gesis

Leibniz Institute
for the Social Sciences

Tools and Workflows for Reproducible Research in the Quantitative Social Sciences

An introduction to Git

Bernd Weiß

2022-11-17

Leibniz
Association

(Source: xkcd, https://xkcd.com/1597/, accessed on 2017-12-23)

Overview

Overview

- In part I, I will be introducing Git -- and I will not talk very much about collaboration

- Part II will focus on collaboration, GitHub, and more exercises

The sample R file

So, let's print the content of the test file:

```
## l1: # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

The concept of version control

Preliminaries

- Git and other tools have been developed in the context of software development (in the Linux community, to be more precise)

- Even though there exits graphical user interfaces (GUIs) for working with Git, it is highly recommended that you have a basic knowledge of how to use the CLI

- Once you mastered using Git at the command line, using a GUI is a peace of cake

Since most of you are working on a MS Windows PC, it is also a good idea to know the meaning of the `PATH` variable (what it is used for, how to access it and how to modify it).

Why use a Version Control System?

- Version Control System = VCS

- For backup

- For collaborative work and syncing

- There is always a (the) "most recent" version of a file

- Given that there are conflicting versions of (text) files, Git is able to clearly identify these conflicts by displaying the differences of conflicting files (given they are in plain text)

- Keeping track of changes (aka, time travel; all changes are tracked and it is quite easy to go back in time).

  So, even if you invented Skynet (popcultural reference) and mankind is about to being terminated for good, you always can go back in time

- And, avoiding the horror of `final_rev2_update12_after-computer-crashed.docx` (see http://phdcomics.com/comics.php?f=1531)

Name	Änderungsdatum	Typ	Größe
C055_Weiss_et_al_revision_13.doc	21.11.2017 13:55	Microsoft Word 9...	109 KB
C055_Weiss_et_al_autoreninfos.docx	21.11.2017 11:43	Microsoft Word-D...	16 KB
C055_Weiss_et_al_revision_12.doc	21.11.2017 11:12	Microsoft Word 9...	108 KB
C055_Weiss_et_al_revision_11_bs-hs_GBD.doc	20.11.2017 15:11	Microsoft Word 9...	313 KB
C055_Weiss et al_revision_11.doc	15.11.2017 09:51	Microsoft Word 9...	307 KB
C055_Weiss et al_revision_10.docx	15.11.2017 07:50	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_7.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_8.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_9.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_6.docx	13.11.2017 07:05	Microsoft Word-D...	271 KB
20171007_mobile-befragungen.docx	06.11.2017 17:44	Microsoft Word-D...	321 KB
C055_Weiss et al_revision_5.docx	06.11.2017 17:44	Microsoft Word-D...	321 KB
20171011_mobile-online-befragungen.pdf	11.10.2017 12:55	PDF-Datei	463 KB
20171011_mobile-online-befragungen.docx	11.10.2017 12:52	Microsoft Word-D...	351 KB
Bernd_Weiss_20171011-03_mobile-befragungen.docx	11.10.2017 12:51	Microsoft Word-D...	351 KB

- For having the possibility to test new code/features in a "sandbox" (aka a new "branch")

  (following up my Skynet reference: create a parallel universe (the branch), do your evil thing and then go back to your reality if you don't like it, i.e., delete the parallel universe/branch)

- While creating a history of changes, you are supposed to provide proper and meaningful messages that describe what has changed. If you do this thoroughly, you have a nice log file of all changes (like a lab notebook)

- Authorship attribution

- Modern web interfaces such as GitHub also allow for social interaction (strangers can send you pull requests to improve your code/document/...)

For what type of files is a VCS useful?

- Most useful for text files (Stata do files, SPSS syntax files, R skripts etc.). Text files can stored very efficient since only changes between version are tracked

- Binary files (Blob = binary large object) (images, word files, Stata data files, etc.) can be stored in a VCS but less efficient than text files since every time the entire file is saved

Terminology and concepts

- There is Git.

- "Git" is the name of the software, and the actual command-line tool is `git` (e.g., in Windows it is `git.exe`).

- And, then there are GitHub, GitLab etc., which are (web-based GUI) front ends to make working with Git easier, especially when it comes to collaborative work.

(Source: https://github.com/jobreu/reproducible-research-gesis-2022)
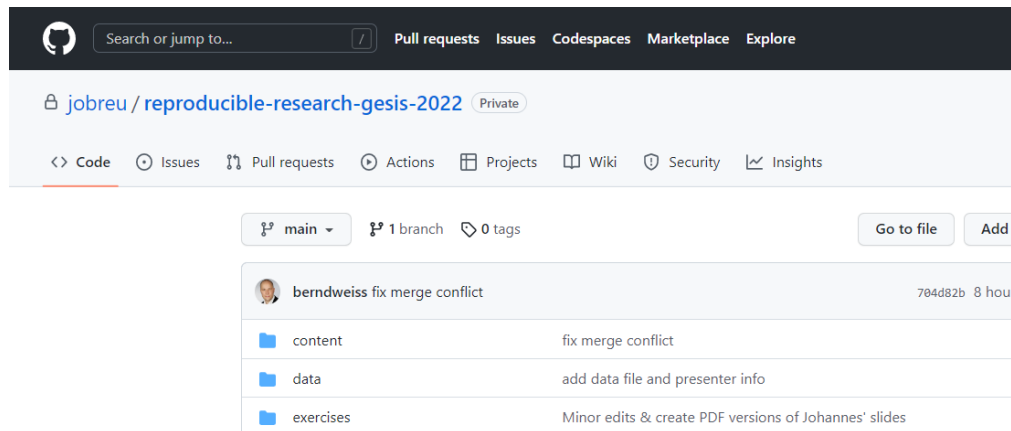
- GitHub and GitLab provide the opportunity to setup a remote Git repository.
- So, in most cases you will need a local installation of Git.
- In addition to being a frontend to Git, GitHub and GitLab also provide project management features and allow to create so-called

Git: A 30,000 foot view

- Git is a version control system (VCS). As mentioned above, a VCS allows you to track the history and attribution of your project files over time in a repository (Narębski, 2016)

- It is, if you will, a (very, very) powerful undo function (well, kind of...)

- To be more precise, Git is a distributed VCS (DCVS) and hence a tool for collaborartive work

- If you want to utilize Git for collaborative work, one approach of using Git in this context assumes that there exists a central and remote repository. Most famous is GitHub, at GESIS we use GitLab

- Workflow in Git (given that a Git repository has already be initialized):

Source: Healy, 2019; see https://plain-text.co/keep-a-record.html
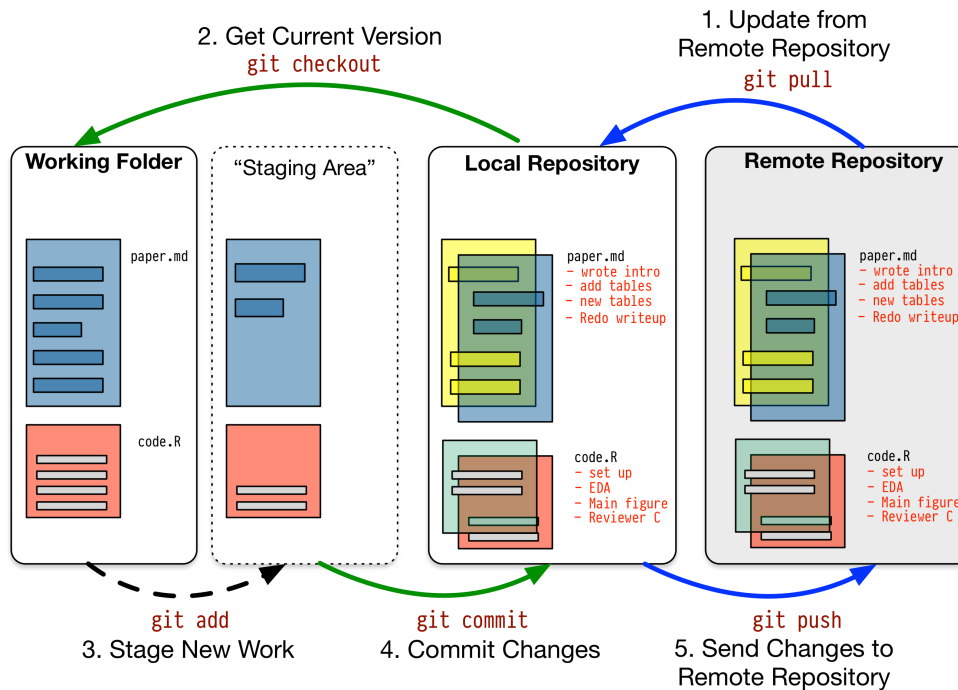
- Work locally (i.e., on your computer) on your files until a certain feature is completed (a function is completed, a paragraph written etc).

- `Commit` your file and write a commit message, i.e., inform git that a certain file (or more) have changed and inform your future self (or someone else) about the nature of your changes (aka write a commit message). This has to be done manually.

- Commit early, commit often!

- When a remote repository exists: send (`push`) your changes to the remote repository.

How I use Git

- Git is a very powerful tool, in my own work, I utilize a rather limited set of its capabilities

- This is what I mostly do with Git:

  - Initialize a new Git repository or clone an existing respository
  - Backup my work on a remote server
  - Track changes
  - Use branches to implement experimental features
  - Search (and undo) previous changes (most of the time using the interface provided bv GitHub or GitLab)

- "Google" (or whatever your prefered seach engine is) a lot ...

Installing Git and setup

Download and installation

Git (for Windows) can be downloaded from: https://git-scm.com/download/win.

Here are a few questions that you will be asked during the installation:

- Default editor (use Notepad++ if you have it on your computed, vim also works)

- Adjusting your PATH environment (you might want to go with the second optionn "Use Git from the Windows command Prompt")

- Choosing HTTPS... (go with default: OpenSSL)

In case you will be working with others, you also will need a remote repository (be able to acces a remote respository). For convenience reasons it is recommended that you also install/set up SSH (see next slides).

For various reasons, I no longer use a standalone version of Git but use a version of Git that can be installed via MSYS2.

> "MSYS2 is a collection of tools and libraries providing you with an easy-to-use environment for building, installing and running native Windows software." -- https://www.msys2.org/.

Well, not so distributed at all...

- Even though Git is called "distributed", most of the time, there is just one central server (e.,g., GitHub, GitLab, ...)

- A Git project is stored in a repository, which can be local or remote

- When using Git to access a remote repository (for backup or collaborative work) on a remote server, you need to authenticate yourself to the server

- There are two ways of authentication: HTTPS or SSH

- Despite its technical details, I always choose SSH, but Johannes, for instance, prefers HTTPS (in part II I will introduce the HTTPS approach)

- More information can be found on these websites:

    - https://happygitwithr.com/index.html
    - https://happygitwithr.com/https-pat.html
    - https://happygitwithr.com/ssh-keys.html
    - https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories
    - https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/about-authentication-to-github#authenticating-with-the-command-line

Basic workflow

Setting up a Git repository

Usually, there are two ways to set up/obtain a Git repository:

1. You create a new Git repository

   (the next step might be: push it to GitHub/GitLab and start collaborating with your colleagues or only yourself)

2. You "clone" an existing repository from a remote Git server such as GitHub/GitLab

Creating a local Git repository

The first step is to create a Git repository. After the repository has been created, we need to tell `git` which files will be subject to version control. So, the following git commands will be utilized:

- `git init`: Creates a new folder `.git`, which contains configuration files and the repository. As of now, `git` does not know anything about our file(s), e.g., `test.R`

- From now on, all examples will refer to a demo repository called `git_test_folder`

The content of the `git_test_folder` is

```
## total 13
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## -rw-r--r-- 1 weissbd GESIS+Group(513) 131 Nov 16 16:48 test.R
```

And, `test.R` contains the following content:

```
## l1: # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

Now, let's initialize the Git repository using the `git init` command (please ignore the `cd /e/tmp/git_test_folder` part).

```
cd /e/tmp/git_test_folder
git init
```

```
## Initialized empty Git repository in E:/tmp/git_test_folder/.git/
```

```
## total 17
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .git
## -rw-r--r-- 1 weissbd GESIS+Group(513) 131 Nov 16 16:48 test.R
```

git status

Let's check the status of our newly created repository using the command `git status`. It shows the status of the current working tree (and branch). As of now, `git` is not aware of any files yet, so it informs us about the existence of 'Untracked files: ...'.

```
cd /e/tmp/git_test_folder
git status
```

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

(Note: the main branch is called `main`; it used to be called "master", the new convention, though, is "main")

Adding files to a git repository: `git add` **and** `git commit`

Now it is time for some file action by adding (a) file(s) to our repository.

In the previous section on `git status` it was recommended that `git add` is used to add files to the git repository:

```
(use "git add <file>..." to include in what will be committed)
```

That is what we are going to do now: To actually 'save' (check-in or track) files in the repository, a *two-step* procedure needs to be performed.

The first step is to call `git add`, the second step is to commit the file(s) using `git commit`. For now, it might be hard to see the benefit of this two-step procedure, see http://gitolite.com/uses-of-index.html for a thorough description (I like the "staging helps you split up one large change into multiple commits" argument).

- `git add -A`: Adds (here ´-A´ means "all files") files to the *index* (or staging area) -- note: this approach is against the principle of "focus on one aspect of the code changes".

- To add a particular files to the index, use `git add my_special_file.do`.

```
cd /e/tmp/git_test_folder
git add -A
```

Again, let's see what `git status` has to say...

```
cd /e/tmp/git_test_folder
git status
```

```
## On branch main
##
## No commits yet
##
## Changes to be committed:
##   (use "git rm --cached <file>..." to unstage)
##     new file:   test.R
```

The second step is to run the command

`git commit -m "your text, verbs in imperative form"` (see below), e.g.

`git commit -m "add function to compute tau^2".`

Since this is my first commit, I always apply the following commit message: `git commit -m "initial commit.`

```
cd /e/tmp/git_test_folder
git commit -m "Initial commit"
```

```
## [main (root-commit) 8a6136f] Initial commit
##  1 file changed, 7 insertions(+)
##  create mode 100644 test.R
```

According to the Git developer site commit messages should follow the "imperative-style":

> "Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behavior. Try to make sure your explanation can be understood without external resources. Instead of giving a URL to a mailing list archive, summarize the relevant points of the discussion."

Again, let's see what `git status` reports:

```
cd /e/tmp/git_test_folder
git status
```

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

So, there are no untracked files, that is, "nothing to commit, working directory clean".

Git's commit history: `git log`

There is another useful command `git log` that informs about `git`'s history, i.e. commited files and folders:

```
cd /e/tmp/git_test_folder
git log
```

```
## commit 8a6136fd6b0e790124064b91a440a963f2d0b209
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:16 2022 +0100
##
##     Initial commit
```

Right now, the history only contains one entry.

The very first line `commit ...` shows the SHA1 hash. The 'Secure Hash Algorithm 1' is used to calculate this long, hexadecimal number for a file. Files with identical content are represented by an identical SHA1 hash, files with different content do not share an identical SHA1 hash. Using these SHA1 numbers, `git` can identify changes in a file.

Exercise

- Open your Git Bash
- Go to your Home directory via `cd ~` (or, actually, go wherever you want)
- Create a new folder (either via `mkdir` or `create-project.sh`)
- Change into the newly created directory via `<your input here>`
- Initialize your new Git project via `git init`
- Copy a few files (PDF files etc. -- does not really matter, but no sensitive material!) in your new project folder
- What comes next? Hint: `git add` and then `git commit <your input>`
- Check the status and the history of your Git repository

Moving back in time

Undo changes / Going back in Git's history

- Undoing changes can be done utilizing three different approaches (`git checkout`, `git revert`, `git reset`)

- Depends on the state of your working directory (clean or uncommitted changes)

- A pragmatic approach is to utilize the search functionality of a web platform such as GitHub or GitLab

- Here, only some basics will be introduced, further information is provided by https://www.atlassian.com/git/tutorials/undoing-changes or https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

git checkout

- In order to undo (a) *uncommitted* changes or (b) going back to an earlier commit, respectively, the command `git checkout` can be utilized

- You have multiple possibilities to undo changes. You can undo changes regarding a particular file or you can go back to an earlier commit, which may contain multiple changes (not a good practice, though)

- `git checkout -- myfile` will discard all changes with respect to `myfile`

- `git checkout -- .` (or use `git restore .`) will discard all changes in your working directory, which can include multiple files (remember the dot `.` from my Computer Literacy slides)

- For more information see https://www.atlassian.com/git/tutorials/using-branches/git-checkout

Modify content

Let's start changing the content of test.R. For instance, remove line 3 (`# Always start with a dumb comment`). First, let's print the original file content again:

```
cd /e/tmp/git_test_folder
cat test.R
```

```
## l1: # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

```
remove_line("e:/tmp/git_test_folder/test.R", 3)
```

Print out the new code file (remember, the comment line (3. line) has been removed).

```
## l1: # Branch: main
## l2: # Author: BW
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

Again, let's check `git status` to see what our repository is doing and what has changed... the important part is `modified: test.R`

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

Run `git checkout`...

```
cd /e/tmp/git_test_folder
git checkout -- test.R
```

Voilà, our beloved comment (line 3) has been risen from the dead...

```
## l1: # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

Introducing more changes to `test.R`

Okay, let's again modify test.R. Now, we do this two times. I will use M1 and M2 to denote these two changes (I will also `add` and `commit` these changes).

Again, print new content of `test.R`.

```
## l1: # M1: New line added
## # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: M2: A new comment
## l6: var(x)
## l7: sum(x)
```

And, let's see the history via `git log`:

```
cd /e/tmp/git_test_folder
git log
```

```
## commit c5f7b2529b6f8a5c65a1795083add938892c6224
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:20 2022 +0100
##
##     add new comment (M2)
##
## commit 231f0ab4a02dbf9c8f97f0df860980f6caf3d132
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:19 2022 +0100
##
##     add new line (M1)
##
## commit 8a6136fd6b0e790124064b91a440a963f2d0b209
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:16 2022 +0100
##
##     Initial commit
```

Now, we would like to discard any changes introduced by M2 by using `git revert.`

git revert

Put simply: `git revert` can undo a certain commit and adds a new history to the project.

For more information see
https://www.atlassian.com/git/tutorials/undoing-changes/git-revert

```
cd /e/tmp/git_test_folder
git revert --no-edit HEAD
```

```
## [main 7678c7c] Revert "add new comment (M2)"
##  Date: Wed Nov 16 16:48:21 2022 +0100
##  1 file changed, 1 insertion(+), 1 deletion(-)
```

See https://nulab.com/learn/software-development/git-tutorial/git-collaboration/ for the specification of a commit relative to the most recent commit (HEAD)

```
cd /e/tmp/git_test_folder
git log
```

```
## commit 7678c7ce3d9132009e6de433dc10683d2e38a0c6
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:21 2022 +0100
##
##      Revert "add new comment (M2)"
##
##      This reverts commit c5f7b2529b6f8a5c65a1795083add938892c6224.
##
## commit c5f7b2529b6f8a5c65a1795083add938892c6224
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:20 2022 +0100
##
##      add new comment (M2)
##
## commit 231f0ab4a02dbf9c8f97f0df860980f6caf3d132
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:19 2022 +0100
##
##      add new line (M1)
##
## commit 8a6136fd6b0e790124064b91a440a963f2d0b209
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:16 2022 +0100
##
##      Initial commit
```

And back to M1...

```
cd /e/tmp/git_test_folder
cat test.R
```

```
## l1: # M1: New line added
## # Branch: main
## l2: # Author: BW
## l3: # Always start with a dumb comment
## l4: x <- c(1:10)
## l5: mean(x)
## l6: var(x)
## l7: sum(x)
```

git reset

Put simply: `git reset` goes back to a certain commit and discards all later commits

Be very careful with `git reset` and do not use it when working with others!

For more information see
https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

Excursus: The magic of DAGs

I am just including the link to an example of a Git DAG:

https://subscription.packtpub.com/book/application-development/9781782168454/1/ch01lvl1sec11/viewing-the-dag

Studying △s

What has changed at the file level?
`git show` and `git diff`

In this chapter we will learn about `git show` and `git diff`, which show differences at the file level. However, for those of you who do not feel comfortable using the command line I highly recommend meld (http://meldmerge.org/).

So far, we have only a few commits. `git log` shows all commits, the SHA1 hash and the respective commit message.

```
cd /e/tmp/git_test_folder
git log
```

```
## commit 7678c7ce3d9132009e6de433dc10683d2e38a0c6
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:21 2022 +0100
##
##     Revert "add new comment (M2)"
##
##     This reverts commit c5f7b2529b6f8a5c65a1795083add938892c6224.
##
## commit c5f7b2529b6f8a5c65a1795083add938892c6224
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:20 2022 +0100
##
##     add new comment (M2)
##
## commit 231f0ab4a02dbf9c8f97f0df860980f6caf3d132
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:19 2022 +0100
##
##     add new line (M1)
##
## commit 8a6136fd6b0e790124064b91a440a963f2d0b209
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:16 2022 +0100
##
##     Initial commit
```

A brief intro to the unified diff format

Using `git show` without any additional arguments shows the differences between the last commit and HEAD. The output follows the so called "unified diff format" (UDF). A good introduction of UDF ist provided by https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html#Detailed-Unified. The following is mostly copy-and-paste from the aforementioned source. It is also imported to note that UDF utilizes so-called (c)hunks to describe changes. A hunk is a paragraph separated by an empty line.

git show

```
cd /e/tmp/git_test_folder
git show
```

```
## commit 7678c7ce3d9132009e6de433dc10683d2e38a0c6
## Author: Bernd Weiss <spam@metaanalyse.de>
## Date:   Wed Nov 16 16:48:21 2022 +0100
##
##     Revert "add new comment (M2)"
##
##     This reverts commit c5f7b2529b6f8a5c65a1795083add938892c6224.
##
## diff --git a/test.R b/test.R
## index 5baac9c..e8eb805 100644
## --- a/test.R
## +++ b/test.R
## @@ -3,6 +3,6 @@ l1: # M1: New line added
##  l2: # Author: BW
##  l3: # Always start with a dumb comment
##  l4: x <- c(1:10)
## -l5: M2: A new comment
## +l5: mean(x)
##  l6: var(x)
##  l7: sum(x)
## \ No newline at end of file
```

Frankly, I go to GitHub or GitLab and check the respective differences between files...

Local and remote branches

Branching

- In addition to providing a powerful undo function, Git also allows to "toy around" with different "versions" of your text or code

- Let's assume that you wrote a first draft of an R script. Everything works as expected. From a programming perspective, though, the script is just ugly and it is therefore quite hard to add additional features

- What I used to do was: save my original file as `my-great-program.R` and start working on a new version of the program using a file called `my-great-program_new.R`

- This is not necessary with `git branch`

Let's start with a list of files that are currently in my project folder:

```
## total 17
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .git
## -rw-r--r-- 1 weissbd GESIS+Group(513) 153 Nov 16 16:48 test.R
```

```
cd /e/tmp/git_test_folder
git status
```

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

What branches are available? Once we have more than one branch, the asterisk * shows which branch is active (or: in which branch we are in)

```
cd /e/tmp/git_test_folder
git branch
```

```
## * main
```

Create a new branch called `testing`

```
cd /e/tmp/git_test_folder
git branch testing
```

```
cd /e/tmp/git_test_folder
git branch
```

```
## * main
##   testing
```

How do we get into the `testing` branch? Use `git checkout testing`

```
cd /e/tmp/git_test_folder
git checkout testing
git branch
```

```
## Switched to branch 'testing'
##   main
## * testing
```

Create a new file `testingfile`

```
cd /e/tmp/git_test_folder
touch testingfile
echo "in testing" > testingfile
ls -la
```

```
## total 18
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .git
## -rw-r--r-- 1 weissbd GESIS+Group(513) 153 Nov 16 16:48 test.R
## -rw-r--r-- 1 weissbd GESIS+Group(513)  11 Nov 16 16:48 testingfile
```

```
cd /e/tmp/git_test_folder
cat testingfile
```

in testing

```
cd /e/tmp/git_test_folder
git add testingfile
git commit -m "new branch testing"
```

[testing f0197e0] new branch testing
1 file changed, 1 insertion(+)
create mode 100644 testingfile

Switch back to branch `main` (and `cat testingfile` should result in an error message, since there is no `testingfile` in branch `main`)

```
cd /e/tmp/git_test_folder
git checkout main
ls -la
```

```
## Switched to branch 'main'
## total 17
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .git
## -rw-r--r-- 1 weissbd GESIS+Group(513) 153 Nov 16 16:48 test.R
```

Now, we can use `merge` to combine `main` and `testing`

```
cd /e/tmp/git_test_folder
git merge testing
```

```
## Updating 7678c7c..f0197e0
## Fast-forward
##  testingfile | 1 +
##  1 file changed, 1 insertion(+)
##  create mode 100644 testingfile
```

```
## total 18
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 ..
## drwxr-xr-x 1 weissbd GESIS+Group(513)   0 Nov 16 16:48 .git
## -rw-r--r-- 1 weissbd GESIS+Group(513) 153 Nov 16 16:48 test.R
## -rw-r--r-- 1 weissbd GESIS+Group(513)  11 Nov 16 16:48 testingfile
```

```
cd /e/tmp/git_test_folder
cat testingfile
```

```
## in testing
```

Working with remote repositories

- As mentioned in the introduction, Git is especially powerful when it comes to collaborative work.

- In order to work with others, you need some sort of connection to these other person(s). The one I am discussing here is having a central repository C.

- Let us assume that you (x) have two other collaborators y and z. Then x (that's you), as well as y and z need to synchronize with the same repository C. There also exists another model which is based on a decentralized approach, where you could individually sync with x-y, x-z, y-z etc.

Establishing a connection to a remote repository

There are two ways to establish a connection to a remote repository:
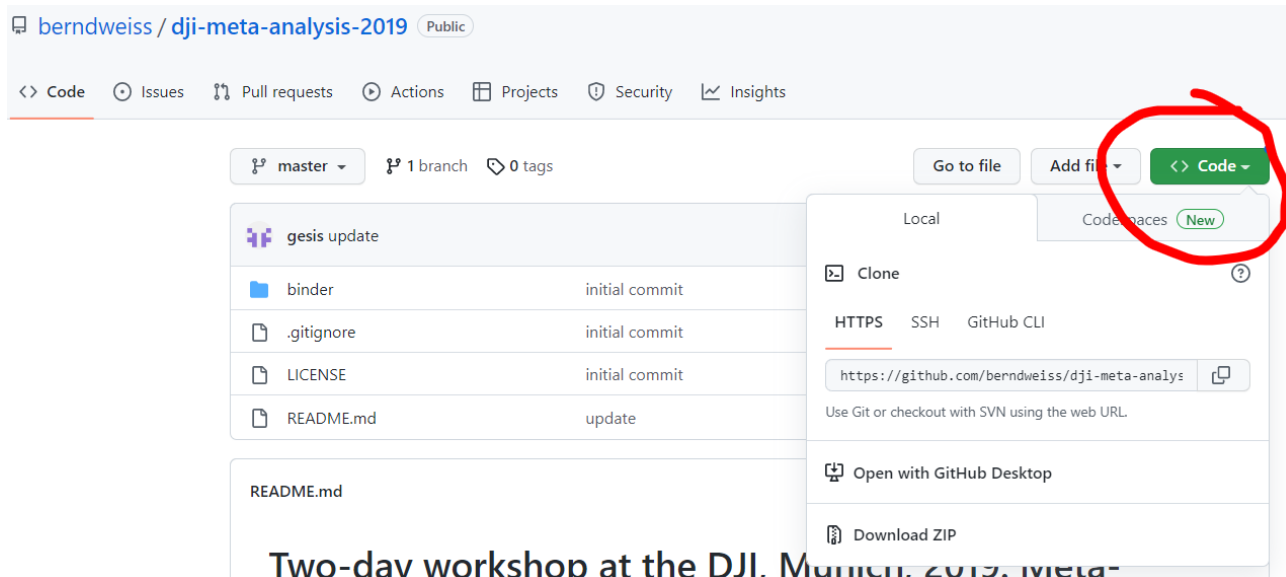
1. Clone a remote repository via `git clone ....`
2. Setting up a new remote repository via `git remote add <name> <url>`.

Cloning a remote repository

- Cloning a remote repository via GitHub/GitLab/... is quite easy

- Visit the website, on GitHub look for the green "Code" button, see also the screenshot below

- Decide wether you would like to use the HTTPS or SSH protocoll

- Copy the link and execute `git clone`

- Here is an example using my workshop on "Meta-Analysis in Social Research", see https://github.com/berndweiss/dji-meta-analysis-2019

- Open a CLI and execute

```
git clone https://github.com/berndweiss/dji-meta-
analysis-2019.git
```

Adding a remote repository via `git remote add ...`

- Using the git command `git remote add <name> <url>`. The usual name for `<name>` is `origin`, however, feel free to choose another name. The `<url>` for this repository looks like `git@git.gesis.org:weissbd/ps2017-xx-intro2git.git`; another example is this one: `git@github.com:berndweiss/ps2017-11_porto-campbell-ma-workshop.git`. The url can be found in the respective github/gitlab repository.

- The most convenient way in working with remove repositories is using SSH. In order to utilize SSH, the remote url has to be start with `git@git....`

- It is also possible to use the HTTPS protocol. In these cases the urls look like so `https://example.com/path/to/repo.git`.

Delete remote branch

```
git push <remote_name> --delete <branch_name>, e.g.
git push origin --delete my_branch
```

Dowloading a remote branch that is not on your computer (yet)

Just run a simple `git pull` (see
https://stackoverflow.com/a/2294385). Then, on your local repository checkout to that remote branch, e.g. `git checkput indepday`:

```
Switched to a new branch 'indepday'
Branch 'indepday' set up to track remote branch 'indepday' from 'origin'.
```

After checkout to `indepday`, git automatically starts tracking the new branch.

Exercise

- Start the Git Bash and choose a location where you can store temporarily some files

- Clone the respository of my workshop
  https://github.com/berndweiss/dji-meta-analysis-2019

- Change into the newly created directory

- List the Git history via `<your input --oneline>` (the `--oneline` is very handy) and determine the first 7 SHA1 digits

Things we could not cover

There is much more...

- Commit hygiene http://www.ericbmerritt.com/2011/09/21/commit-hygiene-and-git.html

- `.gitconfig`

- `.gitignore`

- ...

- A nice tutorial in German: "git - Der einfache Einstieg, eine einfache Anleitung, um git zu lernen. Kein Schnick-Schnack ;) https://rogerdudler.github.io/git-guide/index.de.html

References

Healy, K. (2019, Oktober 4). The Plain Person's Guide to Plain Text Social Science. The Plain Person's Guide to Plain Text Social Science. https://plain-text.co/

Narębski, J. (2016). Mastering Git: Attain expert-level proficiency with Git for enhanced productivity and efficient collaboration by mastering advanced distributed version control features. Packt Publishing.