file for the git workshop.

A new line for testing.

Contains standalone R scripts.
```

Running `git status` you will see that the working directory uncommitted changes

```
git status
## On branch main
## Changes not staged for commit:
##   (use "git add <file>..." to update what will be committed)
##   (use "git restore <file>..." to discard changes in working directory)
##  modified:   hello.R
```

```
##  modified:   readme.md
##
## no changes added to commit (use "git add" and/or "git commit -a")
```

If we want to know all the changes that were made, we can run `git diff` but let's look at the differences on a file by file basis, comparing the old with the new.

```
git diff readme.md
## diff --git a/readme.md b/readme.md
## index 7501d7c..6929e2c 100644
## --- a/readme.md
## +++ b/readme.md
## @@ -4,5 +4,7 @@ A demo markdown file for the git workshop.
##
##  A new line for testing.
##
## +Contains standalone R scripts.
## +
```

You interpret the above as follows.

- Anything prefixed with `-` belongs to the old file and anything prefixed with `+` belongs to the new.
- The section labelled with `@@` gives you some context as to where the change has happened. In this case we can see that the text *A minor revision* has been removed and replaced with the text *The main script implements a loop to capture input from a user.*

Now compare the working version of `hello.R` with the repository version, but this time look at the word by word differences:

```
git diff --word-diff hello.R
## diff --git a/hello.R b/hello.R
## index 0c6d38c..5f09b06 100644
## --- a/hello.R
## +++ b/hello.R
## @@ -1,7 +1,9 @@
## [-cat("Enter-]{+cat(paste0("What is your name?\n"));+}
## {+nme <- readLines("stdin",n=1);+}
## {+cat(paste0("Hi ",nme, " enter+} a string please: [-");-]{+\n"));+}
## a <- readLines("stdin",n=1);
## cat("You [-entered")-]
##
```

```
## [-str(a);-]
## [-cat( "\n" )-]{+entered the following string: ")+}
## {+cat(paste0(a, "\n"));+}
## cat(a, file = "log.txt")
```

The diffs can take a bit of getting used to and some alternative tools are available that we will put to use in due course (see Section 16.1). For now, we will just deal with the commandline output.

Once satisfied that the changes are (at the very least) benign, stage and commit the edits in the usual way:

```
git add hello.R readme.md
git commit -m "Revise approach in capturing user input"
```

## 7.2 Comparisons with staged files

If you want to restrict your attention to the differences that will be made to a repository due to committing staged files, you can use `git diff --cached`.

## 7.3 Comparisons across commit versions

Once working directory changes have been committed to the repository it is still possible to review the differences between commit.

The most common difference that is of interest is that between the last two commits. To achieve this run

```
git diff HEAD 327170a
```

To inspect differences between any commits, you simply supply the commit hashes that you want to compare:

```
git diff b078716 0cd2d52
## diff --git a/myscript.R b/myscript.R
## deleted file mode 100644
## index a12204c..0000000
## --- a/myscript.R
## +++ /dev/null
## @@ -1,3 +0,0 @@
```

```
## -library(survival)
## -print("My script")
## -
```

If you want to restrict attention to a particular file, just add the filename that you want to compare to the end of the command

```
git diff HEAD 327170a readme.md
## diff --git a/readme.md b/readme.md
## index 6929e2c..7501d7c 100644
## --- a/readme.md
## +++ b/readme.md
## @@ -4,7 +4,5 @@ A demo markdown file for the git workshop.
##
##  A new line for testing.
##
## -Contains standalone R scripts.
## -
```

**Exercise 7.1.** Can you remember what the command to show what the contents of a commit was?

# 8 Branches

## 8.1 What is a branch?

> ⚠ Warning
>
> Time to step it up a notch. With branching, practice and repitition is key to understanding. Try not to panic.

Branching is a mechanism that allows you to undertake work independently from the main (or base) content of a project while allowing you to keep in sync and also giving you the option of bringing your work back into the main content. The process allows multiple pieces of work by multiple people to be progressed independently within the same project. You might diverge because you have want to take the project in a new direction altogether or because you are working on a bug that is going to break things more before the fix is realised. Ideally, the work on your branch will be homogenous - it will, more or less, relate to one thing, one feature, one bug etc.

When you run `git status` you saw the text - *On branch main* as part of the output.

```
pwd
# /Users/mark/Documents/project/misc-stats/first-repo
git status
## On branch main
## Your branch is up to date with 'origin/main'.
##
## nothing to commit, working tree clean
```

There are a range of strategies for the way that a group uses branching, but for our purposes, you can think of `main` as your default branch.

Similarly, when you ran `git log` the current commit reported is suffixed with *(HEAD -> main)*. Both of these were a reference to the version on the branch that is currently linked to the working directory. It is where you presently are in the commit history.

```
git log --oneline
## 516ad1a (HEAD -> main, tag: v4.0, origin/main, rm, analysis-04) Code review comments
## a286918 Analysis 4
## 538330d Merge remote-tracking branch 'refs/remotes/origin/main'
## 49c68c2 Mark has also added detail to the readme.md
## 3a3dd8d Testing Sylvie
## 2829471 Merge branch 'analysis-03'
## be09457 (tag: v3.0, analysis-03) Finished surv
## 53115b6 minor
```

For example, initially there is a singular progression of the project, but at some point you will want to create a release for a software product, or a piece of documentation or an analysis. Later you may want to revise the release due to changes in project direction, new data, bugs etc. You use branches to facilitate this process in a logical and coherent way. Typically, you branch off from the default branch of a repository, but you can branch from any branch that exists.

In the examples encountered so far, the branching we have encountered is just a stem (specifically, the *main* branch). Below is a common type of representation of the kind of branch that we have dealth with so far.

The circles represent each commit, which would refer to changes in one or more files. The repository has gone through a series of commits (1, 2, 3, 4 etc.) and the working directory is currently looking at the repository version 4.

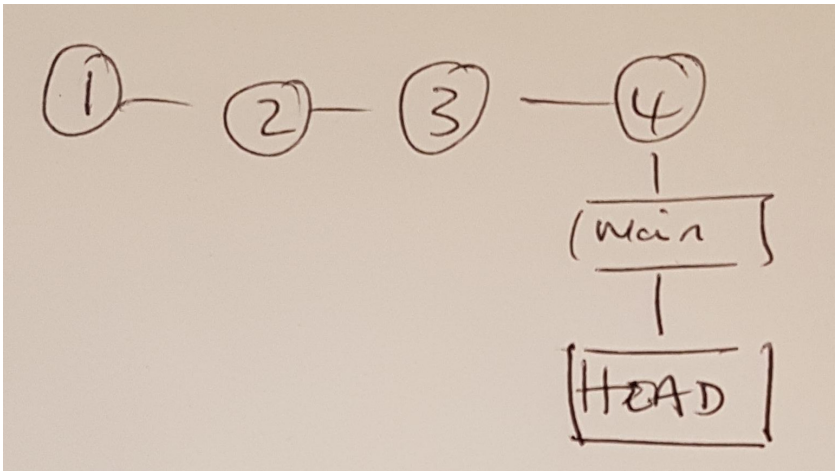


Figure 8.1: Branch more like a stem

The arrow pointing at 4 is the current *HEAD* of the repository. HEAD is a special concept in git. It answers the question "What am I currently looking at?"

## 8.2 Time travel

The concepts associated with branching are easiest understood by demonstration and experimentation.

When you make a new commit to git, the branch reference is updated to point to the new commit. When you move to a new branch, the HEAD reference is updated to point to the branch that you switched to.

Go to the Chapter 14 resources page and save the `branching.R` file to your `first-repo` directory.

The file contains the following:

```
# R script to demo branching
suppressPackageStartupMessages(library(data.table))
suppressPackageStartupMessages(library(survival))
suppressPackageStartupMessages(library(rtables))
suppressPackageStartupMessages(library(ggplot2))

message("--------------------------------")
message("NOW WE WILL MOVE ONTO BRUNCHING\n")
message("--------------------------------")


# Data generation  ------------------
set.seed(1)

N <- 4000
d <- data.table(
  id = 1:N,
  u = rbinom(N, 1, 0.5)
)

d[u == 0, x := rbinom(.N, 1, 0.2)]
d[u == 1, x := rbinom(.N, 1, 0.8)]

b0 <- 3
b1 <- 1
b2 <- -2
b3 <- -1
e <- 1
w_cens <- 4
```

```
# Continuous outcome
d[, mu := b0 + b1 * x + b2 * u + b3 * x * u]
d[, y := rnorm(.N, mu, 1)]

# Binary outcome
d[x == 1, z := rbinom(.N, 1, 0.7)]
d[x == 0, z := rbinom(.N, 1, 0.3)]

# Survival outcome
# Median tte -log(0.5)/0.6 vs -log(0.5)
d[x == 1, w := rexp(.N, 0.6)]
d[x == 0, w := rexp(.N, 1.0)]
d[, evt := as.integer(w < w_cens)]
d[evt == 0, w := w_cens]

# Labels
d[x == 0, arm := "FBI"]
d[x == 1, arm := "ACTIVE"]

d[u == 0, age := "< 50 years"]
d[u == 1, age := ">= 50 years"]

# Descriptive summary -------

message("\nDESCRIPTIVE SUMMARY:\n")

lyt <- basic_table() %>%
  split_cols_by("arm") %>%
  summarize_row_groups() %>%
  analyze("y", mean, format = "xx.x")

build_table(lyt, d)

# Analyses --------

message("\n\nANALYSIS OF CONTINOUS OUTCOME (UNSTRATIFIED):\n")

lm1 <- lm(y ~ x, data = d)
summary(lm1)
```

Run the script:

```
Rscript branching.R
```

Imagine this was the first analysis for a project and will be sent to the clients. The completed work represents a milestone for the project so we stage and commit the file and then create a tag for it.

```
git tag -a v1.0 -m "Analysis 1"
```

We continue with the work for the next deliverable completing a secondary analysis on the binary outcome `z`. Add the following code to the end of the `branching.R` script, re-run with `Rscript branching.R` and then commit the file to the repository.

```
message("\n\nANALYSIS OF BINARY OUTCOME (UNSTRATIFIED):\n")

lm2 <- glm(z ~ x, data = d, family = binomial)
summary(lm2)

pr <- predict(lm2, newdata = data.table(x = 0:1), type = "response", se = T)
d_fit <- data.table(
  arm = c("PBO", "ACTIVE"),
  x = 0:1,
  pr_z = pr$fit,
  pr_z_lb = pr$fit - 2 * pr$se.fit,
  pr_z_ub = pr$fit + 2 * pr$se.fit
)
```

We are not finished with our second deliverable, but at this point we realise that the initial analysis that was sent to the client was incorrect. As you may have spotted, we should have run a stratified analysis due to the presence of a confounder. We urgently need to re-issue the corrected analysis to the client. Bummer.

In contrast to the minor change above, in real life we might be much further along with this next deliverable, which may be vastly more complex than what I have illustrated above. For example, we may have introduced new files, restructured the original analysis, added functionality etc.

While we could go through the process of winding back all the changes, with revision control we do not have to because we can rewind to any point. We'll go over one approach to this using git. It isn't the ideal process to tackle the problem[1] but it was chosen to highlight a few points.

[1]The start of an alternative could be `git revert --no-commit <hash>..HEAD; git commit`.

### 8.2.1 Commit

First thing to do is to check that your code is running ok and then commit any files that have not yet been committed to the repository. Not doing so will cause you some major headaches, so best advice is to not forget to do this.

```
git status
git add braching.R
git commit -m "Commit of files part way through development"
```

### 8.2.2 Rewind

We can rewind the state of our working directory to the time at which the deliverable was made. We do this by first using `git log` to find the commit hash or we can just use the tag that we set for the release. Using the tag is more convenient so let's do that.

```
git log --oneline
git checkout v1.0
## Note: switching to 'v1.0'.
##
## You are in 'detached HEAD' state. You can look around, make experimental
## changes and commit them, and you can discard any commits you make in this
## state without impacting any branches by switching back to a branch.
##
## If you want to create a new branch to retain commits you create, you may
## do so (now or later) by using -c with the switch command. Example:
##
##   git switch -c <new-branch-name>
##
## Or undo this operation with:
##
##   git switch -
##
## Turn off this advice by setting config variable advice.detachedHead to false
##
## HEAD is now at a2cc6f7 Comments from code review
```

In terms of the schematic of the repository, the environment now looks like this.

We have move the HEAD such that our working versions now point to the files that were originally delivered to the client.

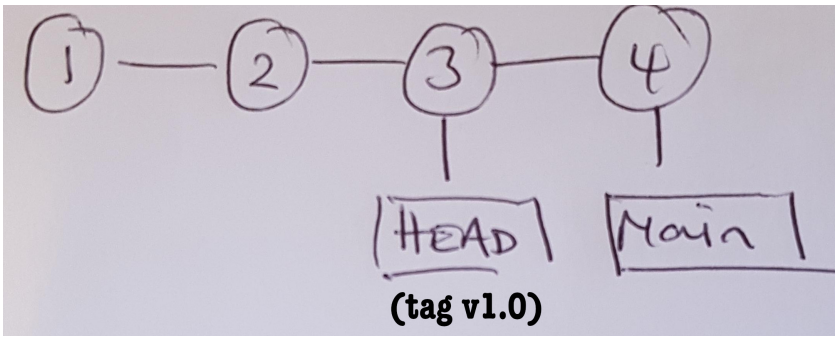


Figure 8.2: Rewind to earlier deliverable

> **Note**
>
> Don't panic overly about the warning about being in the *detached HEAD* state. What it means is that you are no longer on a branch. You have checked out a single commit in the history.

If you look at `branching.R` you will see that the starts of the secondary analysis has disappeared.

### 8.2.3 Fix issue

In order to fix the analysis we need to introduce the confounder as a covariate in the linear model. Introduce the following fixes. First to the descriptive summary:

```
message("\nDESCRIPTIVE SUMMARY:\n")

lyt <- basic_table() %>%
  split_cols_by("arm") %>%
  split_rows_by("age") %>%
  summarize_row_groups() %>%
  analyze("y", mean, format = "xx.x")

build_table(lyt, d)
```

and then to the analysis make these corrections and finally re-run the script to confirm that it produces what we expect.

```
lm1 <- lm(y ~ x * u, data = d)
summary(lm1)
```

Stage and then commit these changes. We can get some insight into the state of the tree now. Below I have added one more commit so that you can get a sense of how things are progressing.

```
git log --oneline --decorate --graph --all
## * 7ab12b3 (HEAD) Code review correction
## * 4cca810 Added emergency fix
## | * c82a48d (main) Started on secondary analyses
## |/
## * a2cc6f7 (tag: v1.0) Comments from code review
## * fa24778 branching.R
## * 3bdac46 Revised approach to capturing input
## * 327170a Minor edit
## * b078716 New file
## * 0cd2d52 First commit
```

The equivalent illustration would look something like this:

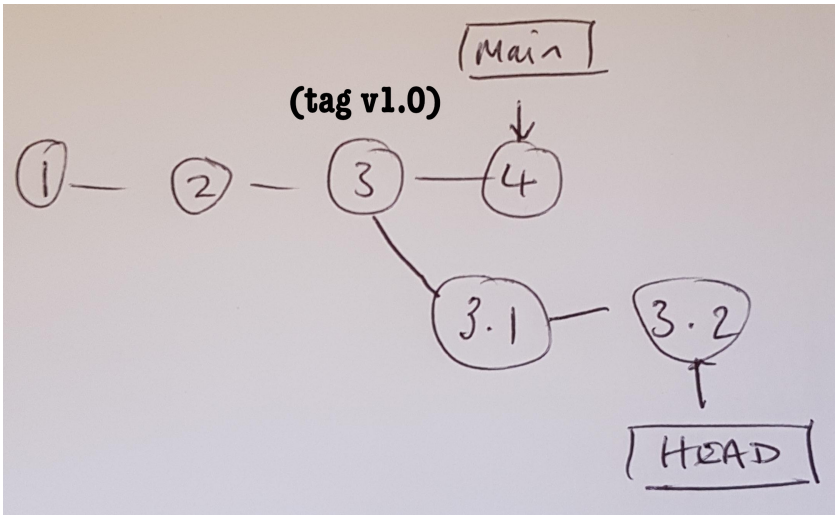


Figure 8.3: Add a fix

### 8.2.4 Make permanent

Now that we have fixed the code, we want to make the change permanent. That is, we want to formally tell git that our alternative history should be maintained. The way to do that is to create a new branch out of the recent changes (which already look like a branch).

Arguably, it would probably have been better to do this first given that we had to make the change. On the other hand, if we were just experimenting, then perhaps there was never any

intention to make the change permanent and a new branch would not be required.

```
git branch fix-01
# Switch to new branch
git checkout fix-01
```

When we look at the tree we see both HEAD and the branch (note the first line text which says *HEAD, fix-01*).

```
git log --oneline --decorate --graph --all
## * 7ab12b3 (HEAD, fix-01) Code review correction
## * 4cca810 Added emergency fix
## | * c82a48d (main) Started on secondary analyses
## |/
## * a2cc6f7 (tag: v1.0) Comments from code review
## * fa24778 branching.R
## * 3bdac46 Revised approach to capturing input
## * 327170a Minor edit
## * b078716 New file
## * 0cd2d52 First commit
```

And now the picture is

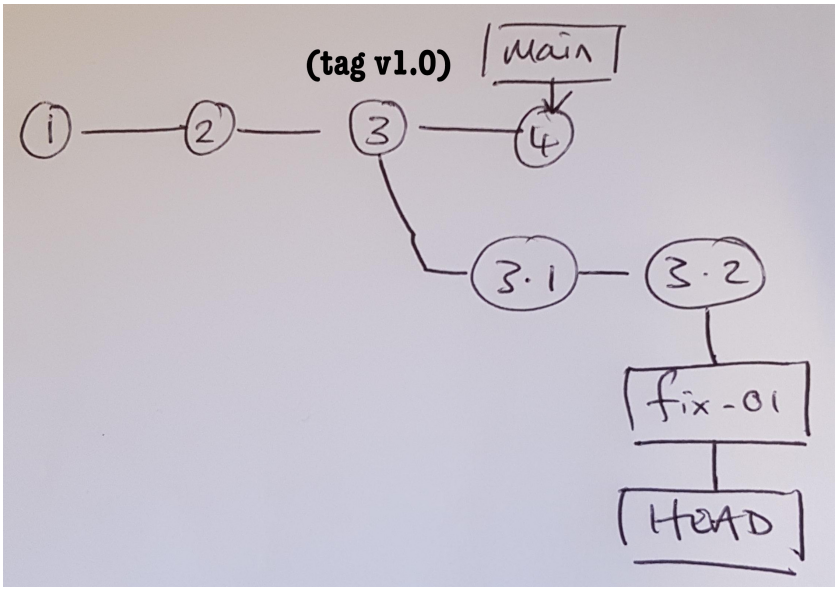


Figure 8.4: Looking at the HEAD of our new branch

If this is the version that we re-issue to the client, we might as well tag it.

```
git tag -a v1.1 -m "Analysis 1 (re-issue)"
```

### 8.2.5 Switching back to the secondary

Switching back to our partially complete secondary analysis is simple:

```
git checkout main
```

If you look at the `branching.R` script you will be able to see the secondary analysis we started some time ago. That is, if you look closely, you will see that the changes we just made in the `fix-01` branch have not yet been propagated to the `main` branch. It is important that we pick up this fix.

This process is known as *merging* and it will be tackled shortly.

### 8.2.6 Clean up

When you are done with a branch as in you have finished all the work, merged (see next) and released, you might want to remove the branch.

You can do this with

```
git branch -d <branch name>
```

noting that you can use a capital `-D` to force the delete irrespective of the merge status (unsafe).

# 9 Merge

## 9.1 Recap

From the previous branching example -

1. We delivered stage one of an analysis to the client
2. We started on the secondary analyses
3. Before the secondary analysis was complete, we realised there was an error in the original analysis that needed an emergency fix
4. We rewound to an earlier state in the repository and then fixed the error and checked in our work
5. We made the work permanent by creating a new branch and then checking out that branch
6. We tagged the fix and the re-issued the analysis to the client
7. We jumped back to our secondary analysis by checking out the main branch

Ok, so now we have a fix to the error in the original analysis in one branch `fix-01` and a partially completed secondary analysis on the `main` branch. We want to bring the changes from the fix into our present work.

Bascially, we want to do something like this:

## 9.2 Merge processes

There are two related approaches for combining your changes into a shared repository; rebase and merge. Personally, I tend not to use `git rebase` and prefer `git merge` and that hasn't caused me any problems.

> **Note**
>
> The distinction between a rebase and a merge is that the merge takes all the changes in one branch and merges them in a single commit whereas a rebase rewrites commits from one branch to another. It might not mean a great deal, but here are what the two look like:

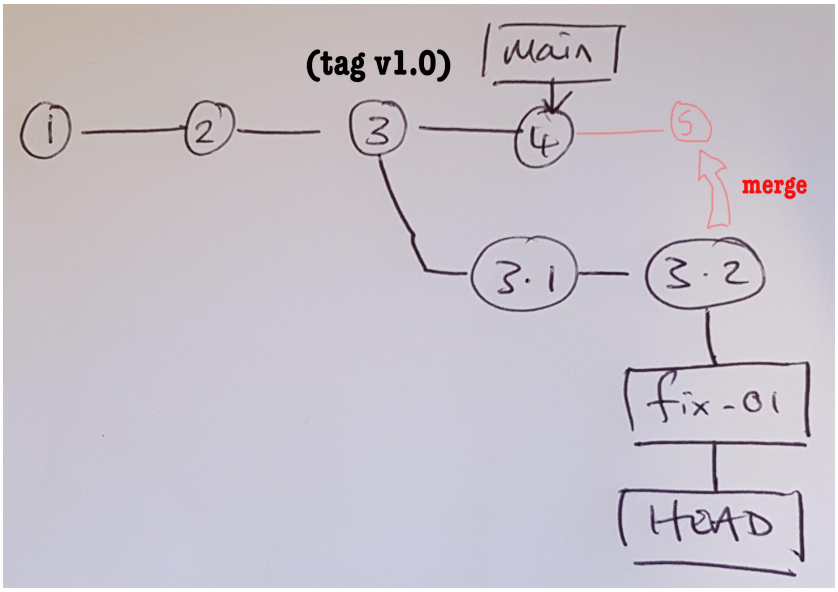


Figure 9.1: Bringing our fix from the past into the present…

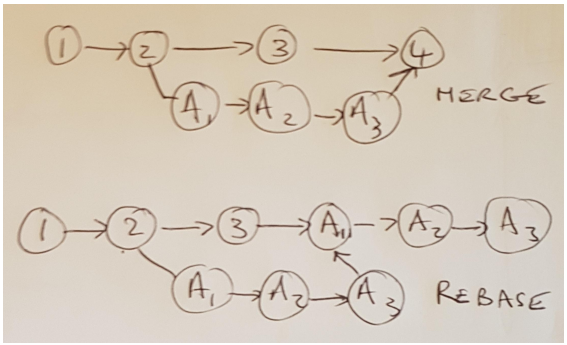


Figure 9.2: Bringing our fix from the past into the present…

There is also `git squash` that is used on GitHub, but I won't discuss that.

Often, merging is a fairly automated process. Run the following.

```
git checkout main
git merge fix-01
```

You will be prompted to enter a commit message

```
Merge branch 'fix-01'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

save and then you will get the following output.

```
## Auto-merging branching.R
## Merge made by the 'ort' strategy.
##  branching.R | 3 ++-
##  1 file changed, 2 insertions(+), 1 deletion(-)
```

If you look at the difference between the HEAD (the `HEAD~1` notation means compare with the previous commit, `HEAD~2` means compare with 2 commits prior and so on) and the commit associated with the merge you will see

```
git diff HEAD~1
## diff --git a/branching.R b/branching.R
## index b60c9f0..8f1c1b3 100644
## --- a/branching.R
## +++ b/branching.R
## @@ -45,6 +45,7 @@ message("\nDESCRIPTIVE SUMMARY:\n")
##
##  lyt <- basic_table() %>%
##    split_cols_by("arm") %>%
## +  split_rows_by("age") %>%
##    summarize_row_groups() %>%
##    analyze("y", mean, format = "xx.x")
##
## @@ -54,7 +55,7 @@ build_table(lyt, d)
##
##  message("\n\nANALYSIS OF CONTINOUS OUTCOME (UNSTRATIFIED):\n")
##
## -lm1 <- lm(y ~ x, data = d)
## +lm1 <- lm(y ~ x * u, data = d)
##  summary(lm1)
##
##  message("\n\nANALYSIS OF BINARY OUTCOME (UNSTRATIFIED):\n")
```

We now have the changes from the emergency fix in the main branch and we can continue

with the secondary analysis.

Sometimes merging doesn't work quite so smoothly and we need to iron out conflicts.

## 9.3 Exercises

**Exercise 9.1.** Complete the analysis by adding the following code to generate a figure from the fitted model.

```
p1 <- ggplot(d_fit, aes(x = arm, y  = pr_z)) +
  geom_point() +
  geom_linerange(aes(ymin = pr_z_lb, ymax = pr_z_ub)) +
  scale_x_discrete("") +
  scale_y_continuous("Probability of response")

ggsave("fig-sec.png", p1, width = 10, height = 10, units = "cm")
```

Here is what you need to do:

- Run the updated script `Rscript branching.R` to make sure it works.
- Edit the `.gitignore` file so that the figure does not get committed to the repository.
- Stage and commit the files and then review the commit history.
- Create a v2.0 tag with an meaningful message.

**Exercise 9.2.** Usually it makes sense to create a new branch for each piece of development we undertake. This ensures that the main branch continues to reflect a working version at all times. The goal of this exercise is to simply give you more familiarity with switching between branches.

Create a new branch from the current state and call it `analysis-03` Run the following code:

```
git branch analysis-03
git checkout analysis-03
git status
```

Add the analysis code into `branching.R`

```
message("\n\nANALYSIS OF SURVIVAL OUTCOME (UNSTRATIFIED):\n")
lm3 <- coxph(Surv(w, evt) ~ x, data = d)
summary(lm3)
```

Run the script to make sure it works then stage and commit along with an updated `.gitignore` files that excludes all `png` files.

Checkout (switch to) main in order to emulate progression in the main branch:

```
git checkout main
```

Add the following change to the `branching.R` code (just adding a new theme to the ggplot figure)

```
p1 <- ggplot(d_fit, aes(x = arm, y  = pr_z)) +
  geom_point() +
  geom_linerange(aes(ymin = pr_z_lb, ymax = pr_z_ub)) +
  scale_x_discrete("") +
  scale_y_continuous("Probability of response") +
  theme_bw()
```

Run the script, then stage and commit.

Re-checkout `analysis-03` branch:

```
git checkout analysis-03
```

Add the following code:

```
png("fig-surv.png")
plot(survfit(Surv(w, evt) ~ x, data = d), lty = 1:2)
dev.off()
```

Run the script to make sure it works then stage and commit. Treat this as the release by tagging it as `v3.0`. Checkout the `main` branch and merge the analysis into main.

```
git checkout main
git merge analysis-03
```

Review the commit history:

```
git log --oneline --decorate --graph --all
## *   2829471 (HEAD -> main) Merge branch 'analysis-03'
## |\
## | * be09457 (tag: v3.0, analysis-03) Finished surv
## | * 7a5e7b9 Surv analysis
## * | 53115b6 minor
## |/
## * 1a9dfeb (tag: v2.0) Edits from code review
## * d1586df Completion of secondary analysis
```

```
## *   bf34b9c Merge from fix-01
## |\
## | * b8de16c (tag: v1.1, fix-01) Edits from code review
## | * 12561d9 Emergency fix
## * | d88f9e5 WIP
## |/
## * a2cc6f7 (tag: v1.0) Comments from code review
## * fa24778 branching.R
## * 3bdac46 Revised approach to capturing input
## * 327170a Minor edit
## * b078716 New file
## * 0cd2d52 First commit
```

At this point, you have been introduced to the fundamentals of git. But there are things that are going to catch you out. For example, in the above work, we used three-way merges. These are convenient but can also get messy when your branching structure gets complex. In most of our daily work merges will probably suffice, but know that there is another way. The other way is rebasing, but we will not deal with it here.

# Part III

# Part 3 - Collaboration

This part is about how git is used in a collaborative context. It is probably the toughest part and so we will just cover the basics.

- Introduce fileserver backed git remotes
- Introduction to GitHub
- GitHub collaboration processes
- Documentation backed by GitHub

# 10 Collaboration 101

What is covered here simply scratches the surface. Git is extremely flexible in the workflows that can be set up. A basic approach is that of a centralised repository. This is easy to understand and is simple to set up for small teams.

The concept of *remotes* is introduced as is an associated workflow.

## 10.1 No GitHub?

GitHub is a hosting service for git repositories. It also extends some of the git functionality. They have captured the market.

Before 2005 (ish), GitHub did not exist, and it wasn't a catastrophe. What did we do? Git has the functionality built into it to allow collaboration via hosting git repositories directly on fileservers. This is a reasonable solution for small teams with access to a common network, but it has its limits and you are probably not going to go down this path unless the infrastructure is formally managed by your IT support. Nevertheless, most bluechip organisations used this approach during the 2000's.

Repositories that are retained outside of your local machine are referred to as *remotes*. It doesn't matter whether they are hosted on LIQ, or a random file server or GitHub or BitBucket or Timbuktu. They are still known as remotes.

Remotes allow you to work on a repository collaboratively with your colleagues. This has been standard practice in software development since about 300 B.C.

We will talk about hosting remotes on fileservers here and then move on to GitHub next.

## 10.2 So remote

What remote (or remotes) are associated with the `first-repo` repository that we have been working on? You can use `git remote` to find out:

```
git remote -v
```

No remotes.

If at any time you want to review the remotes associated with your repository (you can have more than one remote) then invoke the command that we just used.

## 10.3 Initialising remotes

Let's set up a remote on OneDrive. OneDrive is a cloud file-server, but technically we could replace OneDrive with any file-server. For example, we used to have the LIQ drive mounted as a network drive and we could use that.

Anyway, just follow along.

```
git clone --bare first-repo /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydne
```

The above takes the repository that we have been working on and moves the git database to OneDrive. If we look at OneDrive then we see:

```
pwd
## /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.git
ls -lrta
## total 48
## drwx------@  20 mark  staff   640  8 Nov 08:36 ..
## -rw-r--r--    1 mark  staff    73  8 Nov 08:36 description
## -rw-r--r--    1 mark  staff   188  8 Nov 08:36 config
## -rw-r--r--    1 mark  staff   618  8 Nov 08:36 packed-refs
## -rw-r--r--    1 mark  staff    21  8 Nov 08:36 HEAD
## drwxr-xr-x@  11 mark  staff   352  8 Nov 08:36 .
## drwxr-xr-x    3 mark  staff    96  8 Nov 08:36 info
## drwxr-xr-x   16 mark  staff   512  8 Nov 08:36 hooks
## drwxr-xr-x  105 mark  staff  3360  8 Nov 08:36 objects
## drwxr-xr-x    5 mark  staff   160  8 Nov 08:36 refs
## -rw-r--r--@   1 mark  staff  6148  8 Nov 08:36 .DS_Store
```

The first thing to note is that there is no working directory.

Now let's imagine that Sylvie has just joined the team as naive Bayesian statistician. Sylvie needs to start work on the `first-repo` project. First, we need to get the repository onto her local machine.

The process is as follows.

1. Go to you Documents folder (or wherever you intend to store your projects)
2. Clone the remote

Obviously, I am working off a single laptop so I will clone the repo to some name other than `first-repo`. I will refer to this location as *Sylvie's repo* vs *Mark's repo* which is the one that I have been referencing when demonstrating examples.

In contrast to the bare repository remote, Sylvie now has the example files that we have been developing to date.

> **Note**
>
> Note that I called the repo `sylvie-first-repo`. We can call the local version of the repo whatever we want.

```
cd /Users/mark/Documents/
git clone /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney\(Staff\)/first-r
ls -lrta
## total 40
## drwxr-xr-x   3 mark  staff    96  8 Nov 11:07 ..
## -rw-r--r--   1 mark  staff   100  8 Nov 11:07 .gitignore
## -rw-r--r--   1 mark  staff  2274  8 Nov 11:07 branching.R
## -rw-r--r--   1 mark  staff   242  8 Nov 11:07 hello.R
## -rw-r--r--   1 mark  staff    38  8 Nov 11:07 myscript.R
## drwxr-xr-x   8 mark  staff   256  8 Nov 11:07 .
## -rw-r--r--   1 mark  staff   138  8 Nov 11:07 readme.md
## drwxr-xr-x  12 mark  staff   384  8 Nov 11:07 .git
```

Look at the details of the remote associated with Sylvie's repo:

```
cd /Users/mark/Documents/sylvie-first-repo/
git remote -v
origin  /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.
origin  /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.
```

The picture now looks like this

Over on the left we have the remote git database. On the right we have Mark who created the remote from his local repository and Sylvie who cloned the remote repository to her local machine. Git isn't restricted to this centralised architecture (it is possible to have multiple remotes and other variations) but it is a common paradigm.

GitHub uses a different workflow that commonly (though not exclusively) relies on a project owner pulling a developers updates into a main project repository rather than the developer pushing their work directly to the project repository. We will see more about this later.

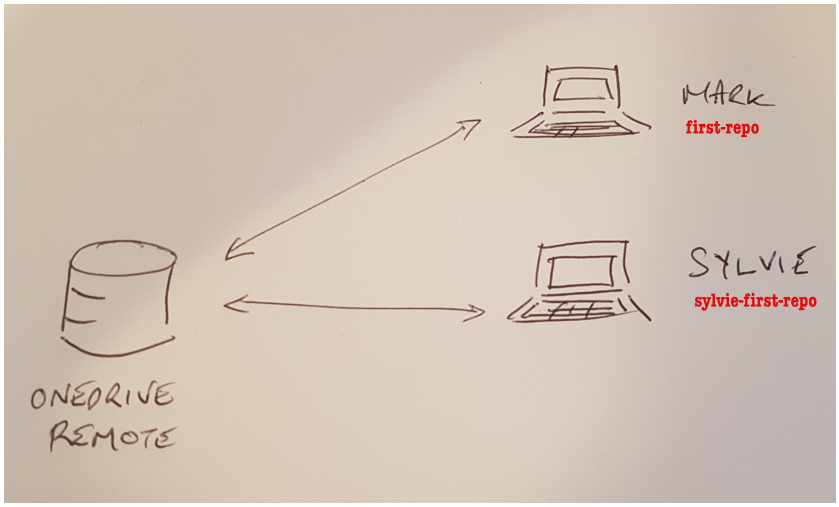


Figure 10.1: Clone repository

## 10.4 Keeping remotes up to date

For now, you can think of a remote as the hub around which collaboration revolves. As a project progresses, the local files are updated and we need to get these updates to the remote. However, others may beat us to the post in this process and thus we may be out of sync with the remote.

**Again, this would mean that the state of the remote is ahead of your local repository.**

In this scenario, git (and GitHub) will not allow you to send your updates to the repository unless you are sync'd. In other words, you have to be up to date in order to send your changes to the remote.

To be able to push your work up to the remote, the steps are:

1. ensure that all your changes are committed to the local repository
2. run `git fetch` to bring down the latest changes from the remote
3. if necessary, merge any changes into your local repository to get up to date
4. ensure (again) that all your changes are committed to the local repository
5. `git push` you modifications up to the remote

Here is another picture.

It sounds (and looks) a bit involved, but in practice it is simple. We will go over the steps below.

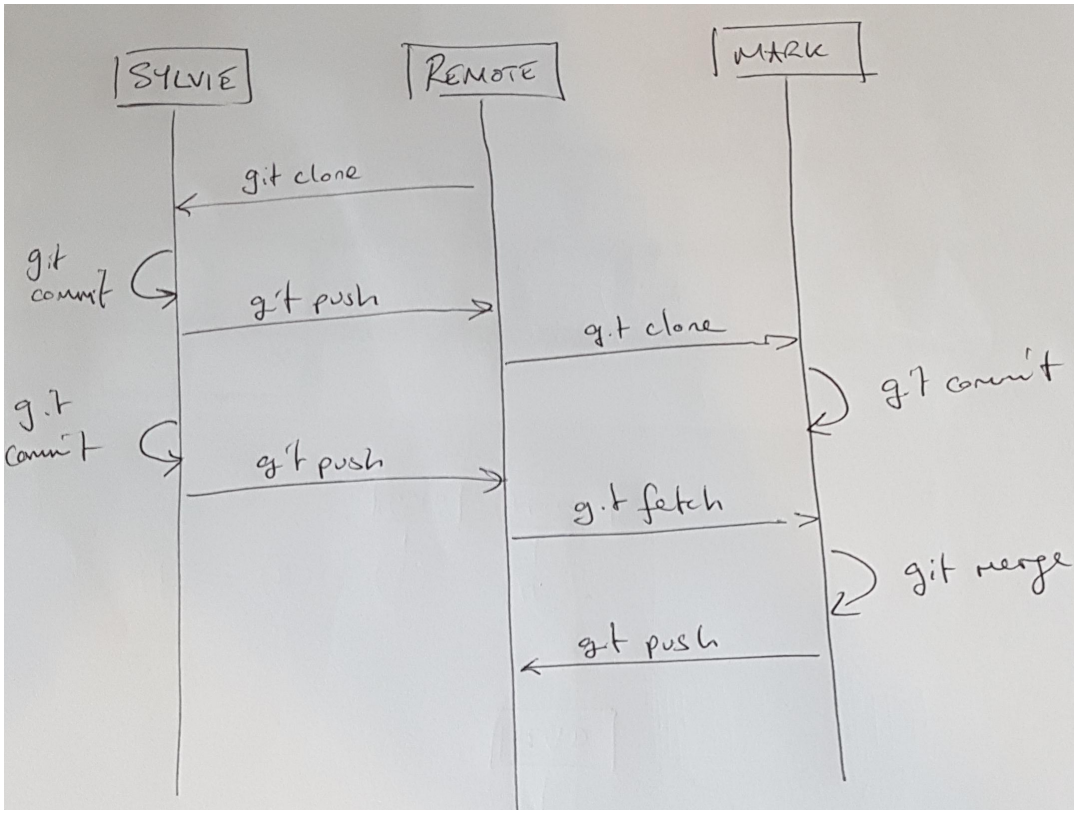


Figure 10.2: Centralised workflow

### 10.4.1 Fetch and push (simple case)

First Silvie ensures that all her local changes are committed to the repository. It looks like Silvie made a change to the `readme.md` when we were not looking.

```
cd sylvie-first-repo
git status
## On branch main
## Your branch is up to date with 'origin/main'.
##
## Changes not staged for commit:
##   (use "git add <file>..." to update what will be committed)
##   (use "git restore <file>..." to discard changes in working directory)
##  modified:   readme.md
##
## no changes added to commit (use "git add" and/or "git commit -a")

git add readme.md; git commit -m "Minor change for testing fetch"
```

No changes have been made to the remote by anyone else (Silvie has not pushed her changes) so when we run `git fetch` there are no updates to be had. Therefore we can go and push the changes.

```
git push
## Enumerating objects: 5, done.
## Counting objects: 100% (5/5), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (3/3), done.
## Writing objects: 100% (3/3), 342 bytes | 342.00 KiB/s, done.
## Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
## To /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.gi
##    7eee4b5..49225cc  main -> main
```

> **Note**
>
> The examples under the `git push` man page are helpful and informative for further detail on the interpretation of `git push`.

Looking at the logs we see the following

```
git log --oneline
## 3a3dd8d (HEAD -> main, origin/main, origin/HEAD) Testing Sylvie
```

```
## 2829471 Merge branch 'analysis-03'
## be09457 (tag: v3.0, origin/analysis-03) Finished surv
## 53115b6 minor
## 7a5e7b9 Surv analysis
## 1a9dfeb (tag: v2.0) Edits from code review
## d1586df Completion of secondary analysis
## bf34b9c Merge from fix-01
## b8de16c (tag: v1.1, origin/fix-01) Edits from code review
## 12561d9 Emergency fix
## d88f9e5 WIP
## a2cc6f7 (tag: v1.0) Comments from code review
## fa24778 branching.R
## 3bdac46 Revised approach to capturing input
## 327170a Minor edit
## b078716 New file
## 0cd2d52 First commit
```

### 10.4.2 Fetch and push (merge required)

In the interim Mark has been working on the `first-repo` and will want to get these changes into the remote. However, Mark hasn't yet linked his local repository with the remote.

```
git remote -v
## nothing
```

To fix this we use `git remote add` which, predictably tells git to configure a link between a local and remote version of the repository. This is a one time process and once we have set the remote, we do not need to do it again until we need to add a new remote.

```
git remote add origin /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney\(Sta
git remote -v
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
```

> **Note**
>
> The use of the word *origin* below is just convention to indicate the primary remote. It is a shortcut for the full directory path (or URL) for the remote. You can call the remote anything you want but most people stick with the convention of *origin*, which is a bit like the GitHub convention of using *main* for the main branch. If you were to add a second

> remote, you would call it something other than *origin*.

Next, we need to tell the local repository to reference this remote.

```
git remote add origin /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney\(Sta
git remote -v
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
```

We see that the remote has been added for both push and fetch. Before we use these, we need to make an existing local branch track a specific remote branch which is achieved with

```
git fetch  # pulls down information on the branches in the remote
git branch --set-upstream-to=origin/main main # links a local with remote branch
```

Checking the status we see that the change that Sylvie pushed has left us out of sync.

```
git status
## On branch main
## Your branch and 'origin/main' have diverged,
## and have 1 and 1 different commits each, respectively.
##   (use "git pull" if you want to integrate the remote branch with yours)
##
## nothing to commit, working tree clean
```

If I try to push my change anyway, git barfs:

```
git push
## To /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.gi
##  ! [rejected]        main -> main (non-fast-forward)
## error: failed to push some refs to '/Users/mark/Library/CloudStorage/OneDrive-TheUniver
## hint: Updates were rejected because the tip of your current branch is behind
## hint: its remote counterpart. If you want to integrate the remote changes,
## hint: use 'git pull' before pushing again.
## hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This is what I was referring to earlier. Git will not allow you to screw up the remote without some strenous effort on your part. It would be nice if git had told us what was different, but it forces us to do this via the following:

```
git fetch # this was done previously, but just to be clear that you need to do a fetch fir
git diff main origin/main
## diff --git a/readme.md b/readme.md
## index 0165627..797358c 100644
## --- a/readme.md
## +++ b/readme.md
## @@ -4,7 +4,9 @@ A demo markdown file for the git workshop.
##
##  A new line for testing.
##
## -Contains standalone R scripts for staged deliverables to client.
## +Contains standalone R scripts.
## +
## +Test by Sylvie
```

You can see the line that I introduced and also the change that Sylvie added and committed to the remote. There are a few options to resolve this. Most people use `git pull` which runs the two steps as one. I prefer to be explicit and feel like it might be a bit safer to split the process up into a `fetch` and then a `merge`, although I think I am in the minority.

```
git fetch
git merge
## Auto-merging readme.md
## Merge made by the 'ort' strategy.
##  readme.md | 3 +++
##  1 file changed, 3 insertions(+)
```

> **Note**
>
> Git offers another command `git pull` that effectively combines `git fetch` with `git merge`. The benefit of using `git fetch` is that it will update your local repository but will not make any changes to your working directory (see Section 5.1.2.1 for a reminder).

And in this case, the merge appears to have worked correctly.

```
# first-repo

A demo markdown file for the git workshop.

A new line for testing.
```

```
Contains standalone R scripts for staged deliverables to client.

Test by Sylvie
```

## 10.5 Remotes and branches

One key aspect that I have not mentioned to date is for the scenario where new branches are created in a local repository.

To list all known branches in both your local repository and the remote (or remotes)

```
git branch -a
##   analysis-03
##   fix-01
## * main
##   remotes/origin/analysis-03
##   remotes/origin/fix-01
##   remotes/origin/main
```

Let's say that Mark is asked to develop a new analysis. Learning from past mistakes, he creates an `analysis-04` branch and adds the following content to run a non-parameteric surivival analysis.

```
git branch analysis-04
```

To complete the analysis, the following is added to the `branching.R` script, which is tested, staged and committed.

```
message("\n\nANALYSIS OF SURVIVAL OUTCOME (UNSTRATIFIED, NON-PARAMETRIC):\n")

lm4 <- survdiff(Surv(w, evt) ~ x, data = d)
print(lm4)
```

Following the earlier process, Mark then switches back to main and merges in the new analysis.

```
git checkout main
git merge analysis-04
```

and then pushes the change up to the remote repository.

```
git push
```

This does not establish the `analysis-04` branch within the remote, which may or may not be a problem (usually not) but it is worth being aware of. Other users will not be able to get access to the evolution of the files that occurred on the development branch. To push the `analysis-04` branch we use:

```
git push --set-upstream origin analysis-04
## Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
## To /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-repo.gi
##  * [new branch]      analysis-04 -> analysis-04
## branch 'analysis-04' set up to track 'origin/analysis-04'.
```

And now uses can fetch and review this branch within the context of their local repositories.

# 11 GitHub introduction

Rightly or wrongly, GitHub has become the de-facto tool for sharing tech work. GitHub is simply a hosting service for git repositories. It is not part of the git software and (unfortunately in my opinion) it is owned by Microsoft.

Around 10,000 companies (3M, Dell, Airbnb, Netflix …) use GitHub as part of their development stack. It sources literally 100's of millions of projects with billions of contributions to these projects per year. Around 95% of developers use git as the version control system of choice.

Alternatives that aim to serve similar goals include GitLab and Bitbucket.

From earlier, you should have set up an account, configured 2FA and created a personal token so that isn't going to be covered again. We will use `https` to access GitHub, which is the default, so you do not need to set up any other protocols.

## 11.1 Walk around the interface

When you log into GitHub you will be taken to your dashboard (homepage) which is a bit of a feed, quick links to some of your repositories and other things that might be of interest. Day to day, what you probably want to see and interact with is your repositories and their settings.

Go to the top right hand corner and click the far right icon to get the drop down, then select your profile. All of the following is based on my account but you can follow along under your login as well, just replace my username with yours in the relevant places.

From there you can edit various details pertaining to yourself, review your stats and so on and then jump into the git repositories that you have put onto GitHub.

Select the repository link to see a list of all your repositories. You can also get here directly from your landing page via the 'Your repositories' link. I have highlighted the `get-going-with-git` repo. You won't have this repository at this stage but that's ok, just follow along.

Selecting a specific repository brings up its landing page which includes the repository files, a description file (if one has been created), stats on the repository, settings and various other items. The page that is shown below corresponds to the git repository used to manage this text and website.

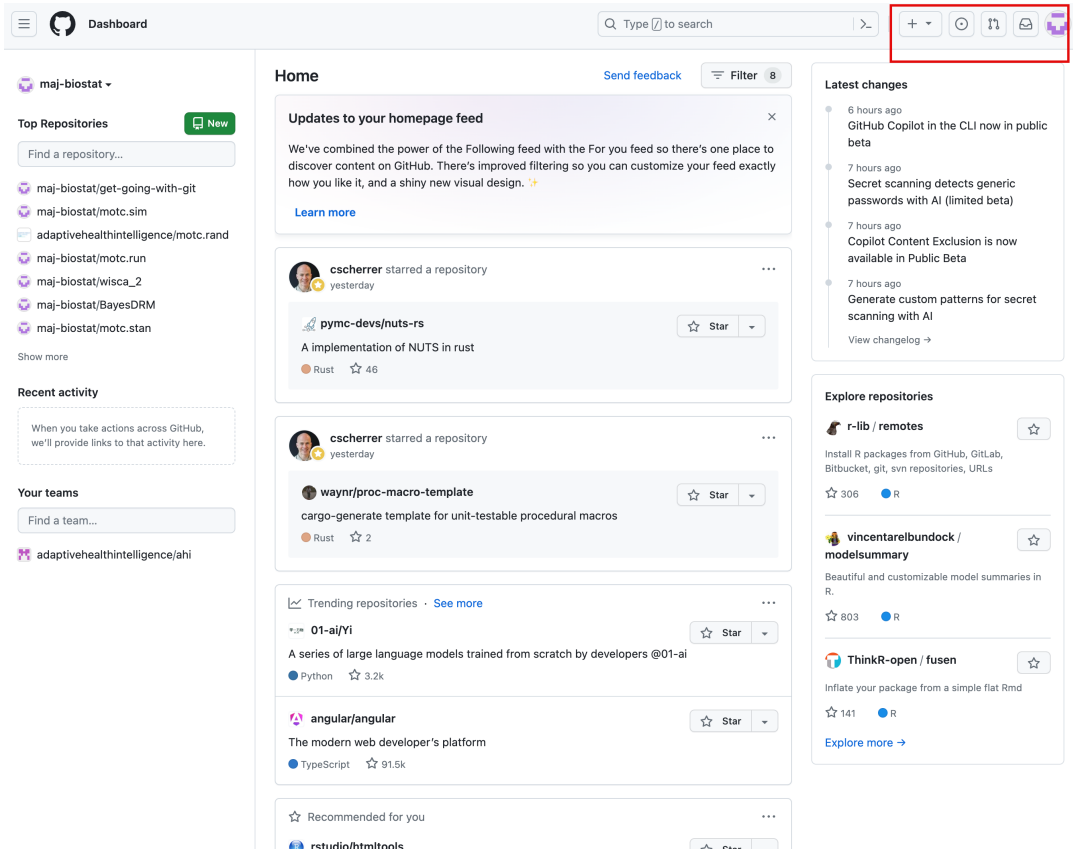


Figure 11.1: GitHub landing

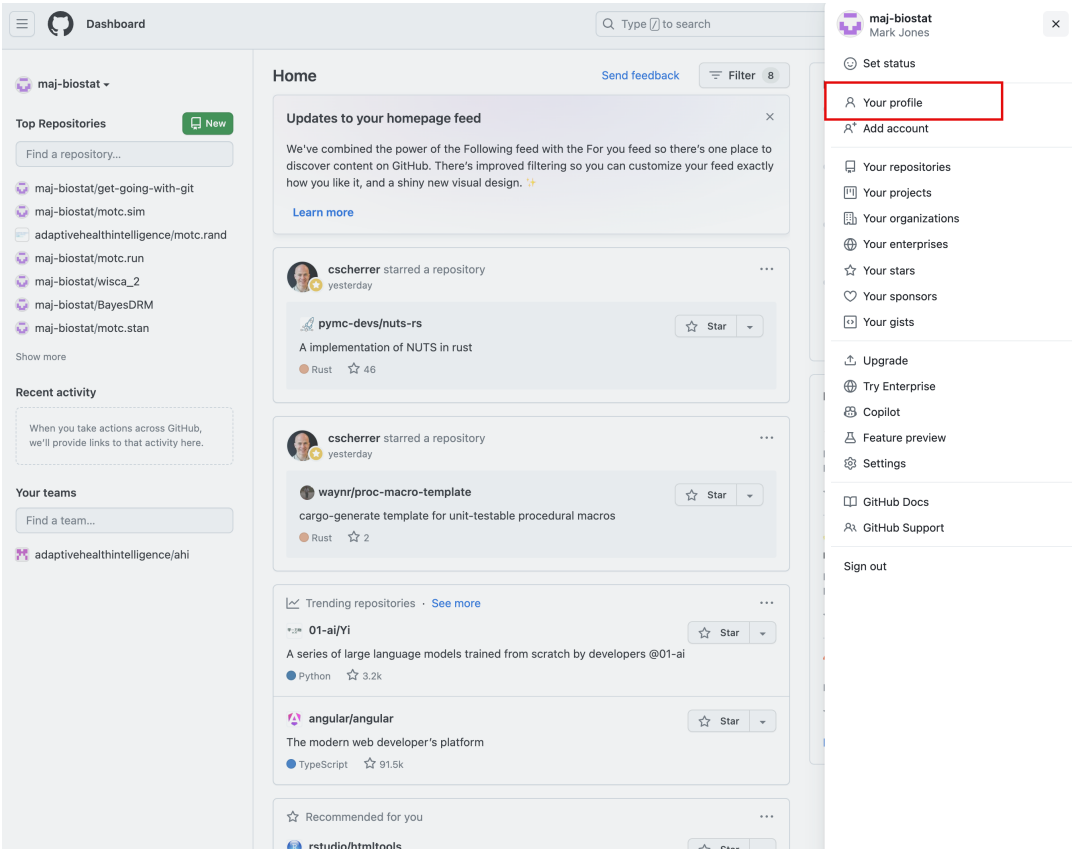


Figure 11.2: GitHub landing dropdown

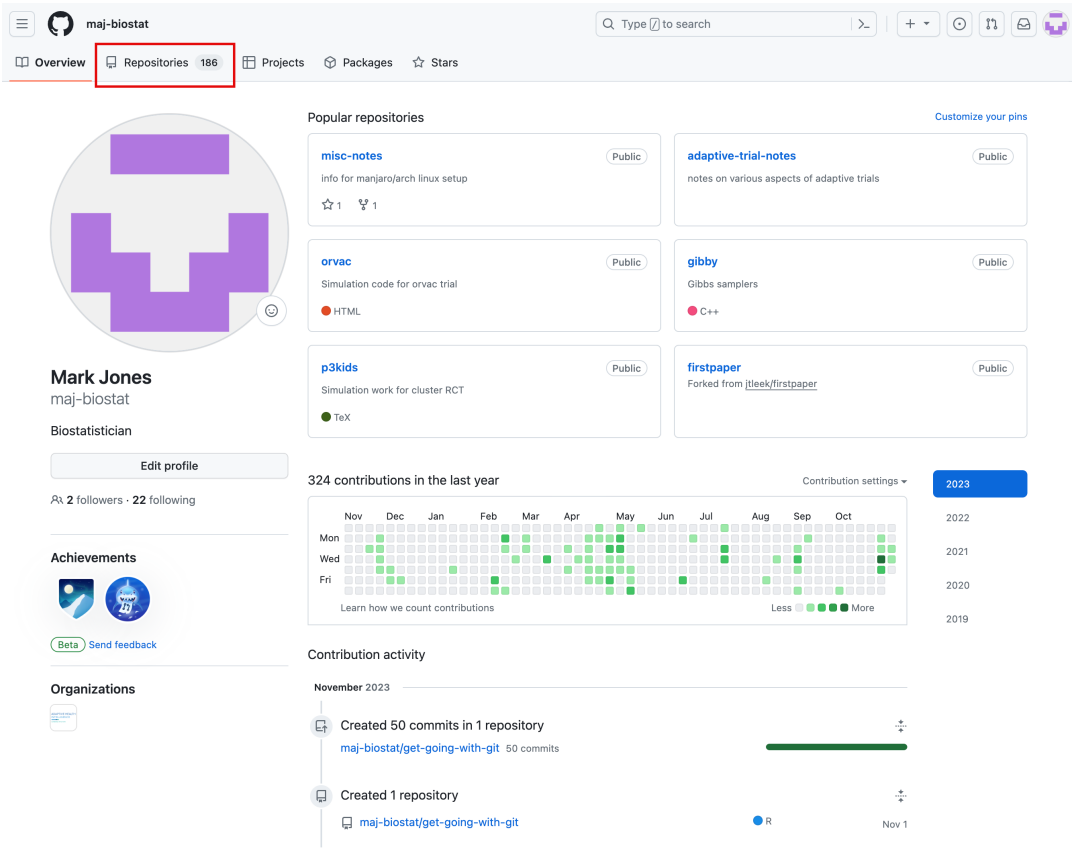


Figure 11.3: GitHub profile

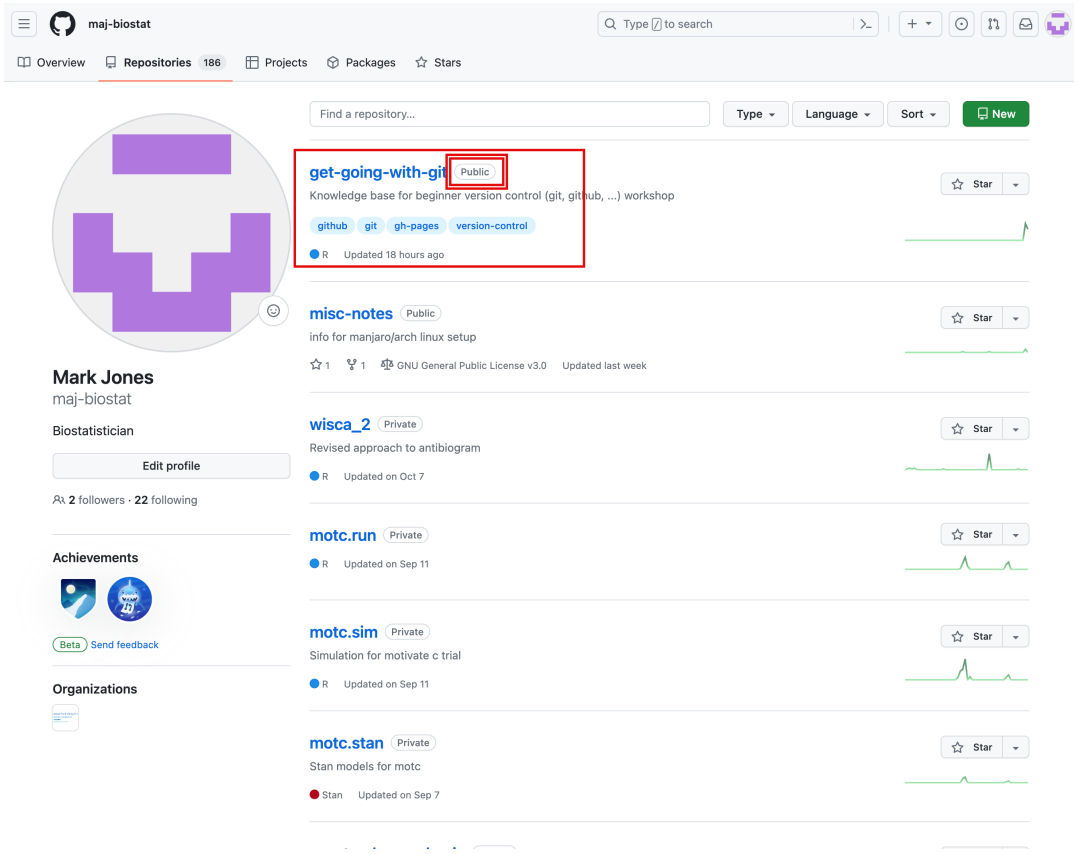


Figure 11.4: GitHub repository listing

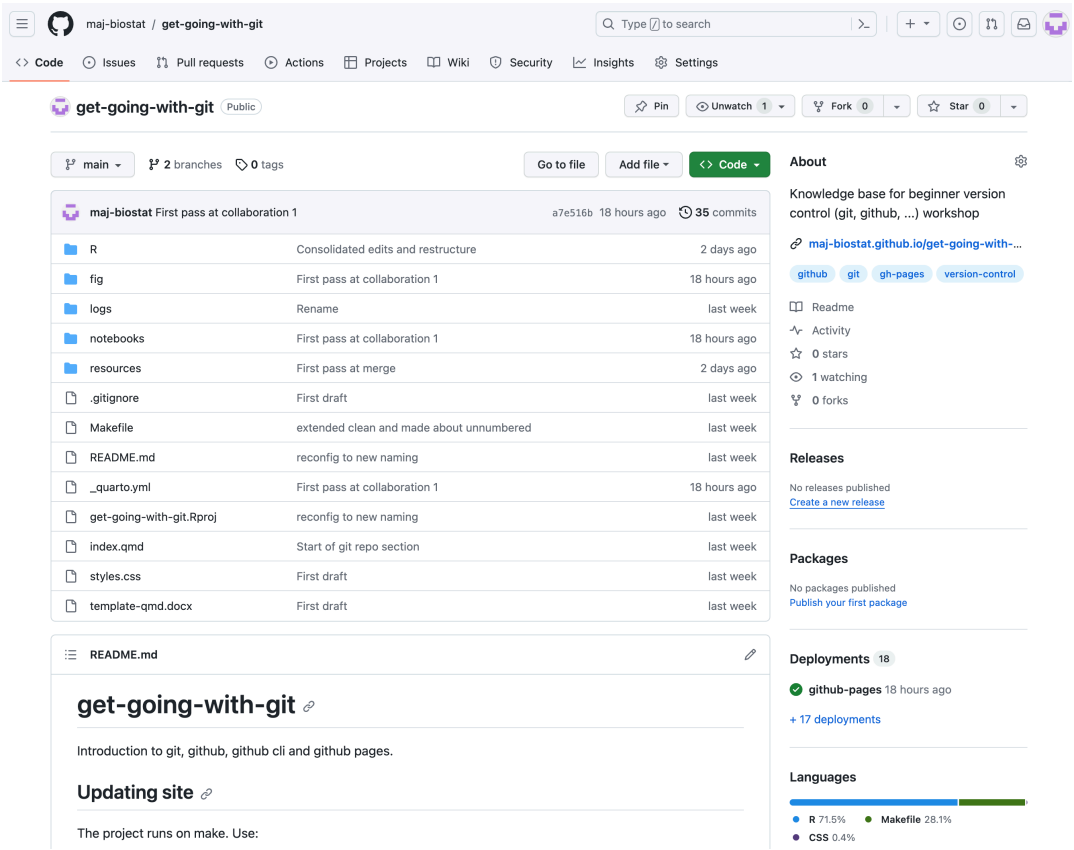


Figure 11.5: GitHub repository landing page

## 11.2 Common functions via GitHub

Some[1] of the functionality that git implements has been integrated into the GitHub platform. Let's go over some of them briefly.

### 11.2.1 Initialise repository

There are a few ways to do this. The simplest is to initialise the repo on GitHub then clone it down to your machine. The preferred way to to start with pre-existing local work (repo) and connect it with a remote on GitHub.

#### 11.2.1.1 Initialise on GitHub

You can achieve this directly from any page.

The configuration is largely self-explanatory.

And then you are taken to your new shiny repository page.

> **Note**
>
> The easiest (but not necessarily the best) way to collaborate via GitHub is to add collaborators to your repository. By adding collaborators, you are giving people read/write access to your repository. It is therefore your responsibility to make sure you are confident that they know what they are doing.
> The topic of pull requests (see later) introduce a mechanism that in part mitigates the risks associated with collaborative repositories.

And if you want to change the visibility (public/private) of the repository, the functions to do this are down at the bottom of general settings under the Dange Zone section.

#### 11.2.1.2 Initialise from existing repository

Again, you can achieve this directly from any page.

The only difference in the setup is that you **do NOT to initialize the repository with a readme**. You will then be shown the following instructions. The latter case is the one that is applicable, as shown.

If you recall from the previous discussion on fileserver remotes, we had set the remote for the `first-repo` repository to be a OneDrive remote. Recapping:

[1] Probably all, but I am guessing.

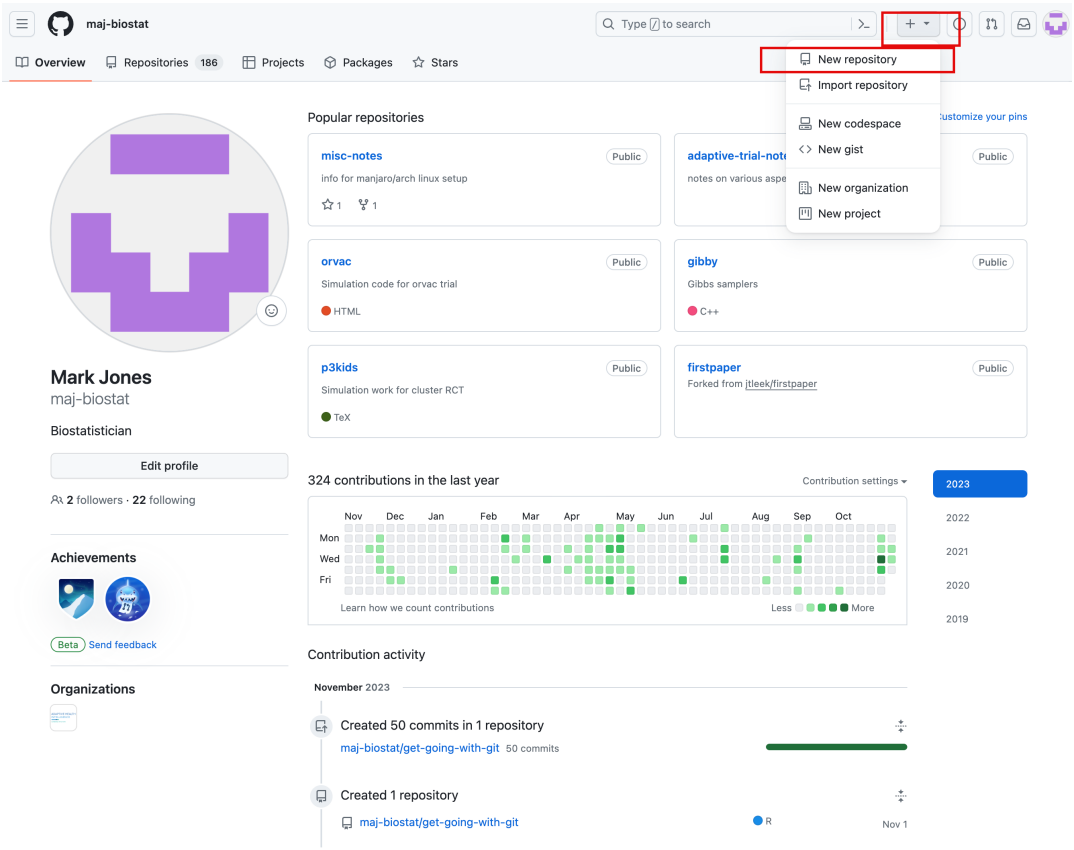


Figure 11.6: GitHub initialise new remote repository

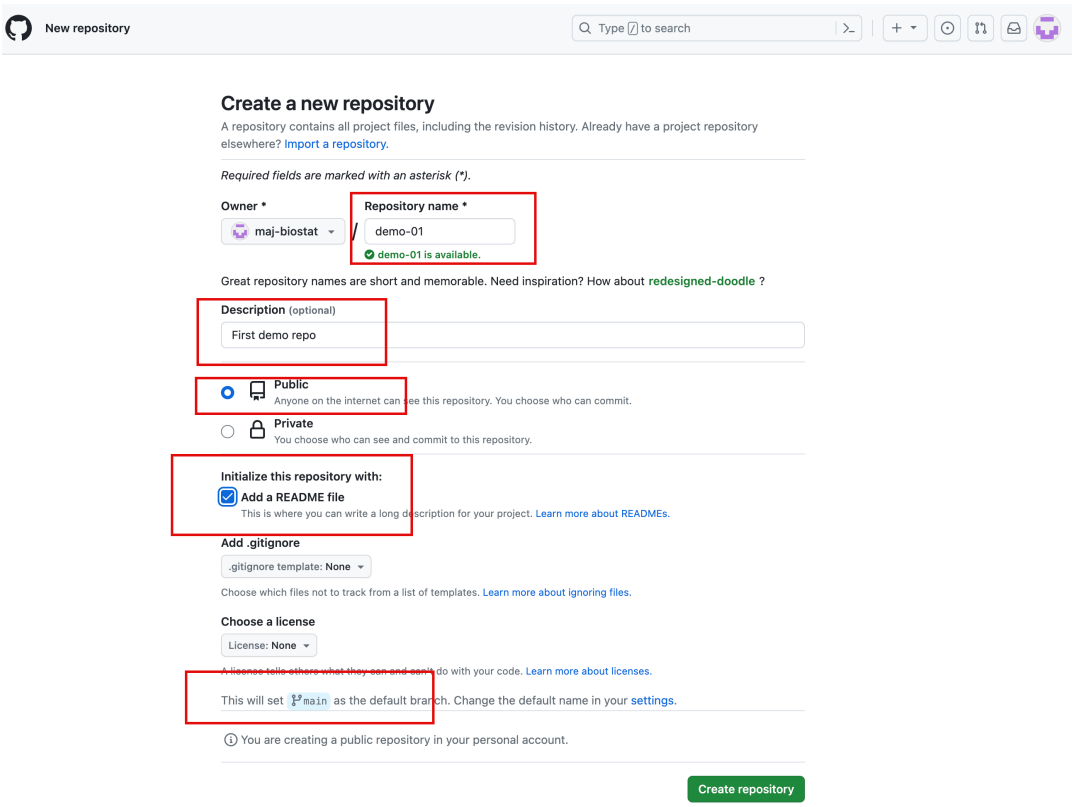


Figure 11.7: Configure the repository

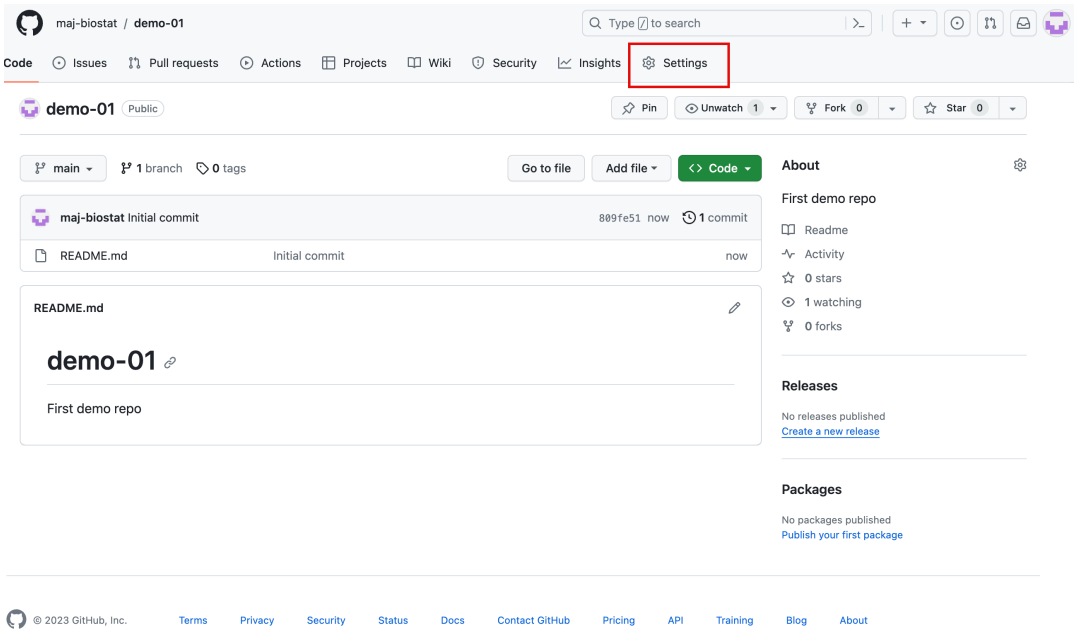


Figure 11.8: New repository page

```
pwd
## /Users/mark/Documents/project/misc-stats/first-repo
git remote -v
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
## origin   /Users/mark/Library/CloudStorage/OneDrive-TheUniversityofSydney(Staff)/first-r
```

While we can technically have multiple remotes, it might be easier to simply unlink from this current remote.

```
git remote rm origin
git remote -v
##
```

And now add GitHub as the new remote.

```
git remote add origin https://github.com/maj-biostat/demo2.git
git branch -M main
git push -u origin main

## Enumerating objects: 68, done.
```

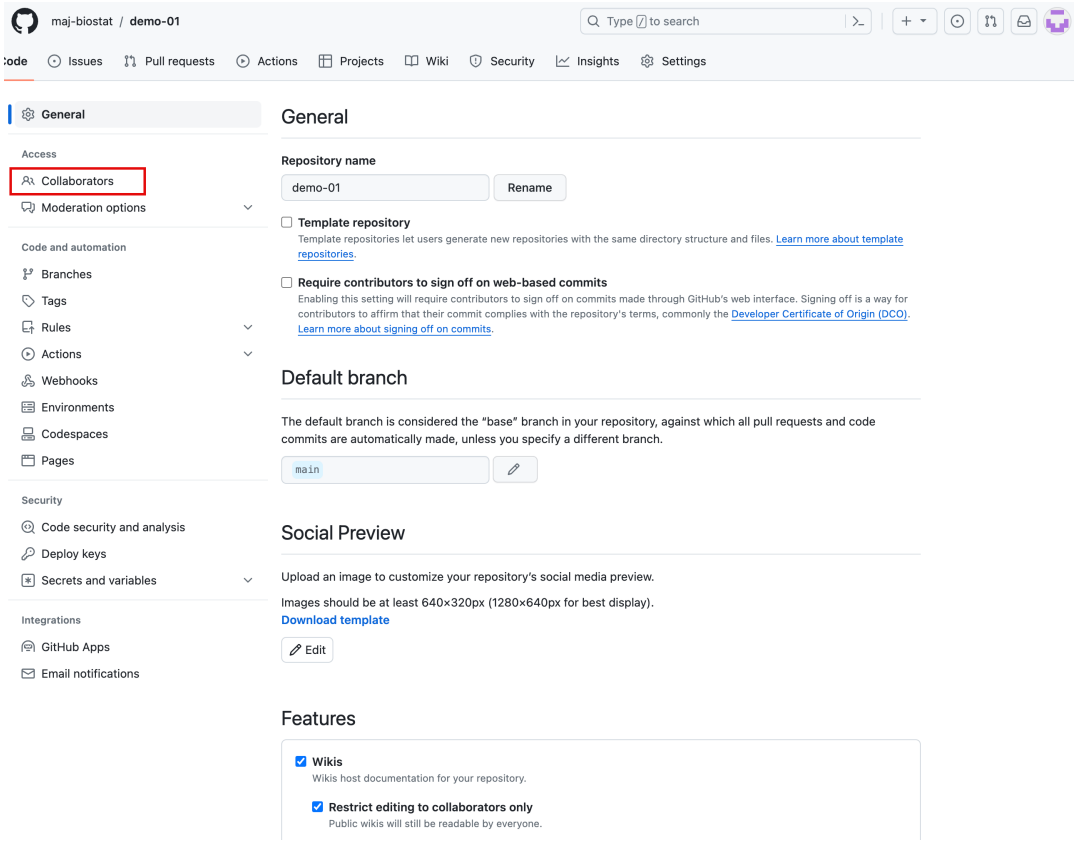


Figure 11.9: Collaborators setting

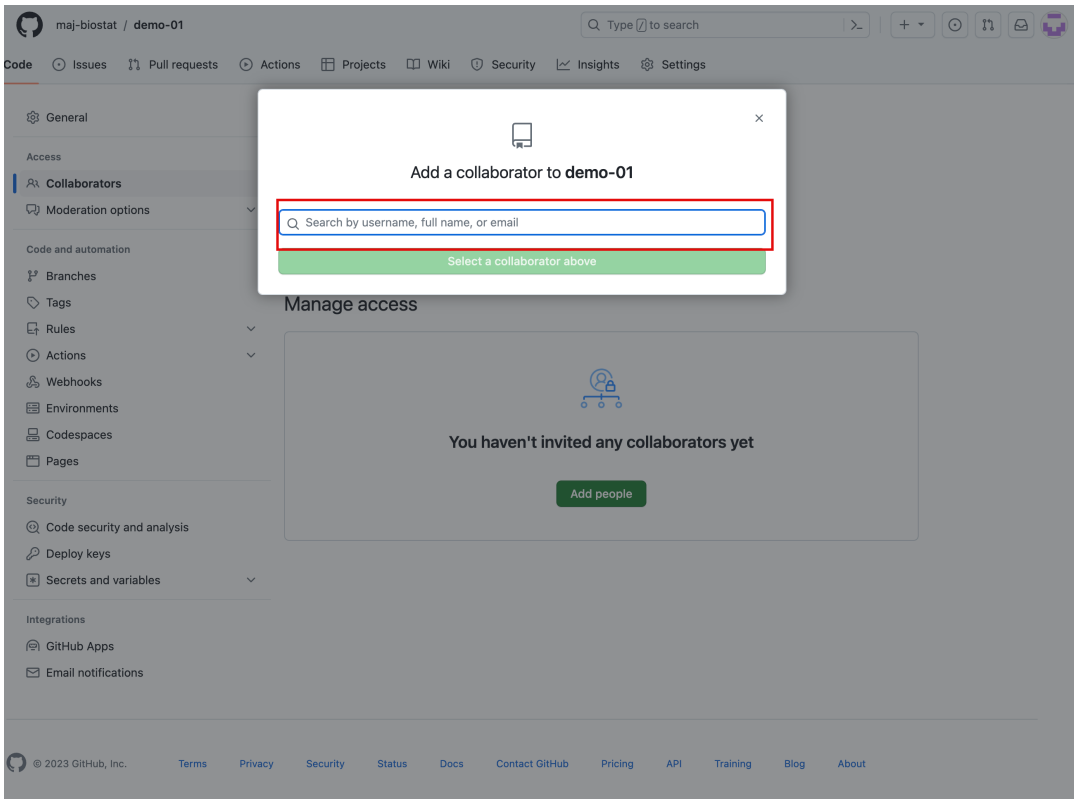


Figure 11.10: Enter collaborator details

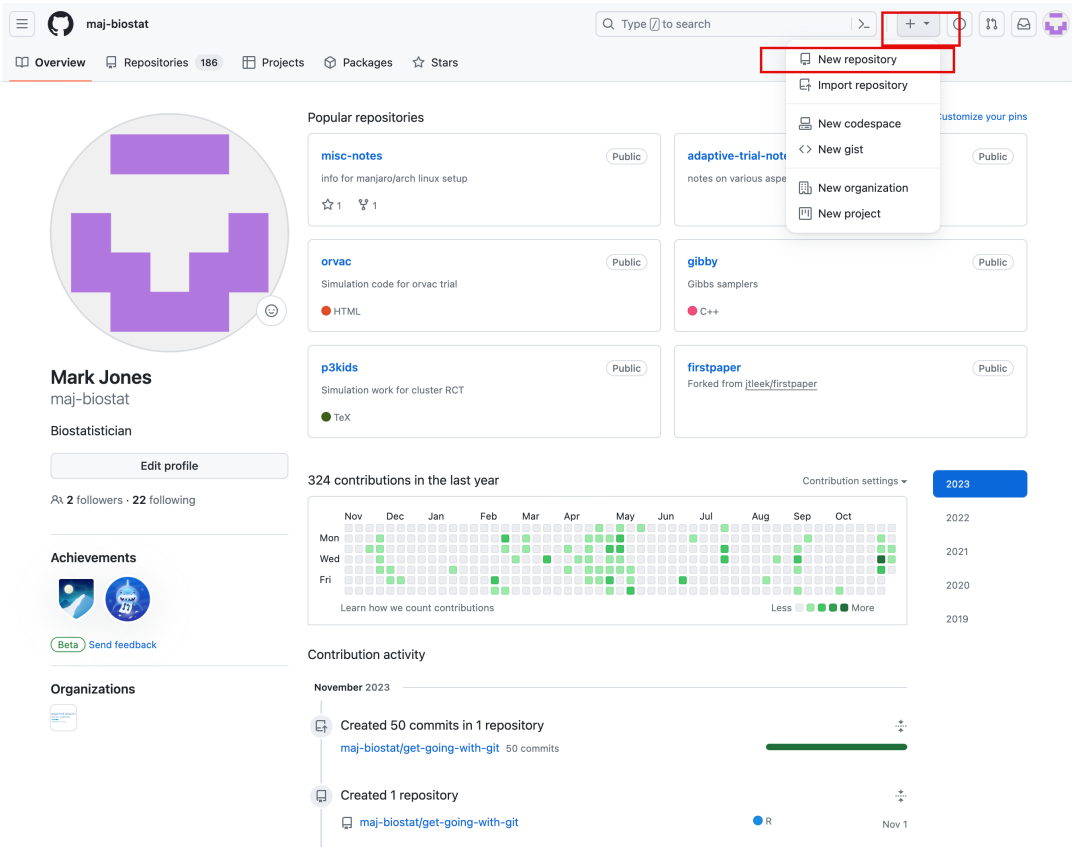


Figure 11.11: GitHub initialise new remote repository

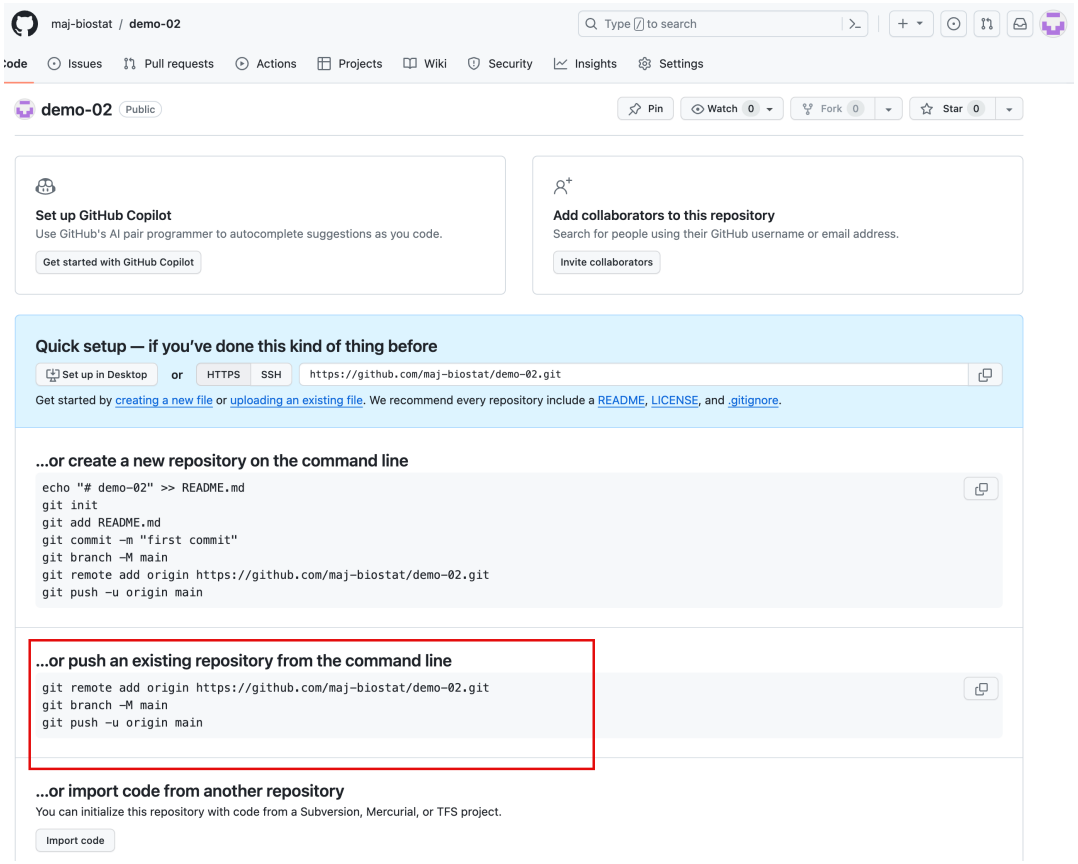


Figure 11.12: Link a local repo with the new remote

```
## Counting objects: 100% (68/68), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (67/67), done.
## Writing objects: 100% (68/68), 6.51 KiB | 3.25 MiB/s, done.
## Total 68 (delta 41), reused 0 (delta 0), pack-reused 0
## remote: Resolving deltas: 100% (41/41), done.
## To https://github.com/maj-biostat/demo2.git
##  * [new branch]      main -> main
## branch 'main' set up to track 'origin/main'.
```

> **Note**
>
> The `git branch -M main` simply renames the current branch to main. In practice, this isn't a problem if you create all of your repositories with a default branch called *main* instead of *master* which is the legacy git preferred name.

> **Note**
>
> The `git push -u` pushes up the branch information and the evolution of the repository. The `-u` is bsically telling git to link your local main branch with the remote main branch. You only need to use this flag once and thereafter you can use `git fetch` and `git push` and git will know which remote you want to communicate with.

When we refresh the github page https://github.com/maj-biostat/demo-02 (replacing my username with your own) then we see that the `first-repo` now appears on GitHub. It doesn't matter that the remote repo is called something other than `first-repo`. Generally you will give things the same names, but not always.

### 11.2.2 Clone a GitHub remote

In the first approach to initialising a remote (Section 11.2.1.1) we did not bring the remote down to the local machine. However, this is fairly trivial process to achieve by using `git clone` in a similar manner to how we used it previously.

Navigate back to the `demo-01` repository on GitHub, select the green clone button and copy the link to your clipboard.

Now head back to your terminal, change directory out of `first-repo` into the local workshop directory and run the following which clones the *demo-01* repository to a local directory named *demo-01*.

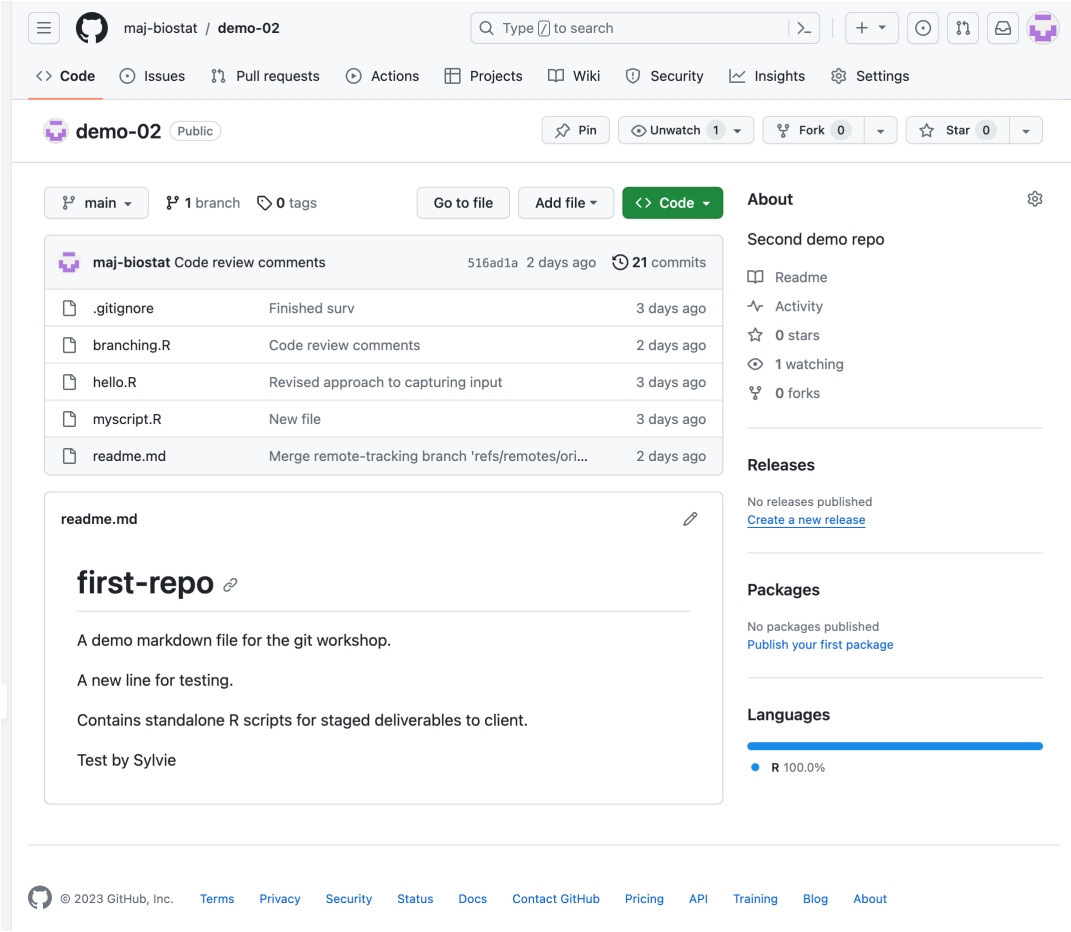


Figure 11.13: first-repo is now on GitHub

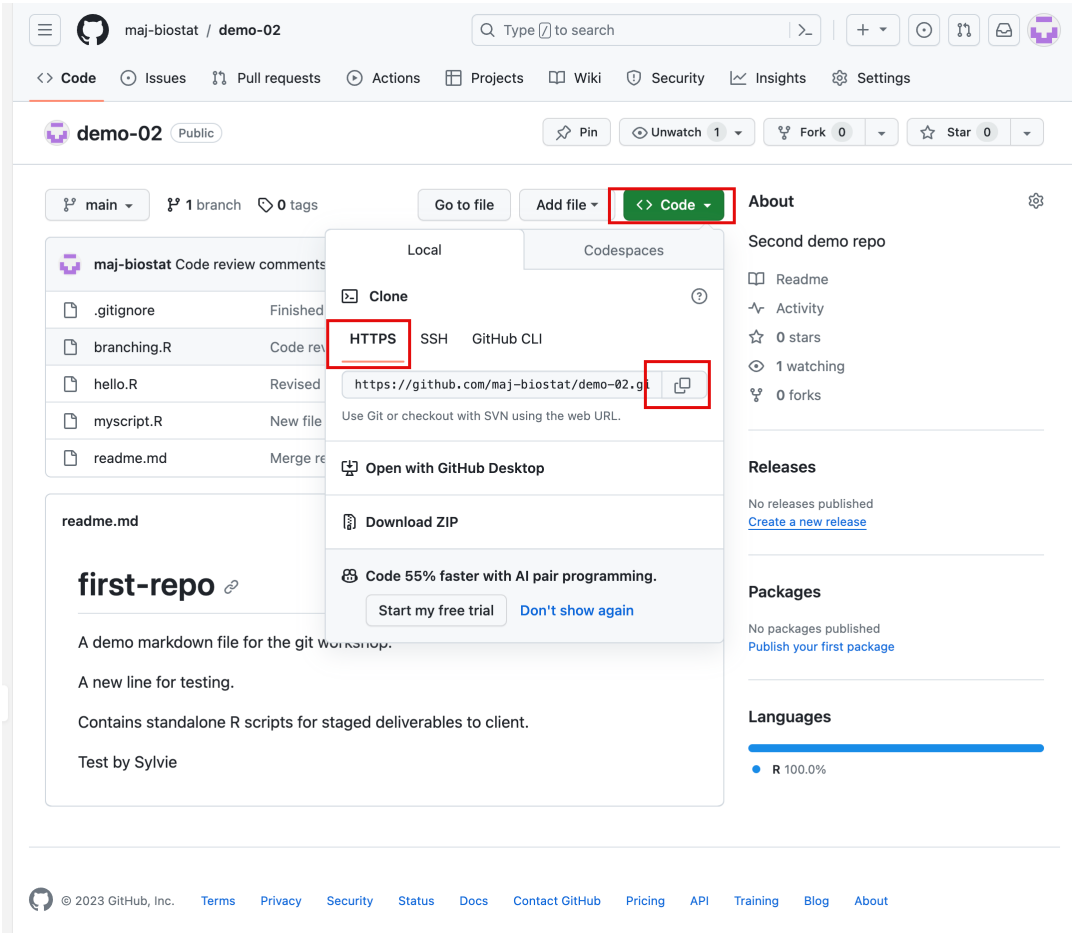


Figure 11.14: Cloning from GitHub

> **Note**
>
> Locally, you can give the remote repository any name you want. Here we have just used the same repository name as the remote, but we could call it *my-demo-01* or anything else you want.

```
git clone https://github.com/maj-biostat/demo-01.git demo-01
## Cloning into 'demo-01'...
## remote: Enumerating objects: 3, done.
## remote: Counting objects: 100% (3/3), done.
## remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
## Receiving objects: 100% (3/3), done.
cd demo-01/
ls -la
## total 8
## drwxr-xr-x  10 mark  staff  320  9 Nov 13:29 ..
## drwxr-xr-x   4 mark  staff  128  9 Nov 13:29 .
## -rw-r--r--   1 mark  staff   24  9 Nov 13:29 README.md
## drwxr-xr-x  12 mark  staff  384  9 Nov 13:29 .git
```

### 11.2.3 Fetch and push

Once you have linked your local and remote repositories, the workflow is identical to what has been previously introduced.

Let's work through the process again.

Following best practice, prior to making a change we create a feature branch and then switch to that branch.

```
git branch dev-01
git checkout dev-01
## Switched to branch 'dev-01'
```

Rather than code something up, simply download the random-numbers.R script (also listed on resources, Chapter 14 page) and save it to the demo-01 repository folder on your machine. Run the script with `Rscript random-numbers.R` to make sure that it works.

```
Rscript random-numbers.R
##         categorical      normal     gamma poisson
## Min.              -3 -3.56987408 0.0329983    3.00
```

```
## 1st Qu.          -2 -0.56524365 0.2845135    8.00
## Median            0  0.03029335 0.7235894    9.00
## Mean              0  0.07900788 1.0749329    9.82
## 3rd Qu.           2  0.74139889 1.3816200   12.00
## Max.              3  3.18586281 5.3176913   20.00
```

Now make an arbitrary change to the `README.md` file then stage and commit the files to the `dev-01` branch.

```
git status
## On branch dev-01
## Changes not staged for commit:
##   (use "git add <file>..." to update what will be committed)
##   (use "git restore <file>..." to discard changes in working directory)
##  modified:   README.md
##
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##  random-numbers.R
git add README.md random-numbers.R
git status
## On branch dev-01
## Changes to be committed:
##   (use "git restore --staged <file>..." to unstage)
##  modified:   README.md
##  new file:   random-numbers.R
git commit -m "Development for demo-01 project"
## [dev-01 fc5f874] Development for demo-01 project
## 2 files changed, 12 insertions(+), 1 deletion(-)
```

Switch back to the main branch and get ready to merge your piece of work into the local `main` branch[2].

```
git checkout main
```

Looking at the logs, you can see the `dev-01` branch is one step ahead of what we are currently viewing in our working directory.

```
git log --oneline --decorate --graph --all
## * fc5f874 (dev-01) Development for demo-01 project
```

[2] I know this is a bit artificial, but it is just for practice

```
## * 809fe51 (HEAD -> main, origin/main, origin/HEAD) Initial commit
```

and the working directory reflects this reality as only the `READ.md` file currently exists.

```
ls -la
## total 8
## drwxr-xr-x   4 mark  staff  128 10 Nov 11:44 .
## drwxr-xr-x  10 mark  staff  320 10 Nov 09:19 ..
## drwxr-xr-x  13 mark  staff  416 10 Nov 11:44 .git
## -rw-r--r--   1 mark  staff   26 10 Nov 11:44 README.md
```

Merge the `dev-01` branch into `main`

```
git merge dev-01
## Updating 809fe51..fc5f874
## Fast-forward
##  README.md        |  2 +-
##  random-numbers.R | 11 ++++++++++
##  2 files changed, 12 insertions(+), 1 deletion(-)
##  create mode 100644 random-numbers.R
```

We now have all the files and updates:

```
ls -la
## total 16
## drwxr-xr-x   5 mark  staff  160 10 Nov 11:49 .
## drwxr-xr-x  10 mark  staff  320 10 Nov 09:19 ..
## drwxr-xr-x  14 mark  staff  448 10 Nov 11:49 .git
## -rw-r--r--   1 mark  staff   66 10 Nov 11:49 README.md
## -rw-r--r--   1 mark  staff  203 10 Nov 11:49 random-numbers.R
```

And the logs indicate that we are now up to date (note the location of *HEAD*)

```
git log --oneline --decorate --graph --all
## * fc5f874 (HEAD -> main, dev-01) Development for demo-01 project
## * 809fe51 (origin/main, origin/HEAD) Initial commit
```

In this case, we know there have been no changes to the remote during the time where we were doing our development. However, in most collaborative projects, you cannot guarantee this, so we would run a `git fetch` followed by a `git merge` to merge any changes that had been pushed to the remote by our collaborators. Finally, we push our updates to the remote, remembering to use the `-u` flag on this first occasion.

```
git push -u origin main
## Enumerating objects: 6, done.
## Counting objects: 100% (6/6), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (4/4), done.
## Writing objects: 100% (4/4), 513 bytes | 513.00 KiB/s, done.
## Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
## To https://github.com/maj-biostat/demo-01.git
##    809fe51..fc5f874  main -> main
## branch 'main' set up to track 'origin/main'.
```

> **Note**
>
> If you wish to push all branches (i.e. both `main` and `dev-01`) in this case then you would need to add the `--all` flag to the `git push`. In the present example, this would push all the branches up to the remote. All people reviewing the remote can then review the complete history should they need to.

```
git push --all
## Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
## remote:
## remote: Create a pull request for 'dev-01' on GitHub by visiting:
## remote:      https://github.com/maj-biostat/demo-01/pull/new/dev-01
## remote:
## To https://github.com/maj-biostat/demo-01.git
##  * [new branch]      dev-01 -> dev-01
```

And the change can be observed once the GitHub page is refreshed.

### 11.2.4 Reviewing history

GitHub provides an interface to review the changes that have occurred on the repo. Clicking on a file within your repository will take you to the file history page where you can review the evolution of the file.

Once the history page opens you can navigate to various versions of a file. One the left you can select the branch of interest (here main is shown). You can view the code or the commit with syntax highlighting or in raw form (which also lets you copy and paste the raw text should you need it). Over on the right are the latest commit point and an open to review the full timeline (History).

When you review the blame view, you can see determine who did what and when.

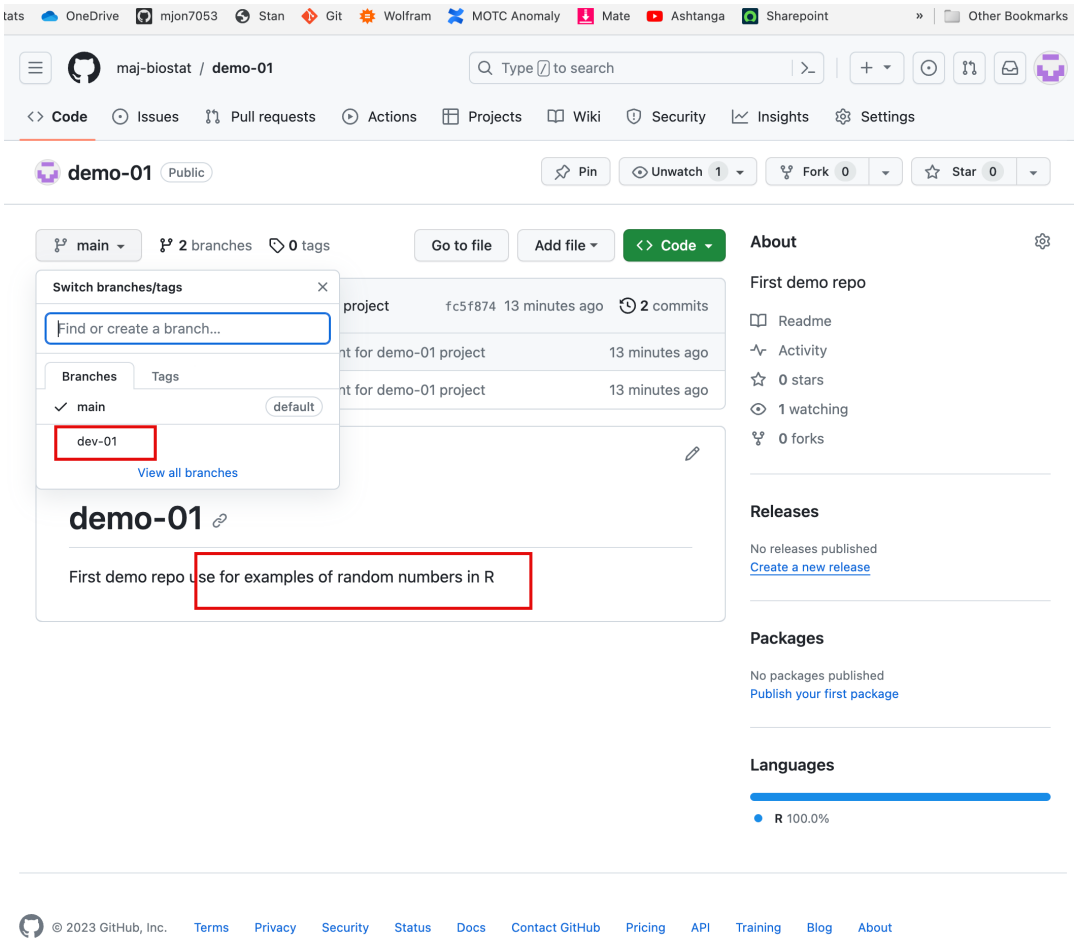


Figure 11.15: Updated files pushed to GitHub

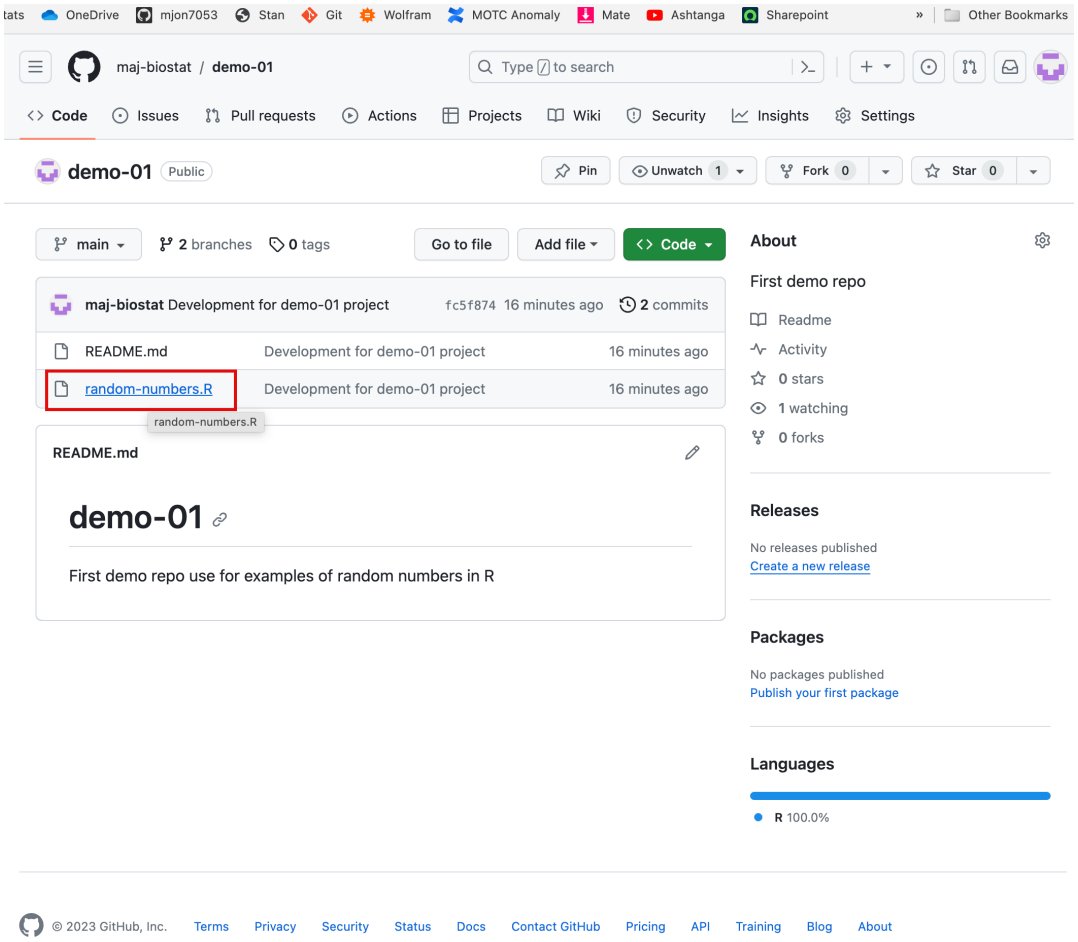


Figure 11.16: Updated readme and pushed to GitHub

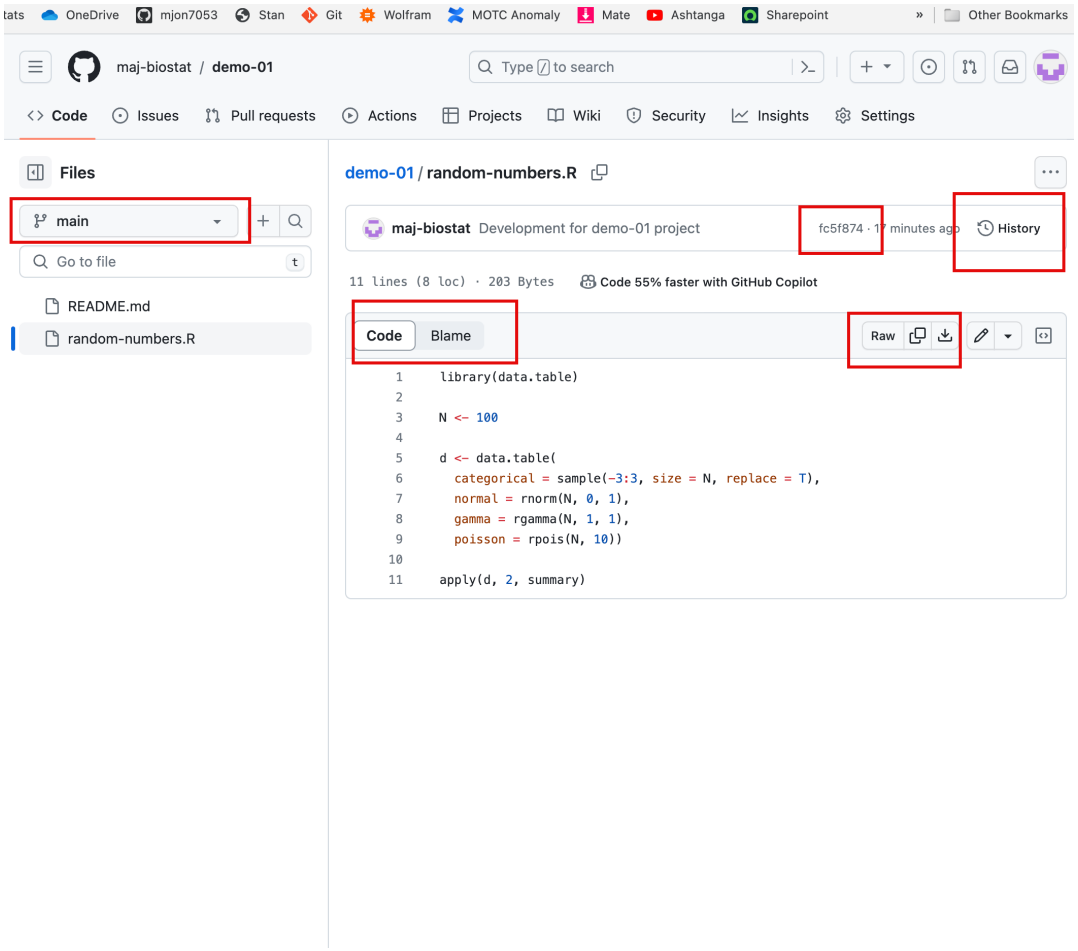


Figure 11.17: File review

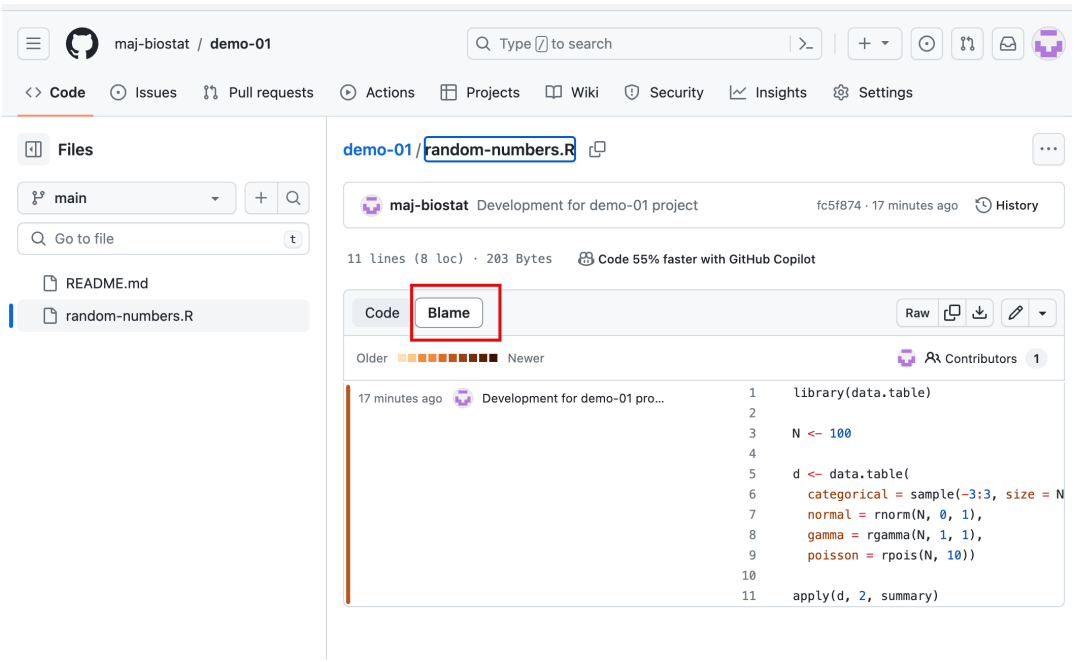


Figure 11.18: Code associated with commits

### 11.2.5 Issues

Issues can be managed directly within GitHub, although they are repository specific. Currently, our team mostly uses Jira to track and manage issues and therefore I will not cover the GitHub functionality beyond mentioning that it is very straight forward to use.

One instance where you may be required to use GitHub issue functionality is when you find a bug in a CRAN package, e.g. https://github.com/rstudio/gt/issues/1415. When doing so, you will find that they require a particular protocol to be followed that include the development of a minimal working example. This is standard practice in software development, it makes a developers life much easier and it is more likely that the problem you are reporting will actually get fixed.

## 11.3 Exercises

**Exercise 11.1.** When creating a new branch, we have needed to take to steps. First, we created the branch and then we checked it out. Is there a way this can be done with one command? Hint - Yes What is that command?

# 12 GitHub - forks and pull requests

We have considered settings where an indiviual creates a repository to which you (and/or others) may be added as collaborators and then contribute via the shared repository workflow introduced in the fundamentals.

However, there are circumstances where you will not be a part of a project team but you would like to contribute to a project. For example, you may want to contribute to an existing CRAN R package for which the code is stored on GitHub, as many now are.

There is a lot of documentation (some of it infinitely and frustratingly dense) on forks and pull requests that I have attempted to summarise, probably poorly, and combined with my own understanding obtained from experience. This section is therefore necessarily incomplete in that it does not reflect all of the possible considerations and functionality. Hopefully, it will give you some understanding and some practice and you can develop as need be from there.

### 12.0.1 Fork repository

Forks are a GitHub function that permit a complete copy of any public repository on GitHub, effectively taking a snapshot at a specific point in time. If you are not the owner or a collaborator on a project then the only way that you will be able to make any changes to the files is if you take a fork.

Clicking the fork button gives you a configuration screen where you can give your own name for the copy when it shows up in your account.

A fork is (more or less) a clone that happens within the bounds of the GitHub hosting service. After you take a fork, you would then clone as usual from that fork (that now sits in your account) as a remote to create a local repo.

You may choose to fork a repository for several reasons, here are four.

1. As a way to contribute to a project (e.g. an R package)
2. To use someone's code as a starting point (e.g. simulation code)
3. To experiment with some code
4. To use one of your own projects as the basis for a new project[1]

[1] Yes, you can fork your own repos, although a template repository might be more suitable depending on your goals.

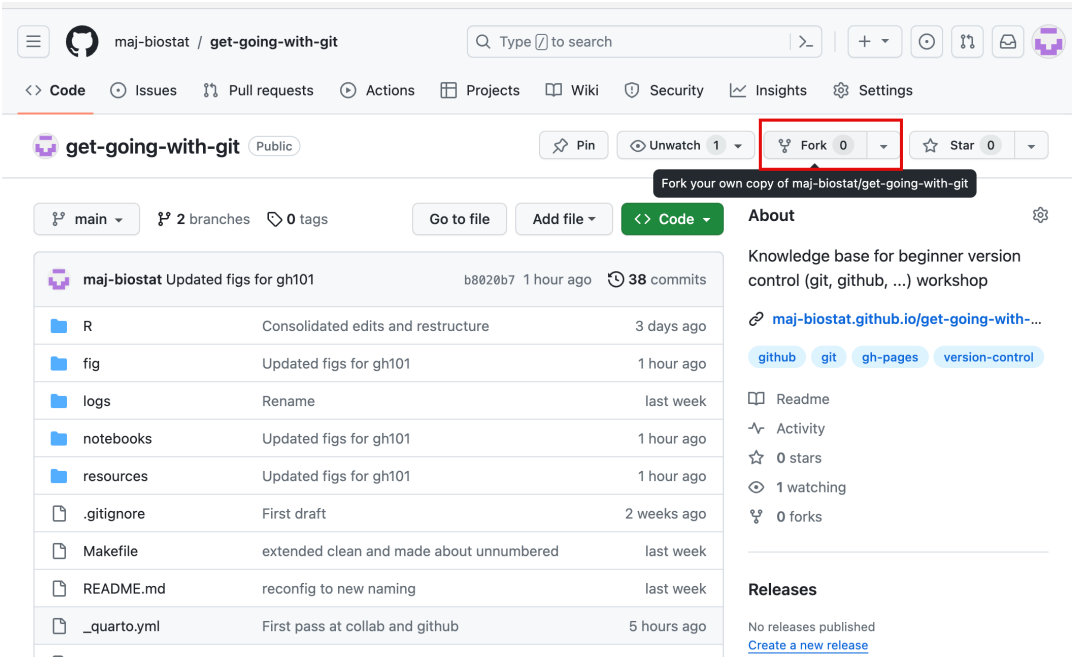


Figure 12.1: Forking on GitHub

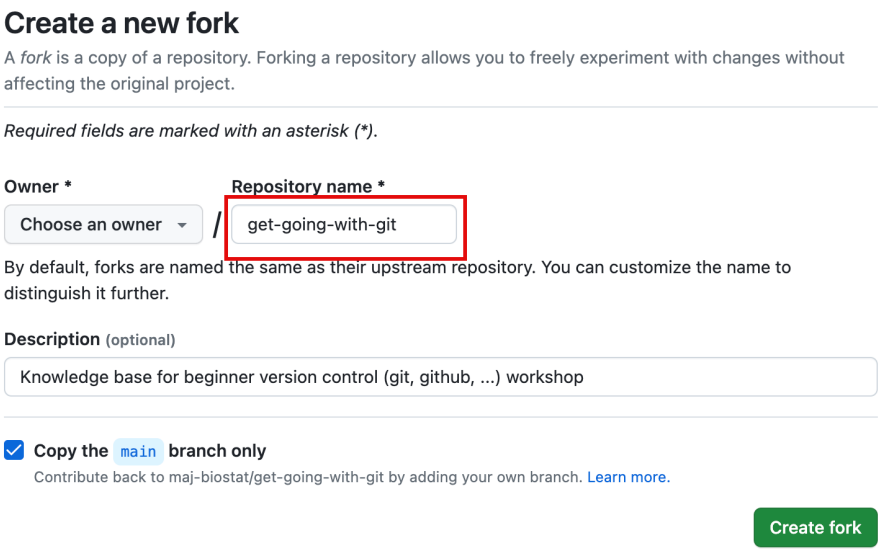


Figure 12.2: Forking configuration on GitHub

Based on the process above, the fork will no longer be directly attached to the original repository but you can take an additional step (see later) so that you have the original repository as a remote in order to keep your version up to date.

> ⚠ Warning
>
> 1. If you fork from a private repository, when/if that repository is deleted, all forks are deleted.
> 2. If you fork from a public repository and the upstream repository is deleted, one of the public forks is selected to be the new upstream repository.

Let's use the digest CRAN package as a quick example. Go to the GitHub source, click fork, accept the defaults and click *create* and then clone the repository to your local machine as shown below.

```
git clone https://github.com/maj-biostat/digest.git
## Cloning into 'digest'...
## remote: Enumerating objects: 3174, done.
## remote: Counting objects: 100% (843/843), done.
## remote: Compressing objects: 100% (274/274), done.
## remote: Total 3174 (delta 471), reused 826 (delta 463), pack-reused 2331
## Receiving objects: 100% (3174/3174), 2.15 MiB | 3.25 MiB/s, done.
## Resolving deltas: 100% (2034/2034), done.
```

There is some common terminology that relates to the fork process, so I will introduce next.

When you clone the forked repository to your local machine, the forked repository (the one now sitting in your account) is considered as the **remote origin**, and the repository you forked from (the original one belonging to Dirk) is **upstream**. The picture looks like this, try to keep it in mind.

Once you have the local repo, you can edit the code and push your changes to your account.

> ⚠ Warning
>
> If you have it in mind that you would like to push back to the upstream source at some point in the future, it is advisable (actually strongly advisable) to leave your forks `main` branch the hell alone and work purely off development branches. *That is,* **do not** *push to your main branch and keep all of your features within their own dedicated feature branches.* This is because if you keep things in branches, it is much easier to see what is yours and what is theirs. That isn't a complete reason, but this is one of those rules that if will make your life easier if you just follow it without too much question.

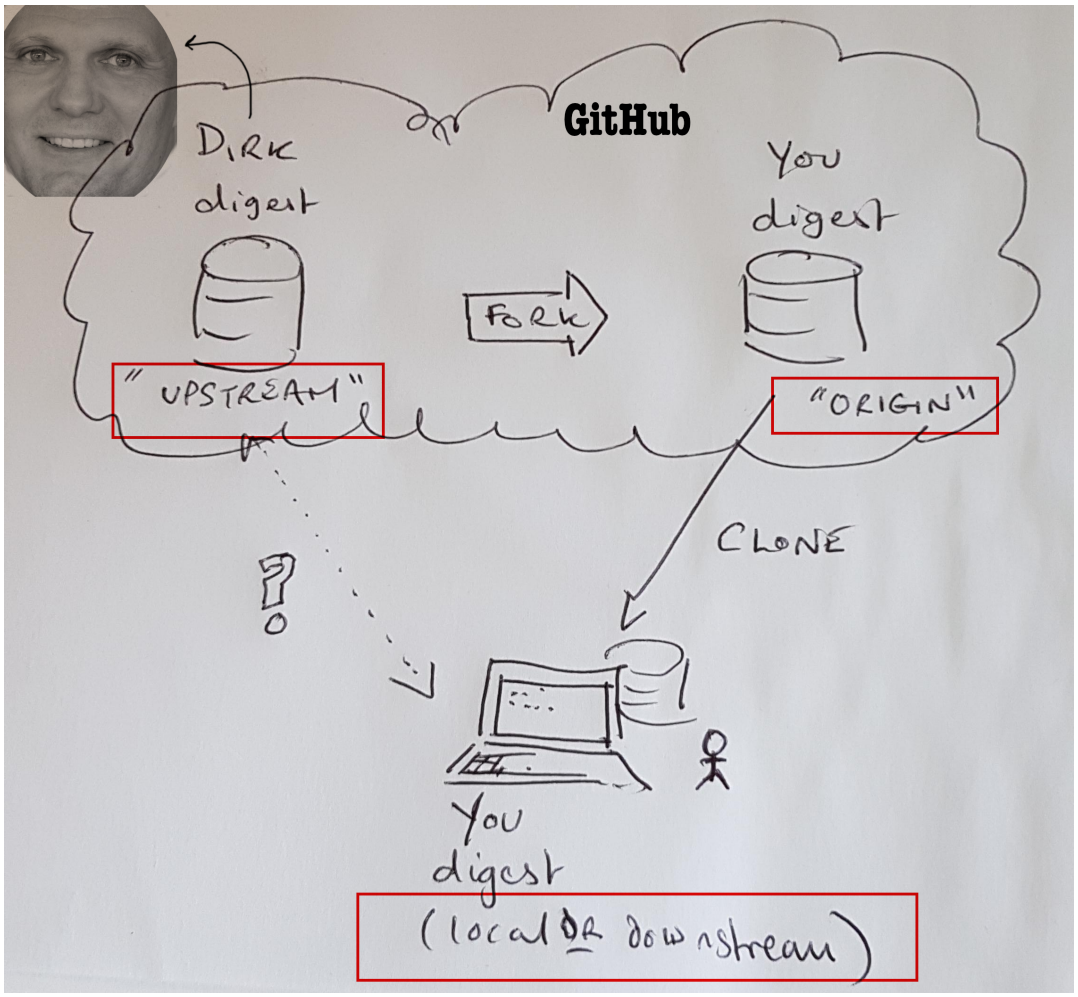


Figure 12.3: Upstream, origin, local

As mentioned, you would not be able to push directly to Dirk's source as you are not a collaborator on that project. Initially, you won't even be able to keep up to date with Dirk's work because there is no direct link between his repo and your local. To make this link, you need to add Dirk's repo as the upstream remote:

```
git remote add upstream https://github.com/eddelbuettel/digest.git
git remote -v
## origin   https://github.com/maj-biostat/digest.git (fetch)
## origin   https://github.com/maj-biostat/digest.git (push)
## upstream https://github.com/eddelbuettel/digest.git (fetch)
## upstream https://github.com/eddelbuettel/digest.git (push)
```

This will make the dotted line in the figure above into a solid line. The final thing you want to do is to track the upstream main branch instead of the origin main branch. This makes sure that you are keeping your repo up to date with any changes that Dirk's is making on his upstream source.

```
git branch -u upstream/main
```

After that, you should be good to go.

As time goes on, you can now fetch the latest changes that Dirk has made and merge them in with any local modifications you have made.

```
git fetch upstream
## remote: Enumerating objects: 30, done.
## remote: Counting objects: 100% (27/27), done.
## remote: Compressing objects: 100% (8/8), done.
## remote: Total 18 (delta 12), reused 15 (delta 9), pack-reused 0
## Unpacking objects: 100% (18/18), 4.95 KiB | 337.00 KiB/s, done.
## From https://github.com/eddelbuettel/digest
##  * [new branch]      feature/crc32c     -> upstream/feature/crc32c
##  * [new branch]      feature/redo_pr195 -> upstream/feature/redo_pr195
##  * [new branch]      master             -> upstream/master
```

> **Note**
>
> I believe the above process (or at least some of it) is also possible through the GitHub interface but I have not used that approach.

If you are keen for Dirk to consider your changes for inclusion in a release of the `digest` package, then you will need to prepare a pull request.

### 12.0.2 Pull requests

A pull request basically tells others that you have pushed a set of changes to a branch in a repository on GitHub. *Pull requests* are not managed by git, these are something that GitHub implemented as a value-add. However, git does integrate pull requests as part of its distributed design, it simply does not care what mechanism is used to implement the pull request. As far as git is concerned, a pull request could simply be an email from you to some third party that informs them about your change and gives them the option to pull the change into their repository. On top of a line of communication, GitHub adds on things like a formal framework for providing review comments on the proposed changes.

Pull requests are (or can be) applicable under both the *fork and pull* workflow processes and the *shared repository* workflow. In the shared repository workflow, pull requests could be useful as a means to initiate code review prior to development branches being merged into the main branch. Here, we will only cover pull requests for the *fork and pull* workflow, but the process for a share repository is very similar.

In brief, the steps are:

1. Fork and clone the repository on which you want to contribute
2. Create a new branch for your development
3. Develop
4. Push your branch to your fork
5. Create a pull request, providing the requisite description of the change
6. Submit the pull request
7. Address any reviewer comments
8. Push request is approved and merged

### 12.0.3 Pull request demo

Again, doing will make this more concrete. In order that we have do all the steps, I created the aliasr repository under the GitHub user t-student.

Just listen to this first and then you can have a go.

#### 12.0.3.1 Fork and clone repo

From my maj-biostat work account, I navigate to the aliasr repo and click the fork button. Note that I changed to dark theme so that you can distinguish the two different accounts.

Now that I have forked the R package repo, I can clone it to my local machine in the usual manner.

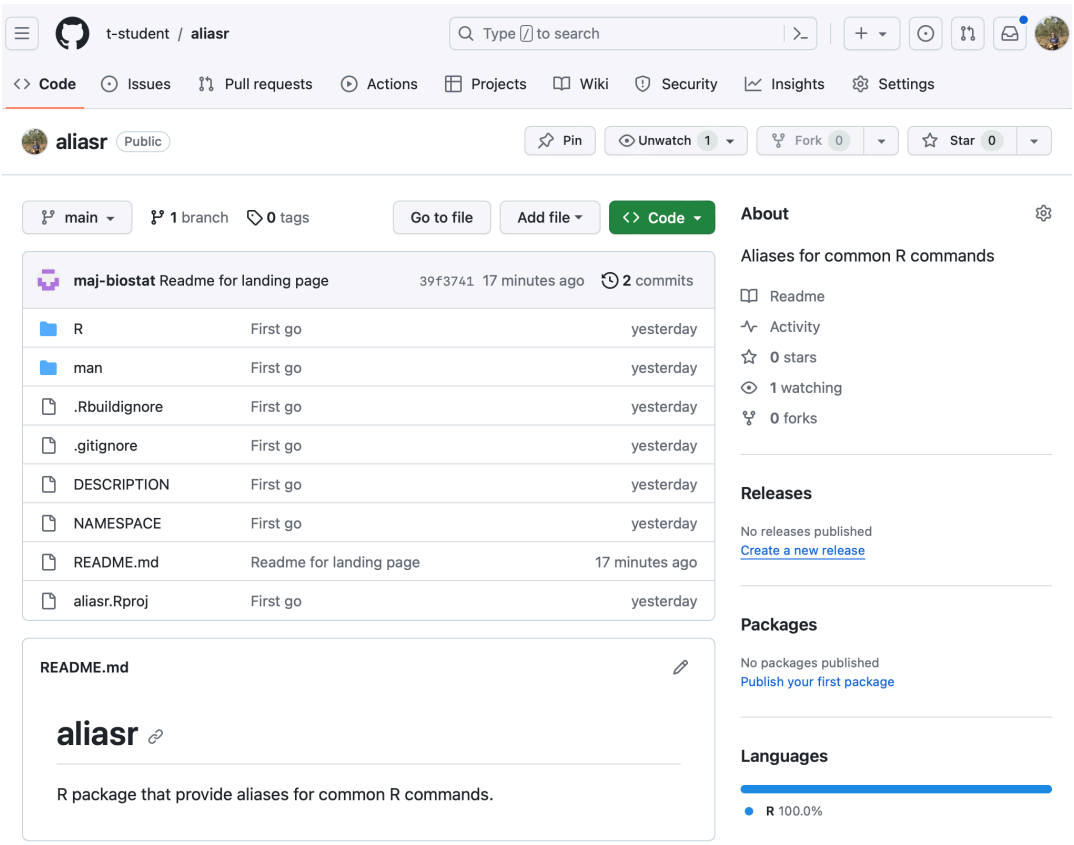


Figure 12.4: aliasr repo

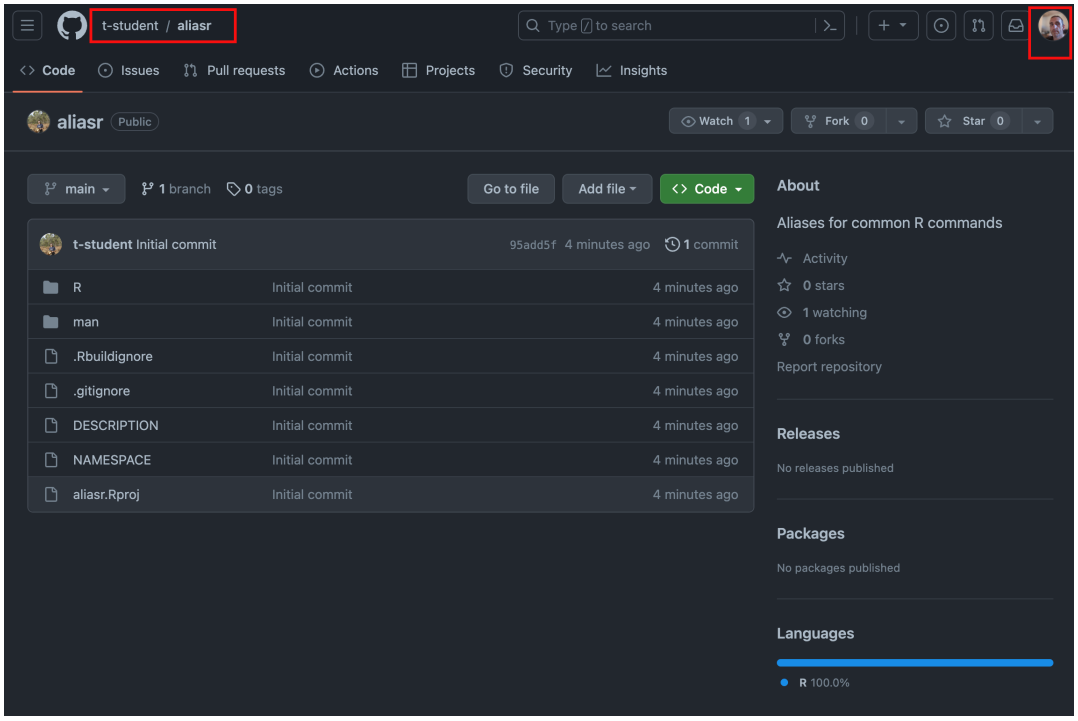


Figure 12.5: Fork aliasr repo

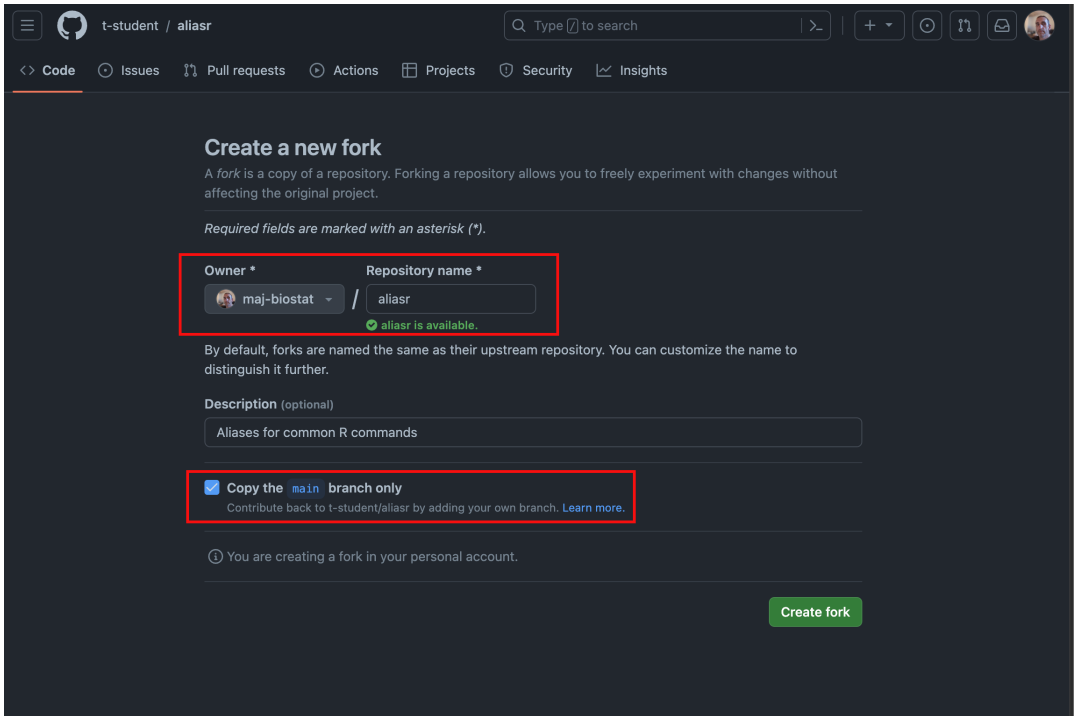


Figure 12.6: Fork aliasr repo settings

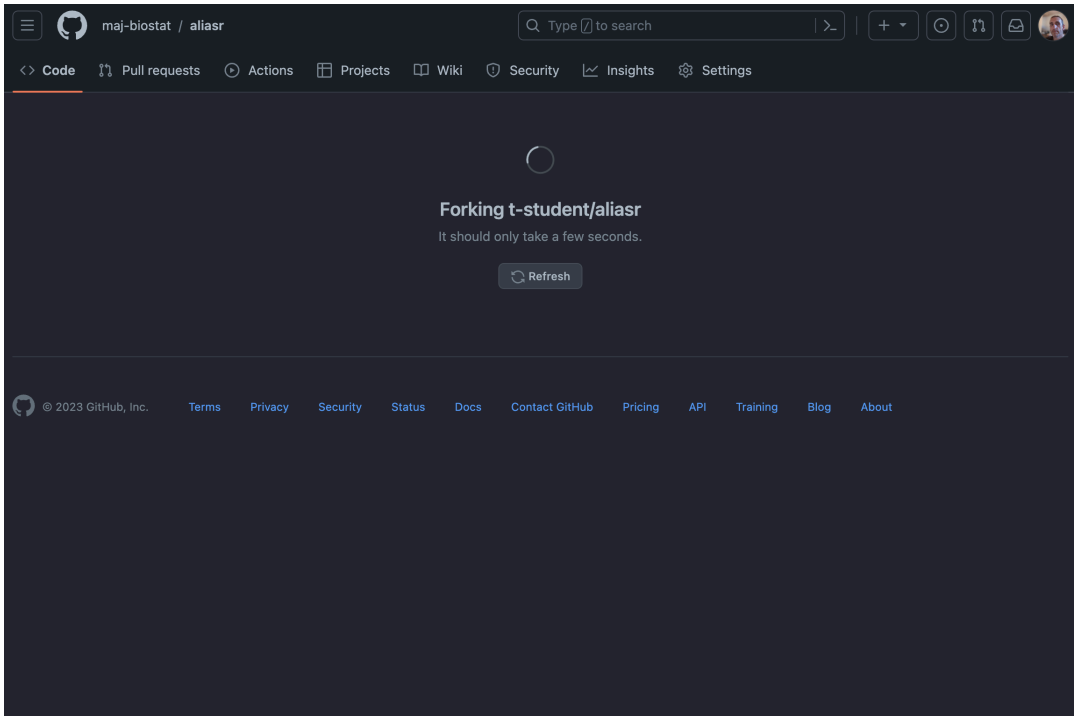


Figure 12.7: Forking...

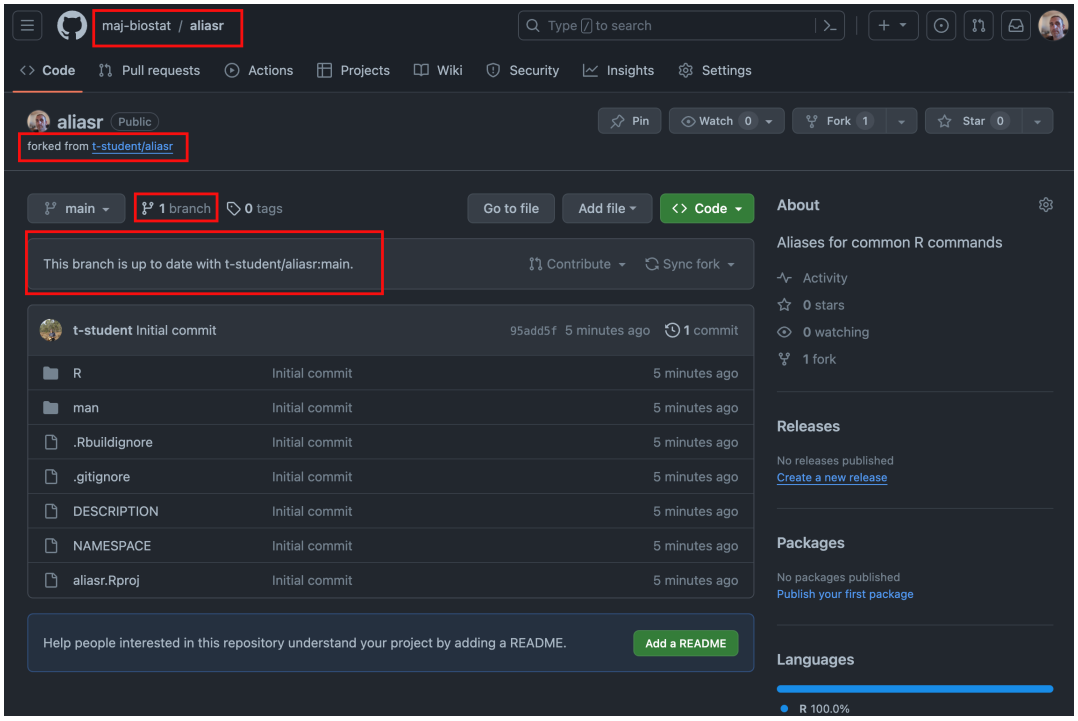


Figure 12.8: Forked repo in maj-biostat dir

```
git clone https://github.com/maj-biostat/aliasr.git
## Cloning into 'aliasr'...
## remote: Enumerating objects: 11, done.
## remote: Counting objects: 100% (11/11), done.
## remote: Compressing objects: 100% (6/6), done.
## remote: Total 11 (delta 0), reused 11 (delta 0), pack-reused 0
## Receiving objects: 100% (11/11), done.
```

Before I do anything else, from the terminal I change directory to `aliasr` and then add the `t-student` repo as the upstream.

```
git remote -v
## origin   https://github.com/maj-biostat/aliasr.git (fetch)
## origin   https://github.com/maj-biostat/aliasr.git (push)
git status
## On branch main
## Your branch is up to date with 'origin/main'.
##
## nothing to commit, working tree clean
git remote add upstream https://github.com/t-student/aliasr.git
git remote -v
## origin   https://github.com/maj-biostat/aliasr.git (fetch)
## origin   https://github.com/maj-biostat/aliasr.git (push)
## upstream https://github.com/t-student/aliasr.git (fetch)
## upstream https://github.com/t-student/aliasr.git (push)
```

And then I first bring any commits down from the upstream and then tell git to track the upstream main branch instead of my remote origin.

> **Note**
>
> If you miss the `git fetch upstream` you will not have access to the main branch and the `git branch -u upstream/main` command will fail as a result.

```
git fetch upstream
## From https://github.com/t-student/aliasr
##  * [new branch]      main       -> upstream/main

git branch -u upstream/main
## branch 'main' set up to track 'upstream/main'.
```

#### 12.0.3.2 Feature branch and feature development

Remembering best practice, I create a new branch in my local repo for my feature development and switch to that branch.

```
git branch feat-csum
git checkout feat-csum
```

Ok, so if I look at the commit logs in my local repo I have the following:

```
git log --all
## commit 8e9007a5ca648770671a1f0b12c352b6690131cd (HEAD -> feat-csum, upstream/main, orig
## Author: Mark Jones <maj684@gmail.com>
## Date:   Wed Nov 15 07:24:02 2023 +0800
##
##     Initiall commit
```

Opening the project in rstudio (or VIM if you are so inclined), I add a new function to the end of the `abbrv.R` script

```
csum <- function(x, na.rm = T){
  x_new <- x[!is.na(x)]
  c_x_new <- cumsum(x_new)
  c_x <- x
  c_x[!is.na(c_x)] <- c_x_new
  c_x
}
```

And can test it with the usual procedures for building and installing an R Package, namely:

```
cd ..
R CMD build aliasr
R CMD INSTALL aliasr_0.1.0.tar.gz
```

and then load that library in an R session and test the new csum function

```
## R version 4.3.0 (2023-04-21) -- "Already Tomorrow"
## Copyright (C) 2023 The R Foundation for Statistical Computing
## Platform: aarch64-apple-darwin20 (64-bit)
##
## R is free software and comes with ABSOLUTELY NO WARRANTY.
## You are welcome to redistribute it under certain conditions.
```

```
## Type 'license()' or 'licence()' for distribution details.
##
##   Natural language support but running in an English locale
##
## R is a collaborative project with many contributors.
## Type 'contributors()' for more information and
## 'citation()' on how to cite R or R packages in publications.
##
## Type 'demo()' for some demos, 'help()' for on-line help, or
## 'help.start()' for an HTML browser interface to help.
## Type 'q()' to quit R.

library(aliasr)
csum(c(1, NA, 3, 5))
[1]  1 NA  4  9
```

Based on this rather superficial testing, I stage and commit.

```
git add R/abbrv.R
git commit -m "Implementation of cumsum alias"
## [feat-csum 5e5be52] Implementation of cumsum alias
##  1 file changed, 7 insertions(+)
```

And now I push the feature branch to my fork (here I am specifying both the remote and the branch).

```
git push origin feat-csum
## Enumerating objects: 7, done.
## Counting objects: 100% (7/7), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (3/3), done.
## Writing objects: 100% (4/4), 442 bytes | 442.00 KiB/s, done.
## Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
## remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
## remote:
## remote: Create a pull request for 'feat-csum' on GitHub by visiting:
## remote:      https://github.com/maj-biostat/aliasr/pull/new/feat-csum
## remote:
## To https://github.com/maj-biostat/aliasr.git
##  * [new branch]      feat-csum -> feat-csum
```

#### 12.0.3.3 Sync with upstream (and merge conflicts)

I am all ready and excited to create a pull request, but reviewing my fork on GitHub, I see that there has been a covert change to the upstream repository while I have been developing the `csum` feature. Specifically, I appear to be 2 commits behind the upstream repo.

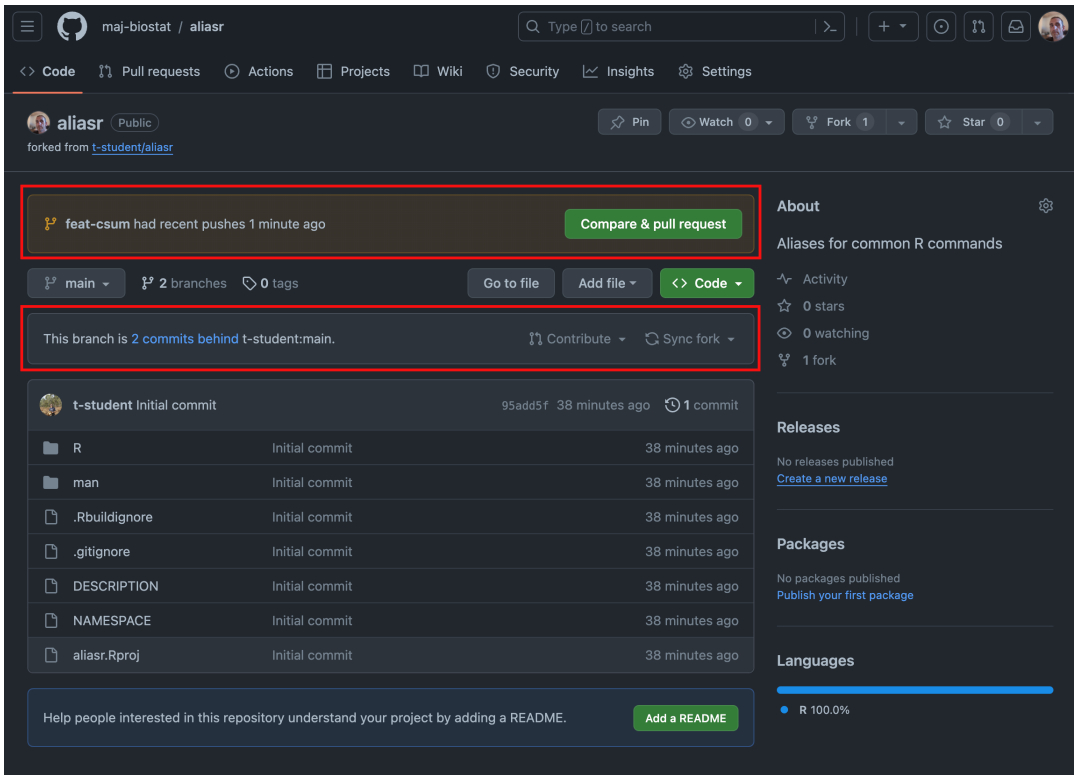


Figure 12.9: Pushed new feature to fork

To address this, I need to merge the changes.

```
git fetch upstream
## remote: Enumerating objects: 10, done.
## remote: Counting objects: 100% (10/10), done.
## remote: Compressing objects: 100% (2/2), done.
## remote: Total 7 (delta 3), reused 7 (delta 3), pack-reused 0
## Unpacking objects: 100% (7/7), 647 bytes | 92.00 KiB/s, done.
## From https://github.com/t-student/aliasr
##    8e9007a..14b1c35  main       -> upstream/main
```

Urgh. It was all going so well. Merge conflicts are common and, if you use git, you are going to see this message sooner or later.

```
git merge upstream/main
## Auto-merging R/abbrv.R
## CONFLICT (content): Merge conflict in R/abbrv.R
## Automatic merge failed; fix conflicts and then commit the result.
```

Running status, we can see that git merged the new README.md into our repository with no problem but it had difficulties with the `abbrv.R` script.

```
git status
## On branch feat-csum
## You have unmerged paths.
##   (fix conflicts and run "git commit")
##   (use "git merge --abort" to abort the merge)
##
## Changes to be committed:
##  new file:   README.md
##
## Unmerged paths:
##   (use "git add <file>..." to mark resolution)
##  both modified:   R/abbrv.R
```

Opening the `R/abbrv.R` script, you will be able to see conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) have been incorporated by git where the merge failed. You see the changes from the HEAD (or base branch) after the line `<<<<<<< HEAD`. In this case, this is what you just introduced. You can also see `=======` dividing the changes from the other branch, i.e. those that are in the upstream version.

```
# TOP PART OF SCRIPT OMITTED FOR BREVITY

csum <- function(x, na.rm = T){
  x_new <- x[!is.na(x)]
  c_x_new <- cumsum(x_new)
<<<<<<< HEAD
  c_x <- x
  c_x[!is.na(c_x)] <- c_x_new
  c_x
=======
  c_x_new
>>>>>>> upstream/main
}
```

At this point you need to decide what you want to keep and so edit the script to the following.

```
# TOP PART OF SCRIPT OMITTED FOR BREVITY

csum <- function(x, na.rm = T){
  x_new <- x[!is.na(x)]
  c_x_new <- cumsum(x_new)
  c_x <- x
  c_x[!is.na(c_x)] <- c_x_new
  c_x
}
```

Stage and commit.

```
git add R/abbrv.R
git commit -m "Resolved merge conflict by retaining majbiostat implementation."
```

As always, the log shows the history of the repository. We can see the two commits that were made in the upstream repo while I was working on the `feat-csum` branch.

```
git log --all
## commit 9c949d06702c6513409e7a2119466437ad060a87 (HEAD -> feat-csum)
## Merge: c615c0c 14b1c35
## Author: Mark Jones <mark.jones1@sydney.edu.au>
## Date:   Wed Nov 15 08:00:32 2023 +0800
##
##     Resolved merge conflict by retaining majbiostat implementation.
##
## commit 14b1c3514033c88155f6a15f87ac9fc3ed915fe6 (upstream/main)
## Author: Mark Jones <maj684@gmail.com>
## Date:   Wed Nov 15 07:55:40 2023 +0800
##
##     Csum implementation
##
## commit cc35ec0a25dab3e388d704b2ca19808808629ed9
## Author: Mark Jones <maj684@gmail.com>
## Date:   Wed Nov 15 07:54:05 2023 +0800
##
##     Readme for landing page
##
## commit c615c0c65e9225436c96bb00df1cb64d0194a4d2 (origin/feat-csum)
## Author: Mark Jones <mark.jones1@sydney.edu.au>
## Date:   Wed Nov 15 07:28:06 2023 +0800
##
```

```
##     Implementation of cumsum alias
##
## commit 8e9007a5ca648770671a1f0b12c352b6690131cd (origin/main, origin/HEAD, main)
## Author: Mark Jones <maj684@gmail.com>
## Date:   Wed Nov 15 07:24:02 2023 +0800
##
##     Initiall commit
```

Based on this, I can push to my fork:

```
git push origin feat-csum
## Enumerating objects: 4, done.
## Counting objects: 100% (4/4), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (2/2), done.
## Writing objects: 100% (2/2), 327 bytes | 327.00 KiB/s, done.
## Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
## remote: Resolving deltas: 100% (1/1), completed with 1 local object.
## To https://github.com/maj-biostat/aliasr.git
##    c615c0c..9c949d0  feat-csum -> feat-csum
```

and if I want to be tidy, I can switch to local/main and run a fetch then merge in order that my main branch is also in sync with the upstream. After that I can push to main. In brief:

```
git checkout main
git fetch upstream
git merge
git push origin
```

> **Note**
>
> Strictly speaking, I believe the above isn't really necessary, but it gets rid of some notifications at the GitHub end that can be a bit concerning if taken on face value.

#### 12.0.3.4 Create PR

With all that done, your fork should look something like the following.

Clicking the `compare & pull request` intiations the pull request process.

Note:

1. The source and the targets along with GitHubs comment that merging is possible.

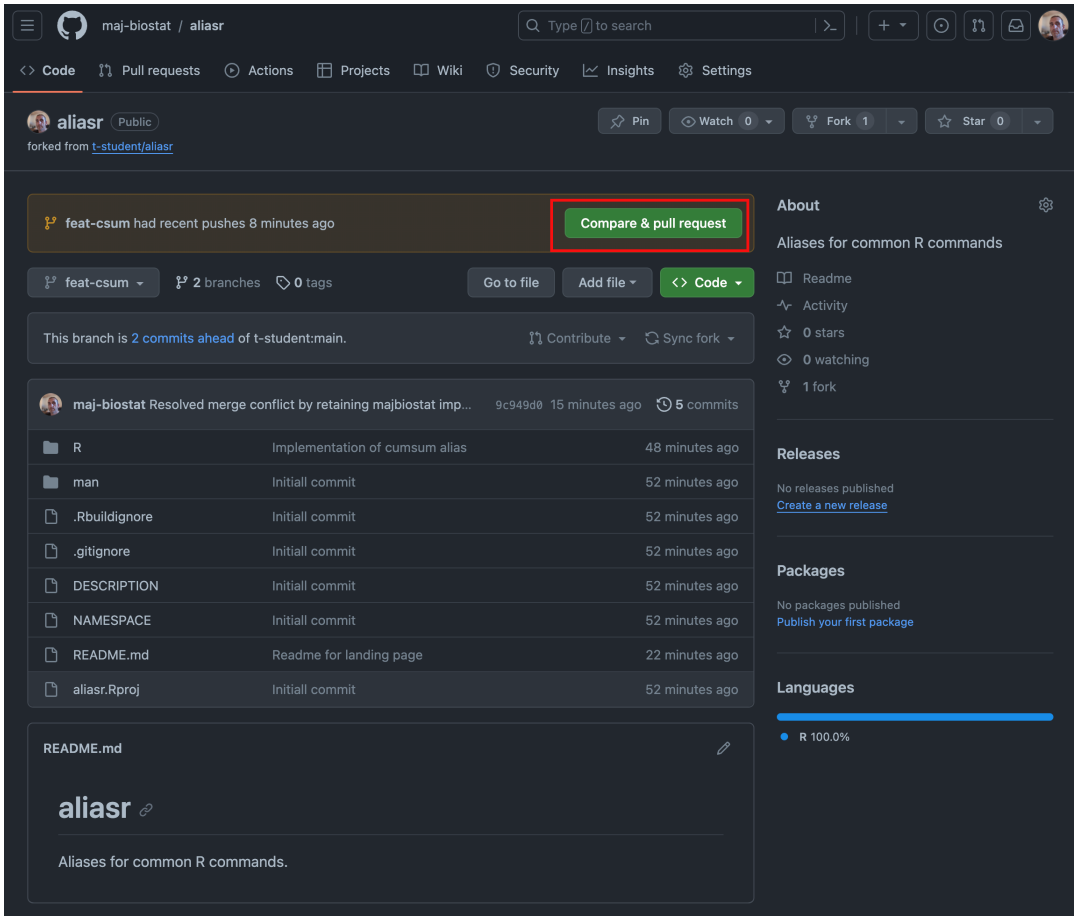


Figure 12.10: Sync'd fork

2. A description providing an overview of the feature
3. Allowing edits by maintainers (i.e. those with appropriate privaleges on the upstream repo can commit to your branch)
4. A visual diff of the before and after implementation of the impacted script (or scripts)

Clicking on create will provide you with a PR summary:

#### 12.0.3.5 Incorporating PR into upstream repo

Heading back to the upstream repo (and importantly under my `t-student` account) I can see that a pull request has been created. Three options are available to do the merge

1. Merge
2. Squash and merge
3. Rebase and merge

You can pick one of the above options (but I am not sure how you would rewind if things go wrong) or you can use the command line.

The first thing you need to do is fetch the pull request detail. To do this you specify `pull/<PR ID>/head:<branch name>` where the `<PR ID>` was listed in the PR page and the `<branch name>` is just some arbitrary branch name you want to give this feature. Again, at the terminal, as the owner of the upstream repo:

```
git fetch origin pull/1/head:feat001
## remote: Enumerating objects: 12, done.
## remote: Counting objects: 100% (12/12), done.
## remote: Compressing objects: 100% (2/2), done.
## remote: Total 6 (delta 3), reused 6 (delta 3), pack-reused 0
## Unpacking objects: 100% (6/6), 698 bytes | 116.00 KiB/s, done.
## From github.com:t-student/aliasr
##  * [new ref]         refs/pull/1/head -> feat001
```

From here, as the owner of the upstream repo and the person that ultimately decides whether the feature will be incorporated into the aliasr package or not, you can make whatever edits you want by simply checking out `feat001`. After you are satisfied, then you can merge back into `main` and push up to the repo.

> **Note**
>
> The `--no-ff` says *no fast forward*. This is a shortcut method git can take when there are no local changes. Use this flag when you want the history of merging a feature from somewhere else.

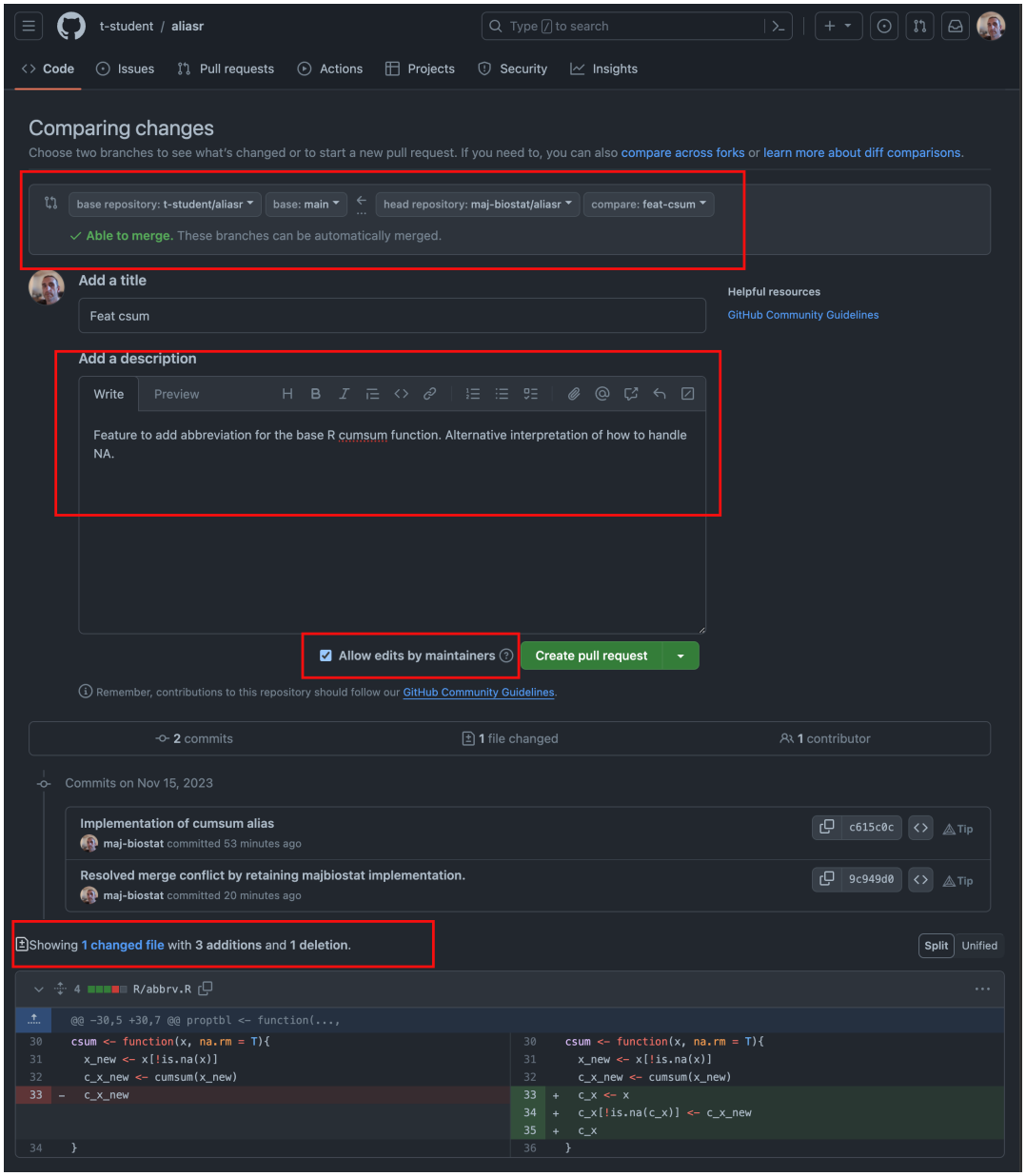


Figure 12.11: Open a pull request

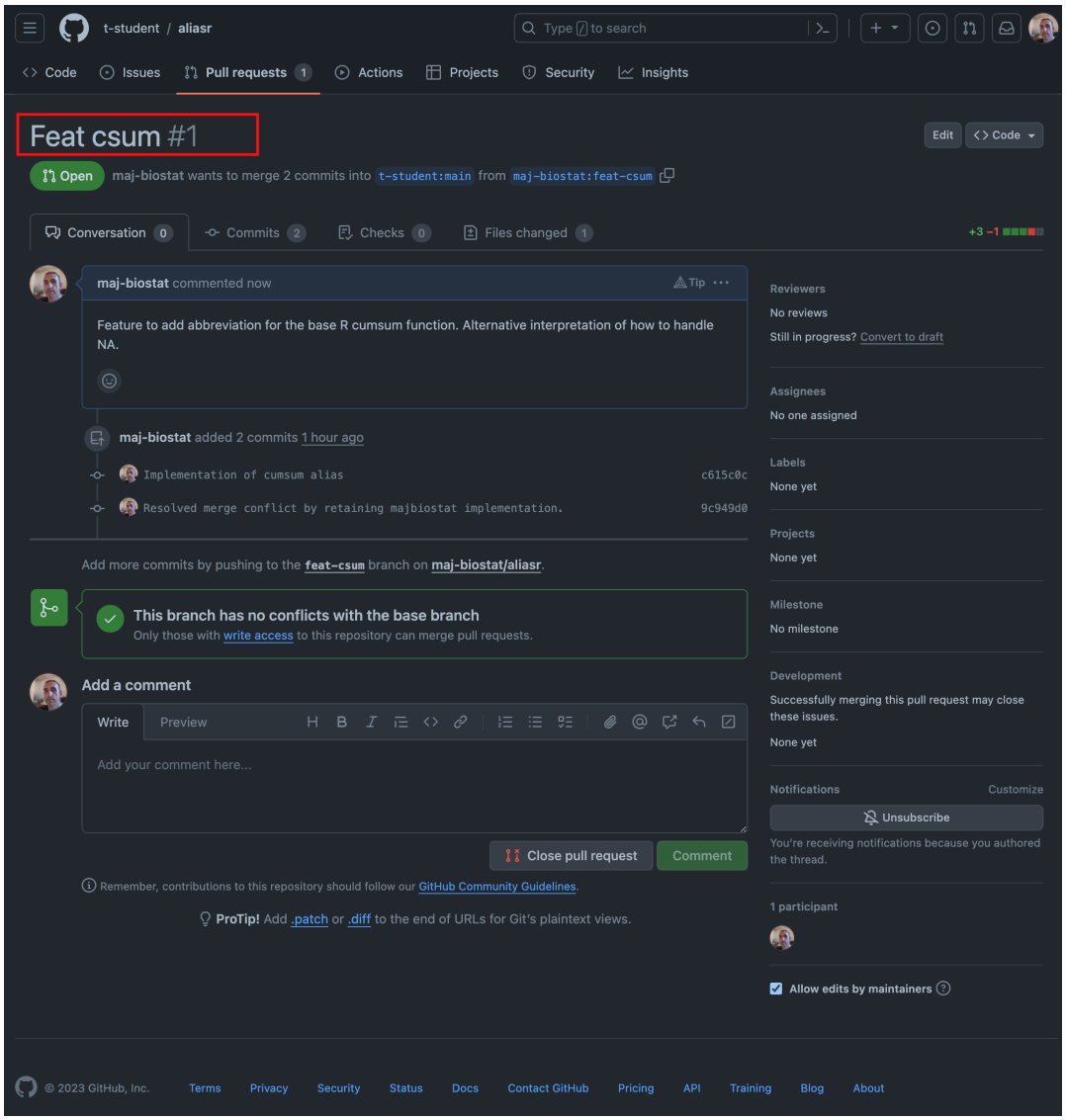


Figure 12.12: Pull request summary

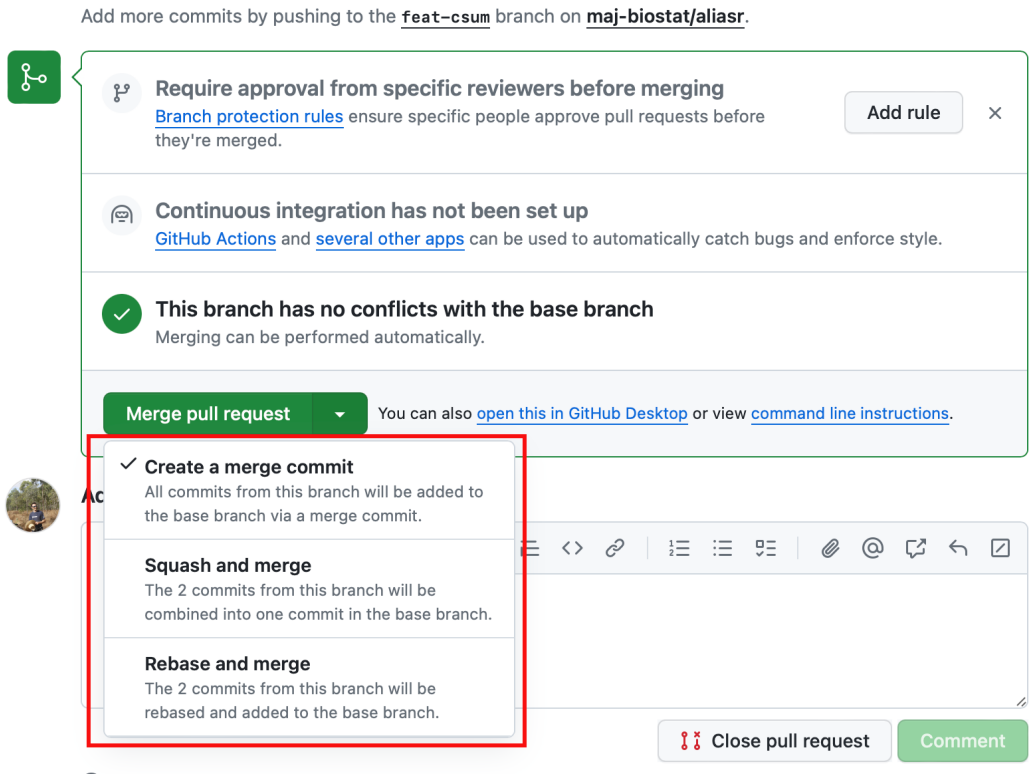


Figure 12.13: Merge PR

```
git checkout main
git merge --no-ff feat001
## Merge made by the 'ort' strategy.
##  R/abbrv.R | 4 +++-
##  1 file changed, 3 insertions(+), 1 deletion(-)

git push origin main
## Enumerating objects: 13, done.
## Counting objects: 100% (13/13), done.
## Delta compression using up to 8 threads
## Compressing objects: 100% (6/6), done.
## Writing objects: 100% (7/7), 814 bytes | 814.00 KiB/s, done.
## Total 7 (delta 3), reused 0 (delta 0), pack-reused 0
## remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
## To github.com:t-student/aliasr.git
##    14b1c35..a904f44  main -> main
```

Now, when you go back to the the aliasr you will see that the pull request has been closed.

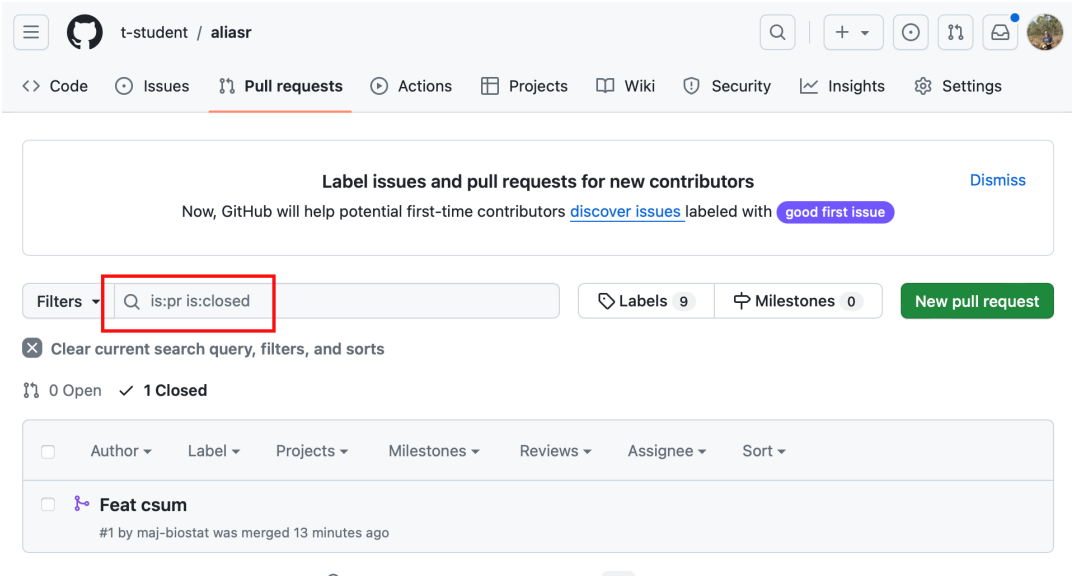


Figure 12.14: Closed PR

> **Note**
>
> I know. That was a lot. The objective here is to give you a preview or at least cursory understanding of the underlying concepts and required actions rather than just simply

knowledge of what buttons to press.

# 13 GitHub and documentation

## 13.1 Overleaf

If you are a Latex fan then Overleaf (depending on the version you have) provides some integration with GitHub.

For now, I am just providing a link to the instructions to set this up. I have used it in the past and it works OK, but I think that GitHub pages and Quarto provides a better alternative for project documentation.

# Part IV

# Part 4 - Other bits

Here are some additional bits that I did not know where to put anywhere else.

# 14 Resources

These files are used in **Part 2 - Fundamentals** from the text:

- hello.R
- readme.md
- branching.R
- .gitignore
- random-numbers.R

These files are used in **Part 3 - Collaboration** from the text:

- None currently required.

# 15 RStudio integration

When you create a new project in RStudio, you are given the option to create a git repository. This appear to effectively just run `git init` after creating the default project files.

If you select to create a git repository or open an exisiting project where you have initialised a git repository independently, then the user interface will have a series of new panels. These new panels comprise a git client (they are entirely dependent on git being installed on your machine). However, they give you an easy alternative to using the command line interface. The concepts that we have introduced here are directly transferable to the RStudio context.

# 16 Git client tools

## 16.1 Diff tools

Instead of parsing the online diff, you can opt to configure a specialised difftool. Delta is one such example. The documentation is good so I will not rehash it here. I mostly use the command line and if necessary GitHub, so I cannot really comment on how good Delta is.

# 17 Common commands

Table 17.1: Common git commands

| Command | Description of common use |
|---|---|
| git init | |
| git clone | |
| git add | |
| git reset | |
| git status | |
| git diff | |
| git log | |
| git commit | |
| git push | |
| git branch | |
| git fetch | |
| git merge | |
| git revert | |
| git checkout | |
| git remote | repos |
| git branch upstr | branch |

# About

## Repository status

Details on GitHub repository files, tags, commits follow:

```
Local:    main /Users/mark/Documents/project/misc-stats/get-going-with-git
Remote:   main @ origin (https://github.com/maj-biostat/get-going-with-git)
Head:     [c0e12c3] 2023-11-16: Fix url

Branches:         2
Tags:             1
Commits:         52
Contributors:     2
Stashes:          0
Ignored files:    6
Untracked files: 32
Unstaged files:   0
Staged files:     0

Latest commits:
[c0e12c3] 2023-11-16: Fix url
[cfdbc2d] 2023-11-16: Editorial
[ba0a5af] 2023-11-16: Editorial
[cfc6bfe] 2023-11-15: Start on gh pages
[024b82f] 2023-11-15: Start on CLI for gh
```

Todo:

tracked common commands doco link overleaf

prep notes