MXN442

Reproducible and Collaborative Practices: Introduction

Mahdi Abolghasemi

Contents

What is reproducible research and replicability? 1

Version control 2

Distributed Version Control: Git 2

Installing Git using the command line 2

Git and Command Line 3

Command Line Basics . . . 4
Navigating between directories . . . 4
How to connect GitHub and our computer . . . 5
Configuring Git in your Rstudio project . . . 5
Clonning a github repo . . . 5
Three Git States . . . 5
Git Workflow . . . 6
General Workflow: other important commands via Terminal . . . 6
Rstudio cloud and GitHub . . . 6
Branching . . . 7
Git HEAD and checkout . . . 8
Updating new branches in the remote repo in GitHub . . . 8
Merging branches . . . 8
Avoiding confusion when creating branches . . . 8
Deleting branches using cli . . . 8
Fetching . . . 9
Fetch workflow . . . 9
Previous branches . . . 9
Merging branches successfully . . . 9
Resolving merging conflicts . . . 9
Creating branches from Rstudio Cloud . . . 10
Edit/Amend previous commit . . . 11
Removing files . . . 11
gitignore . . . 11
Creating a gitignore file . . . 12
Forking a repository . . . 12
Pull request? . . . 12

What is reproducible research and replicability?

Reproducibility and replicability are similar concepts but slightly different.

- Reproducibility: Goodman et al (2016) "refers to the ability of a researcher to duplicate the results of a prior study using the same materials as were used by the original investigator. That is, a second researcher might use the same raw data to build the same analysis files and implement the same statistical analysis in an attempt to yield the same results" That is, → different team, different experimental setup
- Replicability: "refers to the ability of a researcher to duplicate the results of a prior study if the same procedures are followed but new data are collected". That is, → different team, same experimental setup

For more details refer to: The Practice of Reproducible Research : Case Studies and Lessons from the Data-Intensive Sciences by Justin Kitzes

Version control

What is the solution? The solution is to Use **version control**. Version control systems are a category of software tools that help store and manage changes to source code (projects) over time.

There is several advantages in using version control:

- Version control software keeps track of every modification to the source code in a special kind of database.
- If a mistake is made, you can turn back to previous versions and compare the code to fix the problem while minimizing disruption.
- It is easy to manage multiple versions of a project
- It is a very useful (actually essential!) tool for collaborating and for sharing open source resources.

There are several version control software in the market. **Github** and **Bitbucket** are the most popular version control system. Both **Github** and **Bitbucket** are cloud-based hosting services that let you manage Git repositories.

Distributed Version Control: Git

We are going to use a version control system called **Git**.

"Git is a distributed version control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. It is a great tool for tracking your work and reproducibility."

Git was created by Linus Torvalds in 2005 for the development of the Linux kernel and since then many other kernel developers have been contributing to its development. The maintainer since 2005 is Junio Hamano.

Note: Git and Github names are a bit confusing. They may sound the same but they are not. Git is a version control system but Github is a cloud-based hosting service that works perfectly with Git.

We are using Git through Markdown as standard way to reproduce results using R. You can read more about markdown here

Installing Git using the command line

Git is often not installed on your machine by default. We need to install Git on our computer. See below the instruction:

Git for Mac:

- Go to Applications folder –> open Utilities –> double-click on Terminal
- Or open a Finder Window and search for "Terminal"

Once you have found the terminal:

- Open your Terminal and check if you have Git installed:
  - By typing in your terminal `git version`
  - If Git is installed you will see something like *git version 2.20.1 (Apple Git-117)*

If Git is not installed in your Mac:

- Type in your terminal –> `brew install git`
- Alternatively, you can install Git from here

Git for Linux:

- Check if you have Git installed: Type `git version` in your terminal.
- If Git is installed you will see something like *git version 2.20.1*

If Git is not installed:

- Type in your terminal –> `sudo apt-get install git`

Git for Windows:

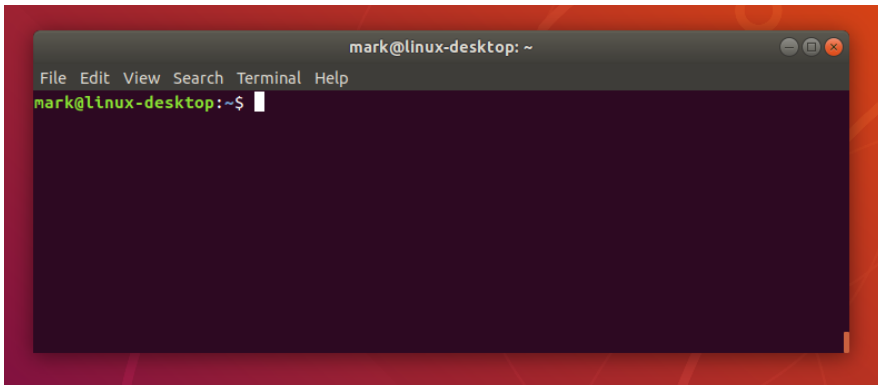
- Please follow the tutorial here. This tutorial teaches you step by step how to install Git and the command line interface in your Windows system.

Git and Command Line

To connect Git and Github, we use the shell or command line interface or terminal. The shell or command line interface is an interface where the user types commands. This interface allows us to control our computer using commands entered via our keyboard.

In most cases (non-linux users) use a Graphical User Interface (GUI) to interact with Git and Github. However, at the beginning of the computing times most people would use the command line interface to interact with their computer. That means that instead of using a GUI and our mouse to open and close programs, create folders and moving files, we are going to type commands.

We will start writing commands after ~$ or ~% depending on the terminal version that you are using. Shell looks like the figure below and you can find it on your Mac by searching for `Terminal` or Windows for `Coomandline`



```
mark@linux-desktop:~$
```

Typically when you open your terminal, it will welcome you with a prompt that looks like this:

`mahdi@computerid-macbook:~$`

or with the new Catalina Mac OX

`mahdi@computerid ~ %`

On Windows it will contain the same elements but look like this:

```
mahdi@computerid-pc MINGW64 ~$
```

Note that the commands that we are going to use are the same regardless the terminal version you have.

Command Line Basics

Here are some of the common commands:

- `pwd` : This prints the working directory/present working directory. This command will tell you what is the path to your current computer location?
- `ls`: list files inside my directory
- `ls` → gives you a list of all the elements in a directory.
- `ls -a` → list of all the files including hidden ones.

Each Linux command (pwd, ls . . .) have lots of options (flags) that can be added.

A reference list of Unix commands with options might be found here

Navigating between directories

You can use the `cd` command to change your directory.

- `cd Research` → means that we move into Research
- `cd COVID` → means that we move into COVID
- `cd` → means the current directory COVID
- `cd ..` → means go to the parent directory, we move back into Documents
- The `~` symbol is a shorthand for the user's home directory and we can use it to form paths. If you are in your Downloads directory (/Users/John/Downloads) typing `cd ~` → will bring you to your Home directory /Users/John!
- `mkdir Project1 Project2` means "make two new directories (folders)" called Project1 and Project2.
- We can create empty files with `touch`. For example, `touch example.Rmd` creates example.Rmd empty file.
- `mv` move files or folders → takes two arguments, the first being files or folders to move and the second being the path to move to. For example, `mv Project1/example.Rmd Project2` mean Move example.Rmd from Project1 into a new folder (or directory) called Project2. The folder Project1 and Project2 should be at the same level.
- `cp` → this command is used to copy files or group of files or directories.
- `rm` → remove files and folders. For example, `rm example.Rmd` removes example.Rmd.
- To remove entire folders: `rm -r` → It requires the -r (recursive) flag.

There are some more commands that you may need to know. You don't need to learn all the commands only some that are more common and we will use.

Excellent summary about the commands that we may be using can be found here

How to connect GitHub and our computer

You can use your academic email address to create a Github account. Here you can sign up for a free GitHub account.

In order to connect Github and your computer, you need to follow these steps:

1. Login into GitHub
2. Click the '+' icon on the top right on the menu bar and select 'New Repository'. You can make a repo public or private by choosing the options. The private project will be only visible to your colleagues once you give them access. The public repository will be visible to everyone that comes to your page. Also, make sure your repo is initialized with `REAME.md`: It is important to have a `README.md` file for every repository. GitHub will use this file as the "presentation" of the repository and should briefly describe what the repo is about.

Configuring Git in your Rstudio project

So far, we have installed R and Git. This is good but now we need to connect them. That is, we need to get your Git configured in Rstudio Cloud (the same follows for your own computer R). This can be easily done. Open the command line interface/terminal interface and type:

- `git config --global user.email "your.email@example.com"`
- `git config --global user.name  "Firstname Lastname"`

For example `git config --global user.name  "Mahdi Abolghasemi"`.

Make sure you use the same email address for this and for setting up your GitHub account. To check that everything is set up correctly, type this command and press enter:

```
git config --global user.email
```

If it shows your email address, then your computer and Github are connected. Once your computer is connected to Github, then your computer and Github can interact.

Note: If you work with RStudio cloud, your project will be automatically connected to the GitHub repo if you use the same email address. To work with the RStudio cloud, you need to create a new project. Go to RStudio Cloud and click the arrow next to New Project. From the dropdown menu, select "New Project from Git Repo." Then copy and paste the URL address of your repo there. Done! You have connected the repo to your RStudio cloud.

Clonning a github repo

When you create a repository on GitHub, it exists as a remote repository. Users can clone your repository to create a local copy on their own computer and sync between the two locations.

In order to clone a document you can follow these steps in your command line/shell:

1. Navigate to the computer location where we want to download the GitHub repo

2. Copy the link as appears on the *Clone or download* as shown in the figure below. Then write `git clone` and copy the address of the remote repo in your command line/shell. For example, `git clone git@github.com:okayama1/Git_demo.git`. This will create a folder in your computer with the GitHub repository files and folders.

Three Git States

Git has three main states that your files can reside in: modified, staged, and committed:

- Modified: you have changed the file but have not committed it to your database yet.

- Staged: you have marked a modified file in its current version to go into your next commit snapshot.

- Committed: the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory. In the next section, we will learn what each of these mean to us. But before that, let's look at the general workflow of Git.

Git Workflow

A typical workflow when we are working on a project is as follows:

1. We clone the remote repository. You can use `git clone` for this purpose. `git clone` is used to target an existing repository and create a clone, or copy of the target repository.
2. We modify an Rmd file from the working directory.
3. We add the modified files to the staging area to be stored. You can use `git add` command. This command adds a change in the working directory to the staging area.
4. We add the files into the git directory (This is called *commit*) using `git commit -m "type your message here"` command. The `git commit` command is used to create a snapshot of the staged changes along a timeline of a Git projects history. (m = is your message for commit)
5. We connect with the remote repository (this is called *push*) and update files. For this you can use `git push origin master`. The `git push` command is used to upload local repository content to a remote repository, in this case to the master branch.

Note: In a git repository, we will have both tracked and untracked files. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about and they are part of the git repository. Untracked files are everything else, any files in your working directory that were not in your last snapshot and are not in your staging area. In fact, their history is not tracked. We will use tracked and untracked files later. We will learn more about this in the coming weeks. You can find more about it here

General Workflow: other important commands via Terminal

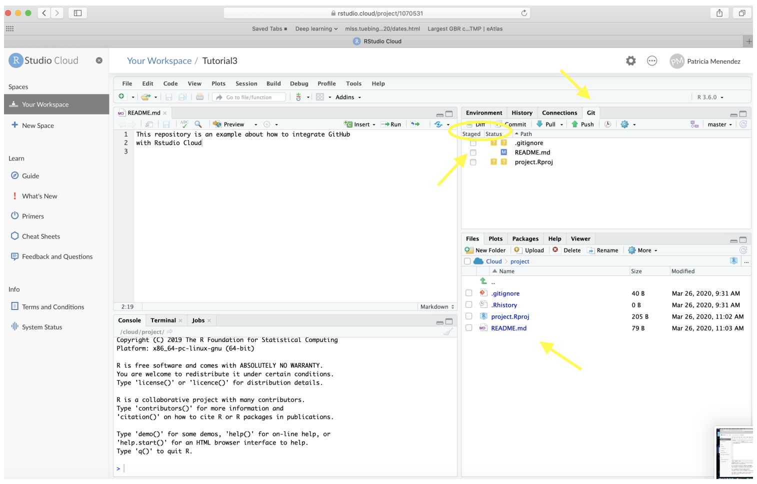
Here we introduce more commands that you will frequently use when working on your reports. There are a lot more commands and we will introduce some of them in the next couple of modules.

- `git pull`: This command is used to fetch and download content from a remote repository and immediately update the local repository to match that content.
- `git status`: This command displays the state of the working directory and the staging area.
- `git add file_name`: This command adds changes in the working directory to the staging area.
- `git commit -m "Write your message here"`: This command is used to create a snapshot of the staged changes along a timeline of a Git project history.
- `git push origin branch name`: This command is used to upload the local repository content to a remote repository in GitHub.

Excellent summary about the commands that will be using can be found here

Rstudio cloud and GitHub

RStudio cloud keeps git constantly scanning the project directory to find any filesthat have changed or which are new. By clicking a file's little "check-box" you can stage it as shown in the figure below:



README.md

```
This repository is an example about how to integrate GitHub
with Rstudio Cloud
```

Git pane:

Staged	Status	Path
	? ?	.gitignore
	M	README.md
	? ?	project.Rproj

Files (Cloud > project):

Name	Size	Modified
..		
.gitignore	40 B	Mar 26, 2020, 9:31 AM
.Rhistory	0 B	Mar 26, 2020, 9:31 AM
project.Rproj	205 B	Mar 26, 2020, 11:02 AM
README.md	79 B	Mar 26, 2020, 11:03 AM

Console:

```
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Understanding the symbols in the Rstudio Git pane:

- `Blue-M`: a file that is already under version control that has been modified.
- `Orange-?`: a file that is not under version control (yet…)
- `Green-A`: a file that was not under version control, but which has been staged to be committed.
- `Red-D`: a file under version control has been deleted. To make it really disappear, you have to stage its disappearance and commit.
- `Purple-R`: a file that was renamed. (Note that git in Rstudio seems to be figuring this out on its own.)

A few other points to have in mind:

- We can interact between Git, GitHub and our local repository using the terminal only
- We can interact between Git, GitHub and our local repository using Rstudio
- Keep refreshing Rstudio cloud as shown below, otherwise some of your branches and changes might not be updated.

Branching

Branching is a great feature of version control. You may ask why?

Because it allows you to duplicate your existing repository, use a branch to isolate development work without affecting other branches in the repository and modify a branch that can be merged into your project.

Branching is particularly important with Git as it is the mechanism that is used when you are collaborating with other researchers/data scientists.

Each repository has one default branch (master) and can have multiple other branches as shown below. It gives you great flexibility to work on your project and collaborate with others.

You can create a branch in GitHub and pull the changes in your repository. To do so, navigate to the main page of your repository on Github. Then click on the branch selector menu as shown below and type a name for your new branch.

More info on branching here

You can delete a branch by clicking on the **branches**, then go to the branch that you want to delete and simply click the delete icon as shown in the picture below:

You can use Termianl/Shell to create branches.

- `git branch`: Shows us the branches we have in our repo and marked our current branch with *
- `git branch newbranch_name`: Creates a new branch but does not move the HEAD of the repo there.
- `git checkout newbranch_name`: Moves the HEAD to newbranch_name

Git HEAD and checkout

How does Git know what branch you're currently on? By using the pointer, HEAD. In Git, this is a pointer to the local branch you are currently on. Internally, the `git checkout` command updates the HEAD to point to either the specified branch or commit.

Another way to create and checkout branches is to use the check out command. `git checkout -b newbranch_name` command creates a new branch and moves the repo HEAD to this branch. You can confirm it by using `git branch` to see in which branch you are currently in. Checking out a branch updates the files in the working directory to match the version stored in that branch. It tells Git to record all new commits on that branch.

Updating new branches in the remote repo in GitHub

We can just update the empty branch into GitHub by typing `git push origin newbranch_name`

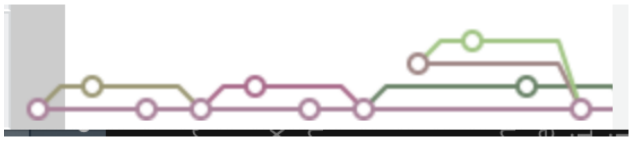
Alernatively if we had files or changes added into that branch:

- `git add .` (adding all the modified files into the staging area)
- `git commit -m  "Updating new newbranch_name"`
- `git push origin newbranch_name`

Merging branches

Switch to the branch that you want to add the stuff into (let's say that is master). Then

- If you want to merge changes into master use: `git checkout master`
- If you want to to check the status of our repo use: `git status`
- If you want to merge branches, use `git merge newbranch_name -m`
- If you want to update the remote repository, use `git push origin master`



Avoiding confusion when creating branches

Creating branches can be sometimes confusing. To avoid confusion, always make sure where your branch is starting from.

It is essential to `git checkout` before creating a new branch. If the branch where you are currently working was already merged with the master branch, you'll need to undo almost all the changes from the old branch that did not make it into the master. This makes all the old changes from that branch to appear as new changes in combination with the changes that are actually new. It makes a mess that you want to avoid!

Don't create branches from a branch that is not the master unless you are deliberately doing it.

Deleting branches using cli

To delete branches from your local repository follow these steps:

1. list all the branches first by typing `git branch -a`
2. Move to master by: `git checkout master`
3. Delete unwanted branch by `git branch -d Name_of_branch`. This will delete branch called `Name_of_branch`

You cannot delete a branch if your HEAD is on that branch.

To delete branches from your remote repository (GitHub), type `git push origin --delete Name_of_branch`

Fetching

Imagine that you are working on your local repository and a collaborator has created a new branch in your remote repo. You are currently working on your local repo and want to have a look at the new branch. That means that the local repo and your remote repo have diverged. That is they are not currently synchronized. To synchronize your work use: `git fetch origin`. This command looks where "origin" is and fetches any data from it that you don't yet have. It also updates your local database repo, moving your origin/master pointer (HEAD) to its new, more up-to-date position.

Fetch workflow

For a good practice, check branches available for checkout and make a local working copy of the branch. Then use the following commands as needed:

- `git remote`: lets you create, view, and delete connections to other repositories.)
- `git fetch origin`: fetch the changes from remote origin
- `git branch -a`: list all the branches available in the local repository + all the branches fetched from the remote.

The branches fetched from the remote origin would be preceded by remotes/origin/

To work on someone's branch, make a local copy of it. Then, work on your local branch (new branch). You can push that new branch to the remote repository using `git checkout -b Mybranch origin/name-of-remote-branch`

Previous branches

You can use `git checkout -` to go back to previous branches. Imagine that you have two branches: Master and Alternative_analysis. To check in which branch you are currently, type `git branch`. To go back to Master (assuming that you were in there) use `git checkout -`

Merging branches successfully

Suppose we have two branches: master and new_development. For merging, go to the master branch and use `git checkout master`. Then use `git merge new_development`. If those steps are successful your **new_development** branch will be fully integrated within the master branch. However, it is possible that Git will not be able to automatically merge them and you may need to resolve some conflicts. This normally happens when two branches have different parts of the same file and Git is not able to figure out which part to use. You will have to resolve them manually.

```
# Auto-merging index.html
# CONFLICT (content): Merge conflict in index.html
# Automatic merge failed; fix conflicts and then commit the result.
```

Resolving merging conflicts

First thing to do, you need to figure out which files are those affected by the conflict by using `git status`. You may see the output below:

```
git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
```

```
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#     both modified:      example.Rmd
#
# no changes added to commit (use "git add" and/or "git commit -a")
```

Open the file with a text editor. Go to the lines which are marked with

`<<<<<<, ````======``` and >>>>>>`

When you open the conflict file in a **text editor**, you will see the conflicted parts marked like this:

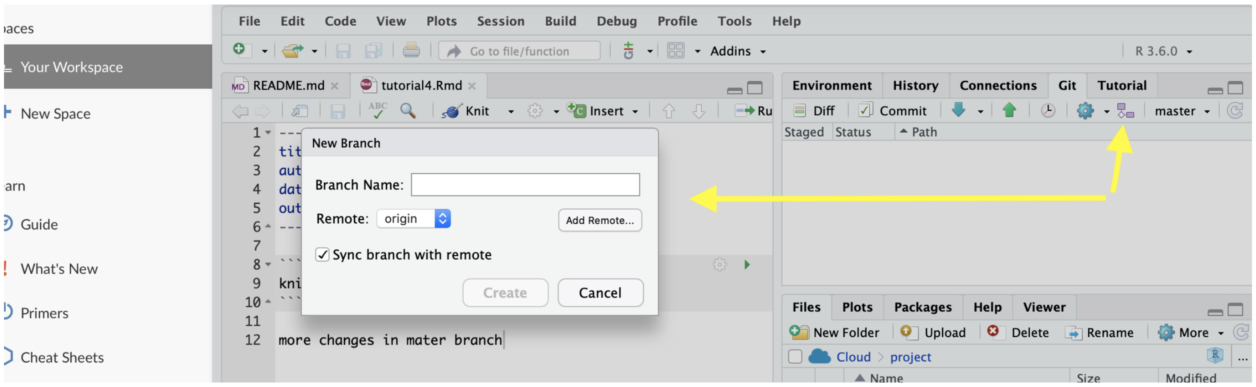
```
/* code unaffected by conflict */
<<<<<<< HEAD
/* code from master that caused conflict */
=======
/* code from feature that caused conflict */
```

When Git encounters a conflict, it adds `<<<<<<<` and `=======` to highlight the parts that caused the conflict and need to be resolved.

Decide which part of the code you need to keep in the final master branch. Remove the irrelevant code and the conflict indicators. Finally, Run `git add` and `git commit` commands on the conflicted files to generate the merge commit.

Creating branches from Rstudio Cloud

You can create branches directly on your RStudio cloud by clicking on the icon as shown in the picture below:



New Branch

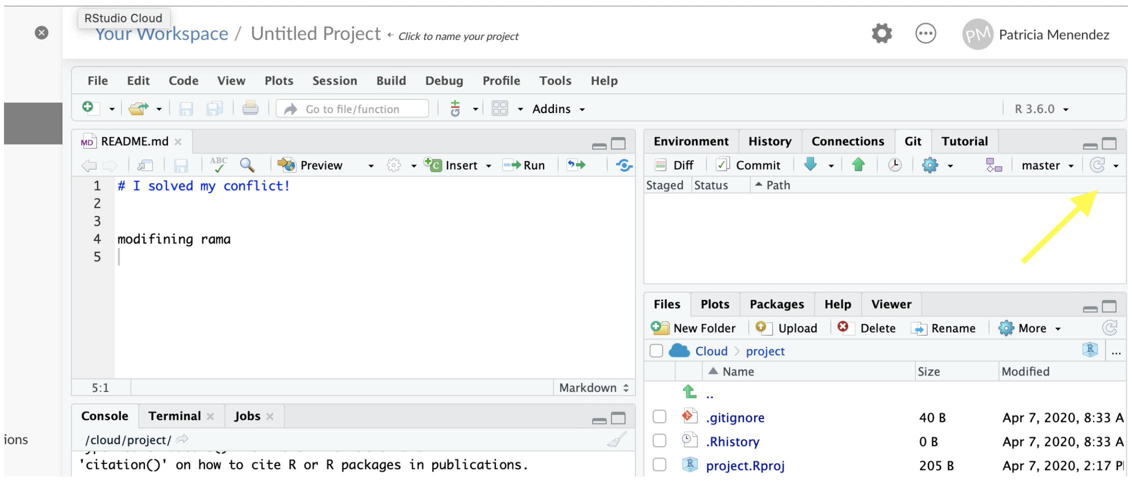
Branch Name:

Remote: origin

✓ Sync branch with remote

```
more changes in mater branch
```

Remeber to keep refreshing Rstudio cloud by hitting the referesh icon as shown in the picture below, otherwise some of your branches and changes might not be updated.



```
# I solved my conflict!


modifining rama

```

```
/cloud/project/
'citation()' on how to cite R or R packages in publications.
```

Name	Size	Modified
..		
.gitignore	40 B	Apr 7, 2020, 8:33 A
.Rhistory	0 B	Apr 7, 2020, 8:33 A
project.Rproj	205 B	Apr 7, 2020, 2:17 P

Now we know what we need to know about branches. There are a few more commands and concepts that we need to learn. Next, we will learn more about pulling and pushig.

Edit/Amend previous commit

You can use `git commit --amend` command that will open the Vim. In your text editor, edit the commit message, and save the commit. Use `git push --force origin branch` to overwrite the previous commit.

Note: Amending commits which are already pushed to a remote are more difficult to apply and would require a force push for the rewrite. Also it will require rebase.

Removing files

If the files are untracked you can just remove them. If the files are tracked there are two possible ways of deleting them:

- Remove the file yourself by following commands:
  - `git rm - file.txt`
  - `git commit -m "Delete file.txt"`
  - `git status`
- Use git to remove the file bu the following commands:
  - `git rm file.txt`
  - `git status` (file already added into the staging area)

gitignore

Ignored files are tracked in a special file named **.gitignore** that is checked in at the root of your repository. There is no explicit git ignore command: instead Git uses a **.gitignore file** which must be edited and committed by hand when you have new files that you wish to ignore. **.gitignore** files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

There are different types of gitignore patterns. Each line in a gitignore file specifies a pattern.

You can see the source here

You can use the characters as those in the table. You can also add comments inside your file using '#'

```
# ignore all logs
*.log
```

You can use '' to escape .gitignore pattern characters if you have files or directories containing them:

```
ignore the file literally named example[01].txt
example\[01\].txt
```

Creating a gitignore file

From the command line you can create a *.gitignore* file for your repository using `touch .gitignore`. You can edit the file using Rstudio (Terminal).

You can also create a *.gitignore* file from GitHub and edit it further later using Rstudio (Terminal).

If you want to ignore a file that is already checked in, you must untrack the file before you add a rule to ignore it. From your terminal, untrack the file using `git rm --cached FILENAME`. The *gitignore* file specifies intentionally untracked files to ignore.

Forking a repository

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. Most commonly, forks are used to either propose changes to someone else project or to use someone else project as a starting point for your own project. Note that you can not fork your own repository. You can fork the repositories that are owned by someone else to work on them.

In order to fork a repo in GitHub, simply navigate to a repo from within your Github account and hit the *Fork* button.

You can find more about forking here

Pull request?

We use GitHub to share our code and projects with others. There are situations when another person makes changes to your code and wants you to consider those changes. You can do this by sending a request to the repo's owner to pull/merge these changes into the owner's original GitHub repo That request is called a pull request.

You can create a pull request to the original repo simply by hitting the `New pull request` button.

When you start a pull request, you will need to determine your base and head repository. Also, you need to choose a branch by which you are going to compare the executed request and write your message.

Once you pull a request, the owner will receive an email. Your pull request to the owner of original repo will look like this on the owner side.