Get going with git

Mark Jones

2023-11-07

Table of contents

Introduction

When you are getting into the nitty gritty, version control is a large topic. The git, github, github cli and github pages … technologies can be daunting to set up and use. However, once you are setup properly and understand how things fit together at a high level, users can go a long way with a minimal set of commands.

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.

Get going with git - aims to provide a minimal knowledge base, sufficient to get you started without having to wade through reams of documentation distributed all over the internet. It can be browsed online or downloaded as a word document for your reference.

What is version control? Setting up git Time to commit: working with a local repository Push and pull: working with a remote repository Making your first repository on GitHub Day 2: Collaborating with Others

Dealing with (git) conflicts Branching for sanity Creating pull requests Collaborating with GitHub tools Day 3: Dealing with Complications

Undoing changes Learning good repository organization principles Ignoring things (in git) Working with large files Using the README Creating GitHub templates Extending git and GitHub with other tools

Pre-requisites

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?

Command line interfaces

A CLI is a software mechanism that you use to interact with your operating system via your keyboard rather than a mouse. You enter in commands as text and the system will do something, e.g. delete a file.

CLIs are software that are supplied with the operating system. Software that implements such a text interface is often called a command-line interpreter, command processor or a shell.

Nearly universally, if I use the word *terminal* I am referring to the operating system command line interface.

Windows refers to its CLI as the *command prompt*, and in macOS we have the *terminal*.

If you do not know how to operate your operating system CLI then you need to address that.

Windows

You can do these tutorials to familiarise yourselves:

- How to use Windows 10's Command Prompt
- Learning Windows Terminal

The following link needs admin rights (TKI people probably won't have nor will be able to obtain these rights) and is not strictly necessary here, but it gives a lot of useful commands and an opportunity to gain a bit more familiarity with the commandline.

- 40 Windows Commands you NEED to know

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

Operating system management - environment variables

You will need to have some minimal technical competence in driving your computer. At a minimum, 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 need to address that.

Other items that are important to be familiar with are file and directory concepts. For example, if you do not know what the command `tree` does, then it would be useful to find out. Similarly, if you do not know what file permissions are then, again, it would be useful to find out.

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`.

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. Go here and confirm that you can login.

Part I

Part 1 - Logistical elements

In this part we will get everything set up. This may well be the hardest part.

1 Resources

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

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

2 What is (this thing) called revision control

2.1 The big picture

First, let's briefly introduce some minimal terminology and context.

Note

TODO - Timelines

A **repository** is the mechanism that is used to implement version control by git.

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.

Git tools
FETCH
COMMIT
Local files

There are two types of repositories, *local* and *remote*.
The local repositories reside on your machine. 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/

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?

2.2 Why commandline

Because it is the best way.

3 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 gets less tiresome.

3.1 RStudio

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

3.2 Install git

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

3.2.1 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:

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

If you have homebrew (see 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.

3.2.2 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 doing 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.sh
```

`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>
```

3.2.3 Windows

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

Download the 64-bit standalone installer, run it, agree to the conditions and license, choose the default location.

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.
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
```

4 Git setup

4.1 Configuration for git

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

Figure 4.1: 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
```

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.

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

5 Github setup

5.1 GitHub account

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

While there are multiple ways to interact with GitHub from your local machine, here we will use the commandline. There are two protocols that can be used, HTTPS and SSH. We will use HTTPS.

First we need to set up a Personal access token.

5.1.1 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.

5.2 Git Credential manager

The GCM is a platform agnostic credential manager (in English, that translates loosely to a *password manager*). Once it's installed and configured, Git Credential Manager is called by git and you shouldn't need to do anything special.

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 and will be used every time you clone an HTTPS URL. Git will not require you to type your credentials in the command line again unless you change your credentials.

5.2.1 GCM install

For Windows users it can be installed by selecting this option during the installation wizard, see Section 3.2.3, step 6.

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/download/
## ==> Downloading from https://objects.githubusercontent.com/github-production-release-asse
## ##########################################################################################
## ==> Installing Cask git-credential-manager
## ==> Running installer for git-credential-manager with sudo; the password may be necessary
## Password:
## installer: Package name is Git Credential Manager
## installer: Installing at base path /
## installer: The install was successful.
##   git-credential-manager was successfully installed!
```

5.2.2 GCM demo

Below I demo the process by cloning a private repository from my GitHub account.

```
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:

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

selecting `Authorize git-ecosystem` will result in

at which point you use the 2-factor authenticator tool (I use google authenticator) to respond with an authentication code.

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

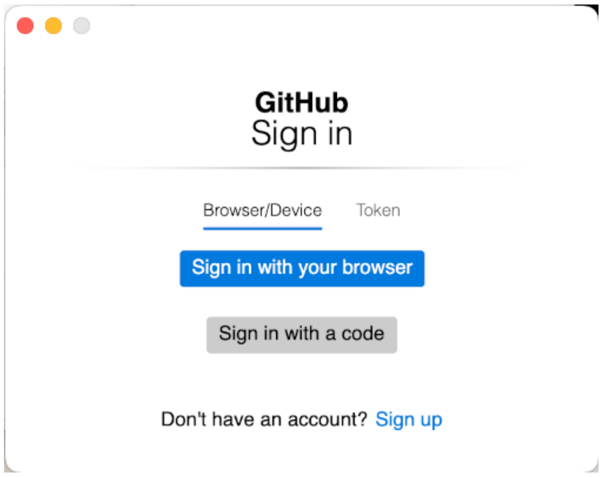
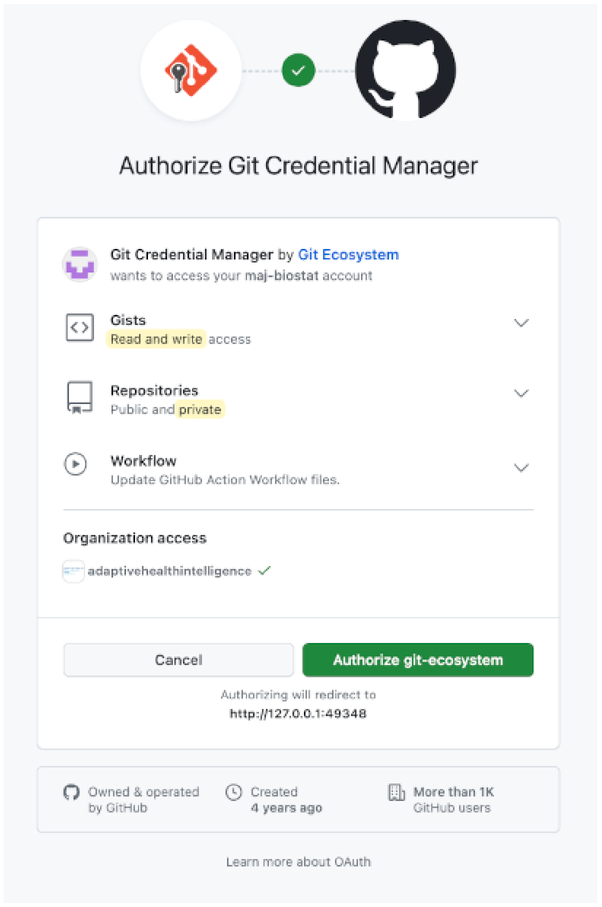


Figure 5.1: GCM



Git Credential Manager by Git Ecosystem
wants to access your maj-biostat account

Gists
Read and write access

Repositories
Public and private

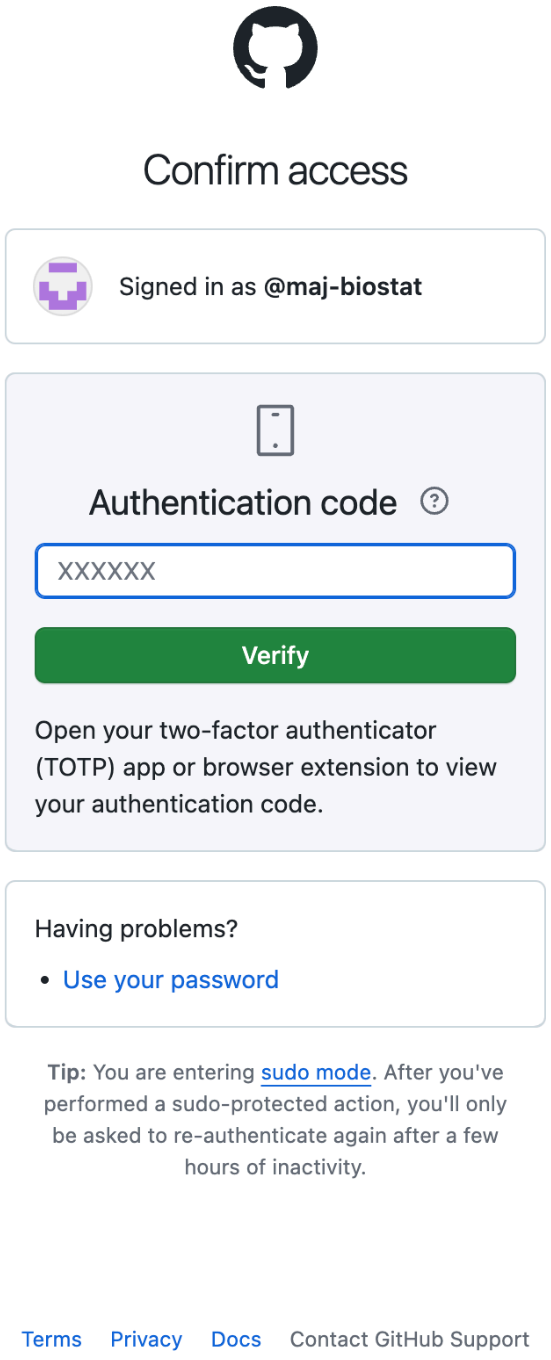
Workflow
Update GitHub Action Workflow files.

Organization access
adaptivehealthintelligence ✓

Authorizing will redirect to
http://127.0.0.1:49348

Owned & operated by GitHub
Created 4 years ago
More than 1K GitHub users

Figure 5.2: Sign in with browser



Confirm access

Signed in as **@maj-biostat**

Authentication code

XXXXXX

Verify

Open your two-factor authenticator (TOTP) app or browser extension to view your authentication code.

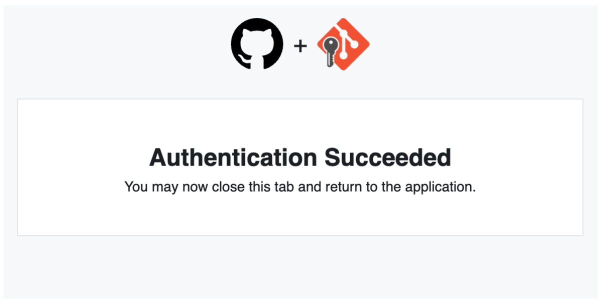
Having problems?

- Use your password

Tip: You are entering sudo mode. After you've performed a sudo-protected action, you'll only be asked to re-authenticate again after a few hours of inactivity.

Terms Privacy Docs Contact GitHub Support

Figure 5.3: Sign in with browser


Authentication Succeeded

You may now close this tab and return to the application.

Figure 5.4: Auth success

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

Finally, you will receive an email of this sort:

```
Hey maj-biostat!

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

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

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.

5.2.3 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
```

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 credential help is configured 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. 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.

5.3 GitHub CLI

In the day to day grind, having to deal with GitHub through its Web interface can be a little cumbersome. You can obviate having to 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 term 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/331c0b76fd34aa97342efa0
## ==> Fetching gh
## ==> Downloading https://ghcr.io/v2/homebrew/core/gh/blobs/sha256:a8c21e08d77963c2d12102ae
## Already downloaded: /Users/mark/Library/Caches/Homebrew/downloads/d0e6a3f8f7a4b138b36484e
## ==> 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`).
```

5.3.1 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 documen
## 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) vacc
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, for the moment, we won't be using github at all. 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.

Part III

Aside.

git remote add origin https://github.com/career-karma-tutorials/ck-git gh repo create my-newrepo –public –source=. –remote=upstream –push

The first part of the one liner: gh repo create my-newrepo creates and names a repo in your account (note: ‘my-newrepo’ should be replaced by the repo name of your choice) The -public flag makes sure the repo is public (swap this for -private if necessary) The -source=. flag specifies the source directory to be pushed Finally, the -remote=upstream flag specifies the remote repository to which the local repository is going to be compared with when pushing i.e the ‘upstream’ default.

6 Repositories

6.1 Git repositories

A repository is the most basic component of git. It is where you store your files and each files history. Through a variety of mechanisms, repositories can be public or private, can involve single people or multiple collaborators and can be stored locally (on your personal computer), in the cloud (in the cloud hosted by a service provider like github) or on a physical server (a basic file server will suffice in most cases).

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

6.1.1 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)
```

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. This directory contains the everything related to the repository. For examples, it contains all the version history and allows you to access files at any stage in their development. Generally, you will not touch this sub-directory directly.

> **Note**
>
> 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`.
> Additionally, 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`.

6.1.2 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 repository:

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

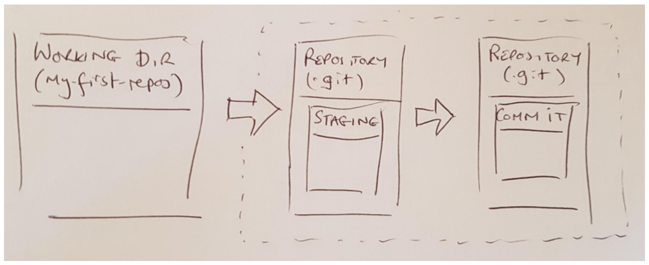


Figure 6.1: Repository structures

6.1.2.1 Working directory

Is the usual files and sub-directories within your project directory. You add, update, rename, delete files and direcotries in this area. 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*.

6.1.2.2 Staging area

Is a special space to where you list the files that are added to be committed as a new version under version control.

6.1.2.3 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. 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.

7 Stage and Commit

7.1 Adding files to projects

Let's start to introduce files for the project. Open a text editor, enter the following contents (or download the readme.md file from the applicable section under Chapter 1) 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 6.1.2 we now have the following:

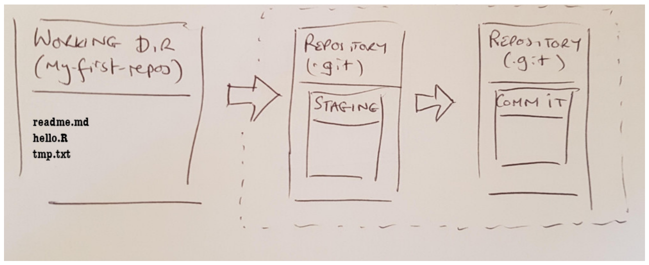


Figure 7.1: Add file to working directory

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)
```

7.2 Commit process

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

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

7.2.1 Staging

To add the files into the staging area run the commands:

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

Note that the `.gitignore` file was added as well as the `hello.R` and `readme.md` files. 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
```

Now the picture looks like this.

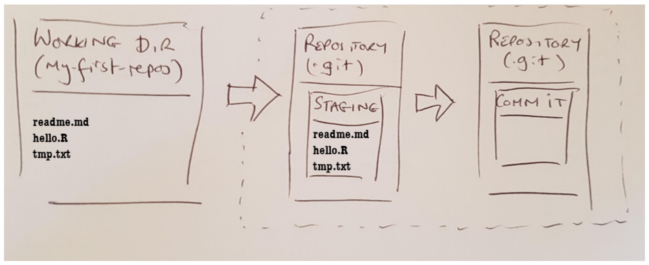


Figure 7.2: Add file to staging area

7.2.2 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. 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
```

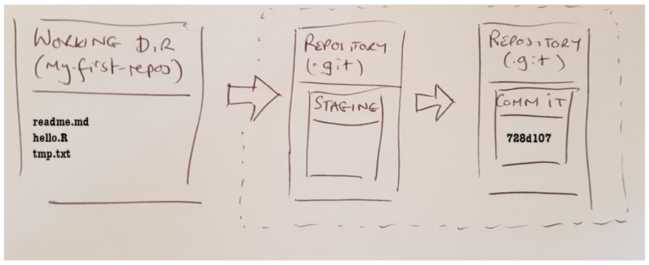


Figure 7.3: Commit files to history

7.3 Exercises

Exercise 7.1. 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 7.2. Add the new script to the staging area by following Section 7.2.1 ensuring that you review the status.

Exercise 7.3. Commit the staged files to the repository by following Section 7.2.2 making sure that you record a message for your commit.

Exercise 7.4. Edit the `readme.md` (use notepad or rstudio) file adding a new line with some arbitrary text. Stage the file and commit.

7.4 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.

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
```

8 Reviewing differences

8.1 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 what changes were made, we can run `git diff` but let's looks 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. For now, we will just deal with the commandline output.

Once satisfied that the changes are benign, stage and commit the edits in the usual way:

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

8.2 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`.

8.3 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.
## -
```

9 Branches

9.1 What is a branch?

⚠ Warning

We are going to step it up a notch. The topic of branching can be challenging for beginners. Practice and repitition is key to understanding. Try not to panic. At a basic level, it is actually straight forward once you get familiar with the processes.

When you run `git status` you saw the text - *On branch main* as part of the output. Similarly, when you ran `git log` the last 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.

Branching is simply a mechanism that allows you to diverge from the main line of development and continue to do work without messing with that main line. They allow multiple pieces of work to be progressed independently within the same project.

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.

In the examples encountered so far, the branching we have encountered is just a stem (specifically, the *main* branch). Here 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.

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?”

9.2 Time travel

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

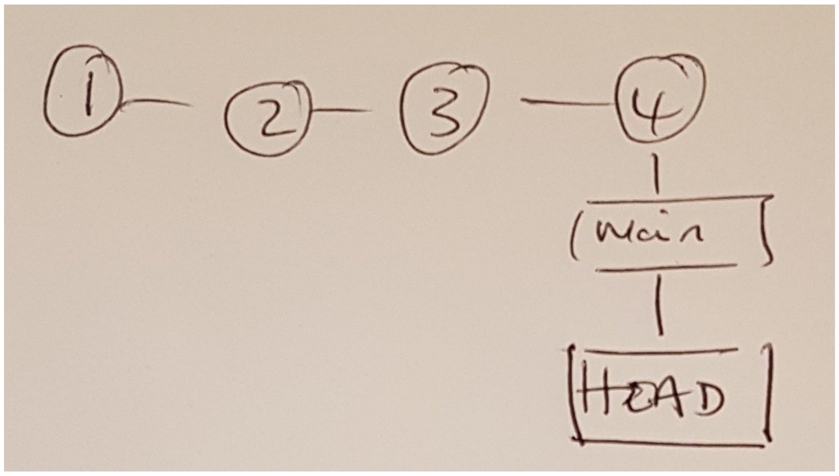


Figure 9.1: Branch more like a stem

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

The file contains the following:

```r
# 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.

Next we go over the processes involved.

9.2.1 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"
```

9.2.2 Rewind

Now we want to rewind our repository back to the time at which the deliverable was made. We can do this by 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.

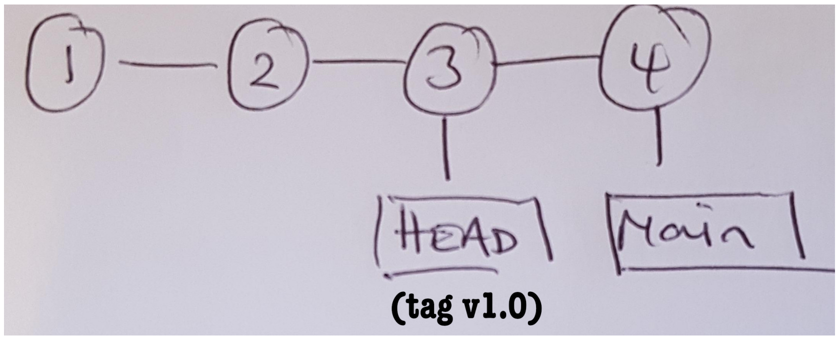


Figure 9.2: Rewind to earlier deliverable

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

Note

Don't panic overly about the warning about being in the *detached HEAD* state.

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

9.2.3 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:

9.2.4 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).

```
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*).

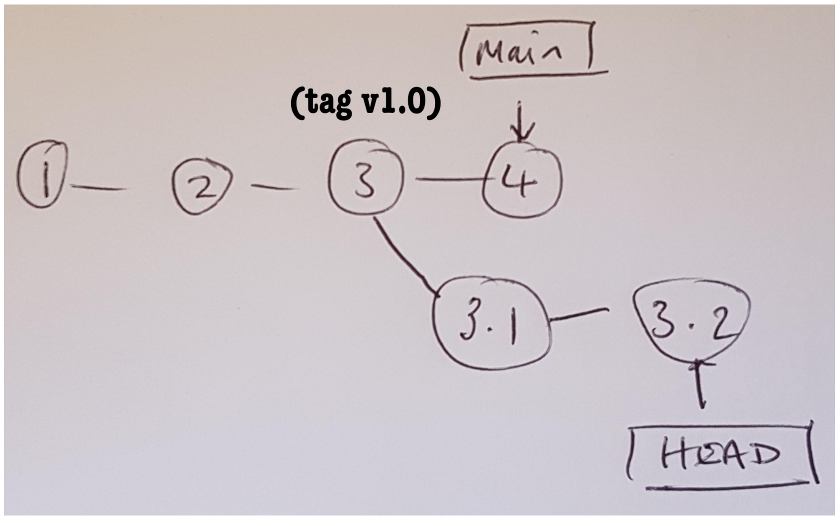


Figure 9.3: Add a fix

```
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

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)"
```

9.2.5 Switching back to the secondary

Switching back to our partially complete secondary analysis is as simple as.

```
git checkout main
```

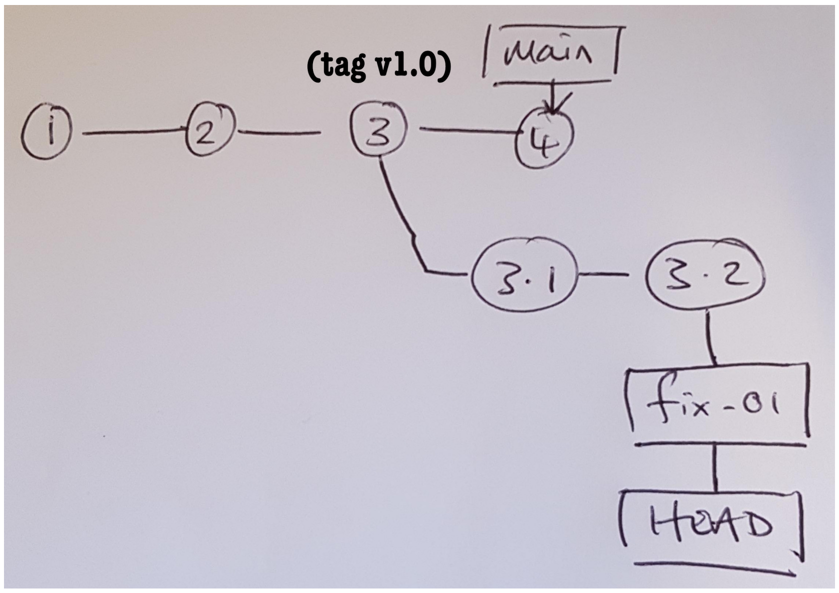


Figure 9.4: Looking at the HEAD of our new branch

If you look at the `branching.R` script you will be able to see the secondary analysis we started some time ago. However, 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 later.

10 Merge

10.1 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.

10.2 Merge processes

Merging is usually a fairly automated process. Run the following.

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

You may be prompted to enter a commit message (alternatively just provide the -m flag).

```
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.
```

and then 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.

10.3 Exercises

Exercise 10.1. 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 10.2. 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. 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 main:

```
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.

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 the fundamentals. However, 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 IV

Part 3 - Collaboration

Remotes

Hosting git

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:     [6e58a92] 2023-11-07: First pass at merge

Branches:        2
Tags:            1
Commits:        34
Contributors:    1
Stashes:         0
Ignored files:   4
Untracked files: 25
Unstaged files:  0
Staged files:    0

Latest commits:
[6e58a92] 2023-11-07: First pass at merge
[1961517] 2023-11-07: Another restruct. Incremental dev
[e9c18e7] 2023-11-07: minor edit
[e9f1bd0] 2023-11-07: Forgot to save
[62a5b81] 2023-11-07: Consolidated edits and restructure
```