GitHub Workshop 2020

Marios Fokaefs, Aquilas Tchanjou, Paraskevi Massara, Elena Comelli

marios.fokaefs@polymtl.ca, aquilas.tchanjou-njomou@polymtl.ca, p.massara@mail.utoronto.ca, elena.comelli@utoronto.ca

Day 1

1. Create a GitHub repository from the web interface

1. Make sure you login with your GitHub account.
2. On the left of your home screen you should be able to see a list of your repositories, which may be empty if this is your first time with GitHub. At the top of this list, you will find a green button labelled “New”. Click on this button to create a new repository.
3. In the next screen you will be asked to add information about your new repository, among which a name for your repository, an optional description, whether it will be public or private, if you want to initialize your repository with a README and if you want a .gitignore file and a license. We will see each of these in details.
    a. NAME: As the advice says, repository names should short and memorable. For our purposes, put “workshop-sandbox-web”. Do not worry that all of you will use the same name as your repository will be uniquely identified by your repository’s name AND your username. This means that you cannot use the same name for two of your repositories.
    b. Description: As we have said, this is optional. If you define a README page for your repository, the description is not really necessary.
    c. Public vs Private: A public responsible can be searched for by users in GiHub and be found base on keywords found in the name, the README page or the description. If you want to protect your code, your data or your documents in the repository, it’s better to make it private. In this case, only users who have been given access to the repository can see it, clone it, commit to it or fork it (we will define these terms later). GitHub used to have restrictions on how many collaborators you could have in private repositories, but these were lifted recently. For the time being, make your repository public.
    d. README: This creates a “first page” for your repository. Here, you can specify the description of your project, probably include a user manual, how people are supposed to treat your repository (how you allow them to extend it or contribute to it) and other useful information. If you select to initialize the repository with a README, GitHub will create a branch for you, which will help you enormously with your first interactions with your repository. So, we will tick this option for our repository.
    e. .gitignore: This is a file that informs GitHub (git in particular) that there are some types of files that are not necessary and should not be committed to the repository every time you submit a change. These are often files that are automatically generated and change every time we do something with the code. The most common example is binaries as the product of code compilations. In some cases, every time you save or you run your

code, these files will be produced over and over again. In this case, it does not make much sense to commit these changes in these files. So, you tell GitHub to simply ignore these files every time you commit (so you don't have to do it manually every time). But how do you know what files you can ignore? Thankfully for us GitHub provides some default gitignore files, based on the language or the technology you'll use in your project. In our case, we will predominantly use R, so we specify this as our gitignore file.

   f. license: This is an important decision, especially concerning projects that involve privacy or copyright. Unfortunately, we will probably not have the time to cover in this workshop (hopefully in a future one). Thankfully for us, GitHub provides a good guide on how to pick a license and what all the licenses include (https://choosealicense.com/). For our project, we will go "simple and permissive" and go with the MIT license which "lets people do almost anything they want with your project, like making and distributing closed source versions".

4. Once everything is set up, go ahead and click the green button "Create repository" at the bottom of this screen.
5. Congratulations! You have created your first repository! Doesn't it look cool??

2. Create a GitHub repository from the app

1. Make sure you have downloaded, installed and set up the GitHub Desktop app.
2. Select File->New repository…
3. You will be asked to provide the information for the new repository, including name, description, README, .gitignore, and license, as previously. GitHub Desktop will ask you for one extra field, which is the local path. This is the directory where all repositories (new and cloned) will be stored in your computer. When installed, GitHub Deskop will create a folder by default. You can leave it as is for now.
4. For the name use "workshop-sandbox-app" and the rest of the parameters as in the previous step.
5. As we are informed by the GitHub Desktop, the repository is only available in our local machine (as opposed to what happens when we created the repository online). Therefore, we have to publish the repository to GitHub, which can be done by the blue button in the application interface.
6. Once we do this, we are asked for a little more information. The name and the description are usually inherited by the repository we just created. If you want the repository to be private, you can keep the respective box checked. For this example, you can uncheck it. You can leave the Organisation as "None", as we have not created any organisations yet. Make sure from the top of this dialog box to select "GitHub.com" since we do not have an upgraded account for GitHub Enterprise Server.
7. Once everything is set, hit "Publish repository" once again.
8. From here we can go to our repository by navigating to github.com or by hitting the "View on GitHub" button from the app's interface.
9. As you can see, the two processes produce almost identical results.

3. Clone a repository from RStudio

1. Open RStudio

2. Go to File->New Project…->Version Control->Git
3. Go to your repository on github.com and click on the green button labelled “Clone or download”.
4. Grab the repository URL and paste on the field in the dialog box in RStudio.
5. In RStudio open dialog box. As the project directory name use the repository name and leave subdirectory path as the default GitHub directory.
6. RStudio will automatically navigate you to the write directory and also make it your working directory.
7. You will notice that RStudio adds one more file to your repository (the project file) and changes the .gitignore file to add files with the .Rproj.user extension.
8. From then on, we are ready to start making changes.

4. Clone a repository from GitHub

1. The simplest way to clone a repository is through the GitHub Desktop app. Now, concerning how you start the process, you still have two options: from the web or from the app.
    a. From the web: In the web interface of the repository, you can click the green button labelled “Clone or download”. This will give you a number of options, including to use Git through the command line, or download the contents of the repository as a compressed file. For our case, we will select the “Open in Desktop” option. This option assumes you have already installed and set up the GitHub Desktop app. If you haven’t, you will be prompted to do so.
    b. From the app: Select File->Clone repository… Initially, you will be presented with a list of repositories to clone from your GitHub account. Since we do not own the repository we want to clone, we will not find in this list. So, from the top select the “URL” tab. There you can either use the URL which is provided by the web interface when you hit the green button “Clone or download”. Alternatively, you can simply specify the username/name of the repository (remember: this combination uniquely identifies a repository). For our case, put “{username}/workshop-sandbox-web”, where you can replace {username} with your actual username in GitHub. You can leave the default local path as is.
2. Whichever method you pick, once you hit clone in the app, the repository will open in the app. Also, if you navigate to the default location you will find that a directory with the name of the repository, and its contents, has been created there.
3. From then on, we can start making changes or creating new content (using RStudio for example) for our cloned repository.

5. Version Control: My first commit

5a. Version control using RStudio

1. Open RStudio and make sure that the project related to the repository is open.

2. Take the R script you have written and move it to the directory created for the repository in the default GitHub directory.
3. After finishing your edits, return to RStudio and go to Tools->Version Control->Commit and select the R script you just added. At this point, you will be asked to also provide a short description of the changes you have made. In the version control vernacular, this is known as a "commit message" and it has been made mandatory by such systems as GitHub. The commit message is vital as it informs developers what has happened and it helps them to track the history of the project.
4. After you complete your commit message you can select "Push Branch". This action in GitHub is known as "Push to the stream", where the stream is the repository. This is because Git uses a "two-phase commit". In the first phase, we commit changes in our local repository. At this point the history is tracked in our local repositories, but changes are not visible to other collaborators yet. To make them visible, we need to push our local commit to the stream (usually called the "master branch").
5. Now your changes have been uploaded on GitHub.

5b. Version control using GitHub Desktop App

1. Take the R script you have written and move it to the directory created for the repository in the default GitHub directory.
2. Open your GitHub Desktop App and make sure you are in the workshop repository. To check that, in the top left corner, make sure that the tab "Current repository" points to the proper repository.
3. If you have copied the file, you will see that there are changes to check in the list on the left of the interface.
4. On the right, you can see the changes on the file you have selected from the left, in terms of lines added or deleted. A modified line is counted as one line deleted and another line added.
5. At the bottom left, there are two fields, one for the commit message and another for the description. Being mandatory, the Desktop app automatically generates a commit message based on the context of the action, in this case "Create file". If you wish, you can change the default message to something more descriptive. For our purpose, we can leave it as is.
6. You can also add a more extended description, which is optional.
7. Once everything is done, click "Commit to master". "master" is the main branch. If we had different branches, we would have to select the desired branch from the second tab at the top of the interface.
8. After committing, we should also push to the stream, by clicking on the "Push origin" button.
9. Now your changes have been uploaded on GitHub.

6. Creating issues in GitHub

1. Issues are an important element of version control. Although named issue, it does not refer only to problems. If you have experience with making requests to a technical support desk, you must be familiar with the concept of "tickets". Essentially, issues are used for any kind of planned change: adding a new feature, creating an enhancement, fixing a problem.

2. In order to create an issue, select the “Issues” tab from the top of your repository, next to the “Code” tab. In this view, normally you would get a list with all the open features. You will be able to search or filter through this list to navigate all the issues.
3. In order to better organize the issues, it is highly recommended (although not mandatory) to label your issues in order to better communicate them to the rest of your team. Write next to the filter field, you will see a button called “Labels”. If you click it, you will see a set of 9 labels that GitHub has defined beforehand for you. The list include such descriptive labels as “bug”, to define a problem, “documentation”, related to non-code artifacts, “duplicate” to say that this issue has been mentioned before, “enhancement” for a new or improved feature, “good first issue” for newcomers, “help wanted” to draw the attention of your collaborators, “invalid” to show that there is a problem with the issue itself, “question” if you have a question about the project, and “wontfix” if you feel that the issue is irrelevant or not a priority. Of course, GitHub allows you to define new labels possibly more specific to your project, for example “data” or “analysis”. As you can imagine labels and issues are a good way to communicate with “the world out there” meaning that your users can create issues to inform you of problems or questions they may have.
4. Now let’s go back, by hitting the “Issues” tab once again, and let’s create our first issue. At the top right corner hit the green button labelled “New issue”.
5. When creating an issue, the first thing to specify is the title. This should be something brief but descriptive, like a sentence. Remember from your script there is a 4th section related to multi-dimensional clustering. Let’s add that as a task.
6. Then we can write a longer description, where we can describe our task or our problem to better guide the developers. The description can be in Markdown which allows for rich format text. In this way, we can play with the emphasis on particular tasks, we can add links, or lists, or even pieces of code, and GitHub will format them appropriately. Something that we will see later is that we can also refer to particular commits in the history of our projects or directly tag a collaborator. For now, let’s write something descriptive about our task.
7. Besides title and description, we can specify some other properties for the issue, although they are not mandatory. A first property is the assignee. There we assign the responsibility of this issue to a collaborator. This person will be notified, and this issue will be added in their todo list. We can also assign the issue to ourselves! This is good for our self-organisation!
8. Next, we need to add a label or two. Given the nature of our change we will add “enhancement”. We can also add the first issue label as well!
9. Since we have not talked about Projects, Milestones or Pull requests we will leave these properties for tomorrow.
10. Now we are ready to submit our new issue.
11. When we do so, we will be navigated to the issue page. As you can see this contains the title, the index of the issue, its status (open), who opened it and when. Underneath we can see the discussion around this issue, with the first comment being our description, the labels and the assignee. We or someone else from the team can add comments to our issue to ask or make clarifications, check or update about the progress and generally discuss the issue.
12. If we go back to the issues tab, we can see our issue added to the list with its basic information apparent. If we click on it, we will go back to its page.
13. So, now we are ready and go do the task that corresponds to this issue.

7. Making changes and closing issues

1. Add Section 4 in the clustering script.
2. In RStudio go to Tools->Version Control->Commit, Stage all changes and add a commit message. Hit Commit and then the Push button at the top right corner to push to the stream.
3. Let's go back to the web interface and refresh to see our commit.
4. If we click on the "commits" tab, we can effectively see the history of our project.
5. At the right of each commit, there are three buttons. The middle button is labelled with the commit index. If we click on it we can see all the changes that were submitted with this commit.
6. At the top right, there is another index next to the word commit. This is essentially a longer version of the commit index. Let's copy it for now as we will use it later. If you want you can paste it in a notepad file so that you don't lose it.
7. Now that the task related to the multidimensional clustering is finished, we can go and inform our collaborators. Not by email or by char, but simply by closing the corresponding issue!
8. So, navigate back to the issues from the tab and click on our issue.
9. It is always good to accompany the closing of an issue with an appropriate comment. Closing an issue can mean that the task was completed successfully or that it was abandoned, in which case a comment will make things clear. Since in our case, everything went well we can give an appropriate message and prove it by including a reference to the commit that corresponds to the task. So go ahead and paste the long commit index that you copied before. Then, hit the button "Close and comment".
10. In the next phase, you will notice two things. First the long commit index that we have copied has transformed back to the short index and it is now a clickable link. If we click it, we will be navigated back to the commit. This is very helpful to inform collaborators exactly what has been done. If we simply hover over it we get a very short summary about the commit message and the size of the change. Back to the issue page, you will also notice that the issue is marked as closed, who closed it and when.
11. Of course we always have the option to reopen the issue if we deem that the task was not sufficiently completed and explain why with a comment.