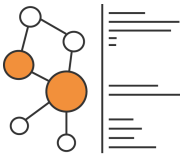

20.440 Problem Set #4

Reproducible, shareable code

Due 31 March 2021

Instructions Work together, but all results must be independently generated and presented.

If there are extraneous factors – whatever they may be – preventing you from completing this assignment, reach out. We are here to listen, help, and make sure you are successful in 20.440

Strive for concise, clear writing and descriptive plots. Refer to the MIT BE Communication Lab for assistance or feedback.

For this problem set, submission will be slightly different. Each member of the class should upload a single PDF file to Canvas containing the following:

Project title
Team member
GitHub repository link if public/note that repository is private

Grading will focus on the repo itself; you will not be graded on your code or figures directly, but rather our ability to understand and reproduce your analyses.

Purpose This problem set is designed to provide students with experience in using GitHub as a tool for managing computational projects. There will also be a focus on best practices for writing and distributing code, with the aim of ensuring reproducibility and usability.

Resources General guides for using GitHub:
https://help.github.com/en/github
https://guides.github.com/
https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners

Markdown cheatsheet:
https://www.markdownguide.org/basic-syntax/

Key sources Example GitHub repos:
https://github.com/cduvallet/microbiomeHD
https://github.com/dynverse/dynbenchmark/

Question 1

In this question, we are asking each of you (as individuals) to make a GitHub repository that contains code and sufficient documentation needed to reproduce a single figure from your project. You are being asked to do this individually to (i) ensure that all team members understand the project and the data and (ii) to ensure that all team members learn how to use GitHub. For the purpose of this tutorial, we will assume you are using a Unix-like terminal. If you are a Windows user, we recommend using Git Bash as your terminal.

1.1 Get started with GitHub (no turn-ins)

For the purpose of this walkthrough, we'll configure a repo from scratch, then commit some code to this repo. We'll also provide toy examples of branching and merging, which may be useful for you in your final project. There are no turn-ins required for this part of the assignment.

1.1.1 (if not using the VM)

Install git by following the Installation Guide https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

If you're a Windows user, we also recommend installing git bash https://gitforwindows.org/

Also install the following:

git-lfs (if working with local datasets that are >500 MB)
nbstripout and nbdime (if using Jupyter notebook)
a text editor such as Atom, Sublime, or Vim
command-line tools for your text editor, if desired

1.1.2 Create a GitHub account if you don't already have one. You can either sign up for GitHub using MIT enterprise: https://github.mit.edu/, which allows you to create repos that are private to the MIT organization, or by creating a public GitHub account. *If you make a private repository you must add the TAs as collaborators on the project to allow us to see and grade your work.*

Cal: cgunnars (public), gunnars (MIT)
Megan: megantse (MIT)
Bianca: biancalepe (public), blepe (MIT)

1.1.3 Create a remote GitHub "repository" by navigating to github.com/new. We'll be making and uploading an existing repository, so you can leave initialization options unchecked.

1.1.4 Create a local git repository, either in an existing folder or a new folder.

In Terminal or Git Bash (or your OS's Terminal-like program), make a folder (`mkdir`), navigate to it (`cd`), and initialize git in that folder:

```
mkdir myfolder
cd myfolder
git init
```

If successful, the message "Initialized empty Git repository in <folderpath>" will appear, indicating that a .git folder has been created in the directory. This folder is where git stores information needed to reconstruct the version history of your project.

1.1.5 OPTIONAL - Tell GitHub who you are.

```
git config --global user.name "your username"
git config --global user.email "your email"
```

Note: it is possible to set your password using git config, but this will store your password in plain text on your computer, which is not secure. Instead, we recommend either (1) entering your password each time you want to pull from/push to your remote repo or (2) generating a passcode-protected SSH key and adding the SSH key to GitHub https://help.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

1.1.6 Type `git status` to see the status of your repo. Since you've just created your local repository, there should be no files currently tracked, no unstaged changes, and no changes to commit.

1.1.7 Tell git to ignore certain files by creating and populating a .gitignore file. Common files to ignore include "autosave" files (e.g. __pycache__ and .ipynb_checkpoints) and OS-generated files (Thumbs.db and .DS_Store). You can find language-specific .gitignores here: https://github.com/github/gitignore.

```
touch .gitignore
```

1.1.8 Tell git to handle files specific, default ways by creating a .gitattributes file.

```
touch .gitattributes
```

In particular, you'll want to use git-lfs to track any large files.
https://help.github.com/en/github/managing-large-files/configuring-git-large-file-storage

```
git lfs track <big_file_name.extension>
```

We don't recommend using Jupyter notebooks for your final project, aside from initial data visualization and exploration, because Jupyter notebooks are difficult to version control. However, if you choose to use Jupyter notebooks with git**,** you'll also need to manage how git determines whether and which changes have

occurred since last commit with nbstripout and how git deals with conflicting changes with nbdime.

```
nbstripout --install --attributes .gitattributes
git-nbdiffdriver config --enable
git-nbmergedriver config --enable
```

Jupyter notebooks encode information about cell execution count and output directly into the code. This means that running a Jupyter notebook changes its source code, and this would register as a "change" to be committed. You can use the package nbstripout to tell git to clear Jupyter notebooks before committing the files *https://github.com/kynan/nbstripout*

When you push changes, git will try to merge your changes with the branch you push to. If there is a difference between the files that git doesn't know how to resolve, git will insert conflict markers `<<<<<<<`, `=======`, `>>>>>>>` *and you will need to resolve this merge conflict by editing the text file. git's conflict markers often break the JSON format used by Jupyter notebooks, so you can no longer view them on a web application. JSON format isn't very human-readable, so it can be hard to resolve merge conflicts without introducing syntax errors. You can use the package nbdime to tell git to use nbdime's diff and merge drivers* *https://nbdime.readthedocs.io/en/latest/vcs.html*

1.1.9 Check `git status` again to see an updated list of untracked files and unstaged changes. You should see that your `.gitignore` and `.gitattributes` files appear as "Untracked files."

1.1.10 Track your `.gitignore` and `.gitattributes` files. These files are new, so tracking them will tell git they exist, that they have been modified since the last commit, and since they have been modified, to stage them for a commit.

```
git add <filename_here>
```

You can also track the whole directory with

`git add <directory_here>` or `git add .`

You can also track files matching a specific pattern

`git add *.zip` (tracks all .zip files in the home directory of your repo)
`git add patt-*.zip` (tracks all files with this pattern in home directory)
`git add */*.zip` (tracks all .zip files in all directories of your repo)

1.1.11 Create and track `README.md`. This file is the first thing most people read to learn more about your project. You'll edit this file later in **1.2.**

1.1.12 Check `git status` again. You should see that your `README.md`, `.gitignore`, and `.gitattributes` files moved from "Untracked files" to "Changes to be committed."

To see what changes have been made to a file since the last commit, you can use

```
git diff <filename>
```

1.1.13 Make your first commit. This will save all changes to staged files and a message to your local repository.

```
git commit -m "<informative_message>"
```

1.1.14 Check the log of commits using `git log`. Your first commit will show up in the following format:

```
commit <hash> (HEAD -> master)
Author:
Date:
    <informative_message>
```

The code next to your commit is a "hash," which uniquely identifies this commit. master is the name of the default "branch." HEAD is the name of a "pointer," which points to the commit that you most recently "checked out."

1.1.15 If you haven't already, move your code (and data) into the directory, track these files, and commit with an informative commit message.

You can also interactively choose which files to track or which chunks of code to stage for a commit using `git add -p`

Note: ***Be sure to make sure your data is being tracked by git-lfs before adding and committing!*** *If you forget to do this, you won't be able to push your commit later. You can't fix this by simply adding another "good" commit where you track the data with git-lfs after this "bad" commit. Instead, you'll need a way to get rid of the "bad" commit, for instance by squashing the "good" and "bad" commits together into a single commit or by resetting the commit history to before you made the "bad" commit and re-committing.*

1.1.16 Check `git log`. There is now a new commit with a unique hash, and HEAD is now pointing to this new commit.

1.1.17 Once you are ready to connect your local repo to your remote repo, add the remote repo, which we name "origin" by convention

```
git remote add origin https://github.com/<username>/<remotename>.git
```

or if using SSH

```
git remote add origin git@github.com:<username>/<remotename>.git
```

and push changes from your local repo to your remote repo

```
git push -u origin master
```

The `-u` flag sets the default push/pull location to the "origin" branch of the remote repo.

You can also pull changes from a remote repo to your local repo using a similar syntax. If your local repo is out of sync with your remote repo, you'll need to pull changes from the remote repo before you can push yours. (This is most likely to happen if you have multiple contributors to a remote repo, and you shouldn't have to do this for this tutorial.)

```
git pull origin master
```

This command is analogous to

```
git fetch origin
git merge origin master
```

We will learn more about merging in the next section.

OPTIONAL: BRANCHING

Branching allows you to implement new features and fixes to code without disrupting the existing `master` code base.

1.1.18 To go back to a previous commit, use `git checkout` with the hash of your desired commit

```
git checkout <previous_hash>
```

or move N commits backward (to move back to the previous commit, N=1)

```
git checkout HEAD~N
```

You'll receive a warning that you are in a "detached HEAD" state. `git log` will show you that HEAD now "points" to the commit you just checked out, rather than your most recent commit.

1.1.19 We'll now create a branch called branch1

```
git branch branch1
git checkout branch1
```

will create the branch, then move us to the latest commit in the new branch, or we can perform this action all-in-one with

```
git checkout -b branch1
```

Your HEAD pointer will now point to this branch. Make a dummy file (`touch branch1.txt`), track, and commit this change.

1.1.20 Create two more branches with the same parent (starting location) as branch1. For each branch, create one dummy file (e.g. `branch2.txt`), track the file, and commit the change.

1.1.21 To see the list of branches, use `git branch -a`. The branch your HEAD is pointed to will be highlighted in green and starred. To visualize all branches as a tree graph, use `git log --graph --all`.

In this representation, commits at the bottom right are the oldest, and commits at the top left are the newest. The first commit is known as the "root" commit. All subsequent commits are linked to a "parent" commit with dashed lines.

1.1.22 `Checkout` each branch (`master`, `branch1`, `branch2`, `branch3`), and the contents of your project directory (using `ls` or your File Explorer) as you change branches.

You'll notice that the project directory in `branch2` does not contain files committed to `branch1`, and vice versa. Instead, for each branch, the project directory contains only the files committed to that branch, as well as any parents of that branch.

1.1.23 Merge `branch2` into `branch1`. Make sure you're not checked into either of the branches you're trying to merge, or you'll receive an error.

```
git merge branch2 branch1
```

Using `git log --graph --all`, you'll see your commit from `branch2` is now connected to `branch1`, HEAD now points to `branch1`, and `branch1` now contains files committed to both branches.

We can now remove `branch2`, since it has been merged into `branch1`.

```
git branch -d branch2
```

OPTIONAL: RESOLVING MERGE CONFLICTS

1.1.24 To simulate a merge conflict, we'll create and commit two files with conflicting text, one in `branch1` and one in `branch3`.

```
echo "x = 1" > merge-conflict.txt
git add merge-conflict.txt
git commit -m "<informative message>"
git checkout branch3
echo "x = 3" > merge-conflict.txt
git add merge-conflict.txt
git commit -m "<informative message>"
```

Running `git merge branch3 branch1` will fail, and a message will appear that the "Automatic merge failed." `git status` will tell us we have unmerged paths, where both paths have attempted to add the file `merge-conflict.txt`.

1.1.25 Open `merge-conflict.txt`. You'll see that git has inserted `<<<<<<<`, `=======`, `>>>>>>>` to delineate the conflicting edits. To resolve the merge conflict, delete these markers and any edits you do not wish to keep. When you commit the file, the merge will be completed, and you can delete `branch3`

```
git add merge-conflict.txt
git commit -m "merged and resolved conflict in merge-
conflict.txt"
git branch -d branch3
```

1.1.26 Since `branch1` was only created to illustrate branching, merging, and resolving conflicts, we'll delete it, resetting your repo to the original state in **1.1.17**

```
git checkout master
git branch -D branch1
```

Using the `-D` flag, rather than the `-d` flag, will force deletion regardless of whether the branch has been merged.

1.2 Distributing and documenting your project code with GitHub

Following the GitHub tutorial from **1.1** (or another method), create a GitHub repository for your project containing:

- Data and code required to generate one figure from your project
  Note: we know your project is a work in progress! Your figure does not have to be a final version. For example, you may choose to upload the data and code used to generate the figure from your 2-pager.
- Documentation (README.md) sufficient to understand and reproduce the

analysis conducted to generate one figure from your project

- **Overview**
  At a high level, what does your repo contain / do?
  Include citations for any scripts your team did not write that are included directly in the repo and any non-standard methods that your analysis is based on.
- **Data**
  At a high level, how was the data generated?
  If it's too large to upload to your GitHub, where can it be accessed?
  Include citations, if any.
- **Folder structure**
  At a high level, what is in each folder and subfolder?
- **Installation**
  How do I run your code?
  What software and package versions do I need to install?

Note: running pipreqs on your code / directory will generate a list of dependencies for .py files. You can confirm the output and the reproducibility of your code by creating a fresh virtual environment, installing the dependencies within your virtual environment, and running the code.

You will be graded on the following criteria:

- Clarity and reproducibility of directions to reproduce your figure (50 marks)
- Detailed description of repo purpose, structure, and data (30 marks)
- Relevant citations for code, methods, and data (10 marks)
- Logical repository organization (10 marks)
  - Code, data, and figures should be separated into folders

Turn in a .pdf file containing your name, your project title, and a link to your GitHub repo, along with a note whether your repo is public or private. **If your repo is private, you must add the TAs as collaborators so we can see and grade your work.**