

R4DS - Unit 3: Git & Github or: How I Learned to Stop Worrying and Love Version Control



Andrea Gilardi
March 27, 2023



Outline and main concepts

- The objective of this unit is to introduce a Version Control System (VCS) named Git and its basic functionalities.
- After installing the software, we will explore a simple but powerful workflow considering a local repository. Then, we will connect our PC to a remote repository on Github.
- Finally, I will show you how to replicate (most of) the same steps using the Rstudio IDE.



About Version Control

- Version control is a system that records changes to a set of files over time so that you can recall specific versions later.
- For example, if you are developing an algorithm for a new statistical procedure, Version Control Systems (VCS) allow you to revert files back to a previous state in case of bugs.
- You can also compare files over times, see which changes introduced the bug and revert back to a working state.
- **If you screw up, you can “easily” recover!.**



What is Git?

- Git¹ is a Distributed VCS, which means that you always have the entire history of the project at your disposal.
- It is a particularly efficient tool for large projects and it has a branching system that allows for non-linear development.
- At the beginning, we will work with the Git Shell. Then, I will show you how to integrate Git and Rstudio.

¹Why is it named like that? See the last paragraph [here!](#)



Install git

First, you need to check if you've already installed git! Run the following from your shell: `git --version`.

If you see `git command not found`, keep reading!

Windows: Download installer from [here](#). A few notes:

- You should install git under `C:/Program Files`.
- When asked about Adjusting your PATH environment, make sure to select Git from the command line and also from 3rd-party software.

Let's do it together!

macOS: The shell should prompt you to install it after running the previous command. See also [here](#).



First time git setup

First, we can slightly customize the Git environment.

Setup identity: Run the following commands

- `git config --global user.name "Andrea Gilardi"`
- `git config --global user.email andrea.gilardi@unimib.it`

Please use the same email as the github account.

Setup editor: See [here](#) for instructions.

Default branch name: You can run

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

The default name for the initial branch is master.

Check settings: Run `git config --list`



Git Basics 😊...

In case of fire



1. `git commit`



2. `git push`



3. leave building

Source. Don't worry, now we are going to review many many more details!



My first Git repo!

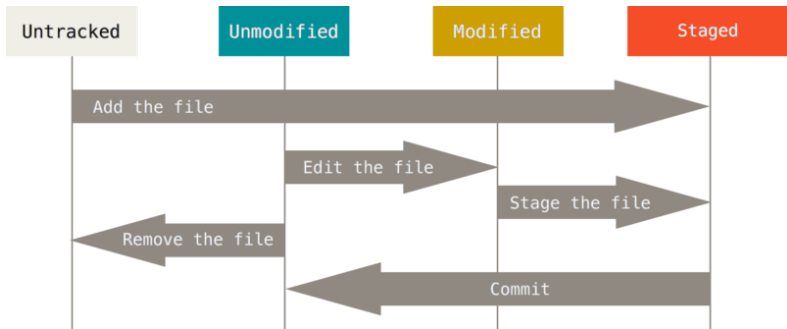
We can create a Git repository in two ways:

1. Take a local directory and initialise a Git structure.
 - Create an empty directory where you will store your files;
 - Open the shell inside the new repo and run `git init`.
2. Clone an existing repository stored online.
 - Open the shell into the parent repository;
 - Run `git clone <url>`. For example, run `git clone https://github.com/agila5/R4DS-PhD-Unimib.git`.

At the end of the class, we will see how to create a Git repo on Github and clone it on your local computer.

Git basics - A graphical abstract

The following figure nicely summarises the algorithm used by Git to record changes in a repository.



Source: <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>



Git Status

- Let's create a local Git repo (e.g. myFirstGitRepo)!
- Open a (Git) shell inside the repo and run `git status`.
You should see the following output:

```
On branch main  
  
No commits yet  
  
nothing to commit (create/copy files and use "git add" to  
track)
```

- Now we need to create a new file (say README.md) and add some content (e.g. "Hello World :)").



Git Status (cont)

- Now you should see

```
On branch main

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)
  README.md

nothing added to commit but untracked files present (use "git
add" to track)
```

- Untracked basically means that Git sees a file you didn't have in the previous snapshot, and which hasn't yet been staged. Let's start tracking the file!



Tracking New Files

- We can run `git add README.md` to start tracking the new file. Then, if we run `git status` again we see

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

- Now `README.md` file is staged² and we can commit³ by running `git commit -m "Some text"`.

²Staged means “ready to go into the next commit snapshot”.

³Committed means “saved in the local database”.



Committing Your Changes

- You should see something like

```
$ git commit -m "first commit - add README.md"
[main (root-commit) 3b94fbf] first commit - add README.md
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

We can see that the message describes some details about the commit (the branch, how many files were changed ...)

- If we check the status again...

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



Let's do it again!

- Let's create a new file (say CONTRIBUTING.md) and add some text to it. Now, if we ran `git status`, we would see the same output as before (just with a different file name).
- Let's modify also README.md adding some text. Now we see

```
[truncated...]  
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)  
    CONTRIBUTING.md  
  
[truncated...]
```



Let's do it again! (cont)

- The output says that the **tracked** file README.md has been modified in the directory but is not **staged** yet. We can stage both files running `git add .` and now we see

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   CONTRIBUTING.md
    modified:   README.md
```

- Now we could run `git commit -m "..."` as before. But what happens if we modified a staged file? Let's add even more text to README.md.



Let's do it again! (cont)

- We see the following

```
$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
    new file:   CONTRIBUTING.md
    modified:   README.md

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
```

- If you commit now, you will preserve the currently staged version of README.md. You can run `git add README.md` again in case you need to update the file version.



Let's do it again! (cont)

- The `git diff` command can be used to see what we changed but not yet staged:

```
$ git diff
diff --git a/README.md b/README.md
index dd577f0..3d6d21b 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
Hallo World :)
-Hallo World 2 :)
\ No newline at end of file
+Hallo World 2 :)
+Hallo World 3 :)
\ No newline at end of file
```

- And if we want to compare the currently staged file with the previous commit? Then, use `git diff --staged`.

Let's do it again! (cont)

- Now you should see

```
$ git diff --staged
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
new file mode 100644
index 0000000..b3da348
--- /dev/null
+++ b/CONTRIBUTING.md
@@ -0,0 +1 @@
+You are welcome to contribute to my repo :)
\ No newline at end of file
diff --git a/README.md b/README.md
index bf231b1..dd577f0 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
-Hallo World :)
+Hallo World :)
+Hallo World 2 :)
\ No newline at end of file
```



Let's do it again! (cont)

- Now, after staging all files and committing, we should see

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

- **NB:** If you add the `-a` flag to `git commit`, then Git automatically stage every file **that is already tracked** before doing the commit.



Ignoring files

- A .gitignore file can be used to tell Git not to track certain types of files (e.g. log files).
- Let's create a example.log file inside our repo. We see

```
$ git status
On branch main
Untracked files:
(use "git add <file>..." to include in what will be committed)
    example.log

nothing added to commit but untracked files present (use "git
add" to track)
```



Ignoring files (cont)

- If we create a `.gitignore` file that includes the path of the new file (i.e. `example.log`), then we see

```
$ git status
On branch main
Untracked files:
(use "git add <file>..." to include in what will be committed)
    .gitignore

nothing added to commit but untracked files present (use "git
add" to track)
```

- More details: <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>



I f*ucked up 😭 --amend!

- The --amend option can be used to rectify a git commit message. For example, if we run

```
$ git add .  
$ git commit -m "a typo here :("
```

then we can adjust the text with

```
$ git commit --amend -m "the right text now!"
```

- Check the commit history with `git log`!
- The --amend option works analogously if we add more files to the staged area.



I f*ucked up 🤦 git restore!

- As we saw from the previous messages, `git restore --staged <file>` can be used to unstage a file. We can test it by modifying the `.gitignore` file explicitly forcing Git to ignore `.Rdata` files (see Unit 1!).
- Similarly, `git restore <file>` can be used to discard unstaged changes so that a file looks like when we last committed it! Let's remove the `.Rdata` string from `.gitignore`.



Other commands

As you can imagine, there are several more useful Git commands but we are not going to review all of them. A (not-comprehensive) list:

- `git rm`: remove a file from Git (see also previous messages);
- `git mv`: record a file movement;
- `git log`: explore the commits' history.

During the next units, we will explore `git blame` and `git bisect` (which are useful for debugging Git projects).



Github!

- **Github** is an hosting service that provides home for Git-based projects. Github allows other people to see your work, but it is a also good idea to save your (solo) projects on Github (as a backup in case you screw up badly).
- The first thing to do is to create an account. [Signup](#).
- See [here](#) if you need a few suggestions for choosing your github name. See also [this](#) Chapter (and the whole book) for a more complete description of Git/Github.
- Now we need to complete the most difficult step: synch (local) Git with (remote) Github...



Can you hear me now?

- When you interact with a remote Git server, you need to include your credentials into the request (otherwise everybody can add files to your repo, which is bad...).
- Git can adopt two different protocols: HTTPS and SSH. Here we focus on HTTPS. Check the references of this course if you want to explore the other protocol.
- An HTTPS protocol uses a Personal Access Token (PAT) to add private credentials into your request.



Can you hear me now? (cont)

- For us, the easiest way to generate a PAT is running the following R commands:

```
install.packages(c("usethis", "gitcreds"))  
usethis::create_github_token()
```

- Then, store the PAT explicitly by running

```
gitcreds::gitcreds_set()
```

to get a prompt where you can paste your PAT.

- See [here](#) for many more details and troubleshooting.
- Now we can finally [create](#) our first Github repo and clone it.



Git and Github

Now let's recap the whole process:

1. Create an (empty) repository on Github;
2. Clone the repository on your computer (i.e. `git clone`);
3. Add a `README.md` file with some content;
4. Stage the new file and create a new commit;
5. Push the new commit on Github (i.e. `git push`);
6. Check the result and celebrate 🥳



Git, Github and Rstudio

Now, let's delete the Git repo (after pushing the new commit) and repeat the same process from Rstudio.

1. Open Rstudio and Click on "Create a Project";
2. Select Version Control -> Git and copy the URL from github. Then click on Create Project.

The output is a Git folder with the same file as before but it includes also an Rstudio project. Finally,

1. explore the new (automatically-defined) .gitignore file;
2. create a new .R file and commit with the Git control panel.



More details on remotes

- As you can imagine, there is nothing special with Github and we can use a similar process to connect a local Git repo with other hosting service (e.g. Gitlab) and other remotes.
- The command `git remote` can list the remotes:

```
$ git remote -v
origin  https://github.com/agila5/R4DS-PhD-Unimib.git (fetch)
origin  https://github.com/agila5/R4DS-PhD-Unimib.git (push)
```

- See <https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes> for more details. We will also re-examine this topic when we talk about branches.



Homework!

1. Create an (empty) repository on Github named GithubHomework;
2. Clone it on your computer using the Rstudio IDE;
3. Create a README.md file that contains a minimal description of the repository. Push the new file on Github including an informative commit message and check the output locally and online. **NB:** Run `git status` frequently.
4. Add a new directory named R and, into the new directory, create two new files named GithubHomework.log and script.R, respectively.
5. Ignore the .log file and commit the .R file using the following message: "BLABLABLA". Double check it to be 100% sure.



Homework! (cont)

6. Amend the previous commit message adopting a more informative one.
7. Push the new changes to the remote (i.e. Github).
8. How can you modify the message for a commit that you've already pushed? **Google is your friend** 😊.
9. Check that you completed the previous task by exploring the history of the project both from Git (i.e. `git log`) and from Github (again, ask to Google!).