Contents

1 Introduction to the tools that will be used.

1.1 git-annex.

1.1.1 intro.

git-annex allows managing files with git, without checking the filecontents into git. While that may seem paradoxical, it is useful when dealing with files larger than git can currently easily handle, whether due to limitations in memory, time, or disk space.

- In our framework, it will be used to keep hold of the multiple versions of the files that are used during the process of research and data analysis. It's imperative to keep track of the versions so as to facilitate the task when reproducing a result. (in this case most files are large and git-annex is the best at handling the size).

1.1.2 useful links.

`https://git-annex.branchable.com/walkthrough/` `https://git-annex.branchable.com/special_remotes/external/`

1.2 Zenodo.

1.2.1 intro.

Zenodo is a general-purpose open-access repository developed under the European OpenAIRE program and operated by CERN. It allows researchers to deposit research papers, data sets, research software, reports, and any other research related digital artifacts.

- We will be using Zenodo as the database where the articles and research papers will be deposited at the end of the mooc. The API is easily accessible through Python with the use of the package requests which allows the use of the basic HTTP queries.

1.2.2 useful links.

`https://developers.zenodo.org/#quickstart-upload`

1.3 Datalad.

1.3.1 intro.

DataLad builds on top of git-annex and extends it with an intuitive command-line interface. It enables users to operate on data using familiar concepts, such as files and directories, while transparently managing data access and authorization with underlying hosting providers. A powerful and complete Python API is also provided to enable authors of data-centric applications to bring versioning and the fearless acquisition of data into continuous integration workflows.

1.3.2 additional functionalities to Git / git-annex.

- minimize the use of unique/idiosyncratic functionality.
- simplify working with repositories.
- add a range of useful concepts and functions.
- make the boundaries between repositories vanish from a user's point of view.
- provide users with the ability to act on "virtual" file paths.

1.3.3 key points.

- DataLad manages your data, but it does not host it.
- You can make DataLad publish file content to one location and afterwards automatically push an update to GitHub.
- DataLad scales to large dataset sizes.
- dataset = superdataset = subdataset unless it's registered in another dataset: sub / super.
- Converting an existing Git or git-annex repository into a DataLad dataset: $ datalad create -f

1.3.4 exporting the content of a dataset as a ZIP archive to figshare.

Ideally figshare should be supported as a proper git annex special remote. Unfortunately, figshare does not support having directories, and can store only a flat list of files. That makes it impossible for any sensible publishing of complete datasets. $ datalad export-to-figshare [-h] [-d DATASET] [–missing-content {error|continue|ignore}] [–no-annex] [–article-id ID] [PATH] (*)

1.3.5 useful links.

`https://handbook.datalad.org/en/latest/basics/101-180-FAQ.html` `http://docs.datalad.org/en/stable/generated/man/datalad-export-to-figshare.html` (*) `http://handbook.datalad.org/en/latest/usecases/ml-analysis.html` `https://handbook.datalad.org/en/latest/beyond_basics/101-168-dvc.html`

1.4 Snakemake.

1.4.1 intro.

Snakemake is a workflow engine that provides a readable Python-based workflow definition language and a powerful execution environment that scales from single-core workstations to compute clusters without modifying the workflow.

- This will be used to handle the workflows in our project and and it's useful in the case of reproducibility because it allows the definition and the use of environments. For instance, we can define a seperate conda env in a rule in the Snakefile and we can inculde the packages we want to use while configurating the new conda environment (command: snakemake –use-env).

1.4.2 useful links.

`https://snakemake.readthedocs.io/en/stable/` `https://snakemake.readthedocs.io/en/stable/tutorial/tutorial.html#tutorial` `https://www.youtube.com/watch?v=NNPBDOBHlxo&ab_channel=EdinburghGenomicsTraining` `https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html`

1.5 GUIX.

1.5.1 intro.

"GNU GUIX is a general-purpose package manager that implements the functional package management paradigm pioneered by Nix" (*). it allows the creation and the deployment of isolated software environments, operating systems and the maintaining of a large range of packages.

- By definition, this software is a great addition to our arsenal in the subject of reproducible research. And among the many uses it has, tools like time-machine, for instance, allow the execution of old programmes. By finding the right version of the commit and installing these versions of dependencies, we can get back to a similar state to the old one and so, by executing the programme, we can get the same results.

1.5.2 useful links.

`https://link.springer.com/content/pdf/10.1007%2F978-3-319-27308-2_47.pdf` (*) `https://guix.gnu.org/manual/en/html_node/Binary-Installation.html` `https://guix.gnu.org/en/manual/en/guix.html`

1.6 Docker.

1.6.1 intro.

Docker is a software for developing, shipping, and running applications. It allows the user to create seperate containers for their applications which seperates them from the infrastructure of the computer of the user and facilitates their use from other individuals.

- Having a container which isolates the app from the environment and where the developper can state the packages and the versions that they want to use makes it possible for them to reproduce the same results. All they need to do is to create an image using a Dockerfile and to store it so as to be used to create a container whenever needed.

1.6.2 useful links.

`https://reproducible-analysis-workshop.readthedocs.io/en/latest/8.Intro-Docker.html`

2 The tutorials that were done in order get a better grasp of the tool.

2.1 git-annex.

2.1.1 useful commands.

- creating a repository:

```
git init
git annex init
```

- adding files:

```
git annex add .
git commit -a -m added
```

- adding a remote (usb drive):

```
sudo mount /media/usb
cd /media/usb
git clone ~/annex
cd annex
git annex init "portable USB drive"
git remote add laptop ~/annex
cd ~/annex
git remote add usbdrive /media/usb/annex
```

- getting file content:

```
cd /media/usb/annex
git annex sync laptop
git annex get .
```

- syncing:

```
cd /media/usb/annex
git annex sync
```

- removing files:

```
git annex drop iso/debian.iso
```

2.1.2 useful links.

https://git-annex.branchable.com/walkthrough/ https://git-annex.branchable.com/special_remotes/external/

2.2 Zenodo.

2.2.1 Uploading through the API.

1. with cURL:
    - LINK: https://felipecrp.github.io/2021/01/01/uploading-to-zenodo-through-api.html
    - PROCESS:
        - We tested to see if the API is accessible by using a GET request and there wasn't an error message.
        - We then sent a POST request to the API to request the creation of a new deposit which we will be using to upload our files later on. We then receive a JSON message confirming the creation of the deposit and extra information (date, title, owner, ..).
        - We then sent a PUT request to upload the files in the deposit using the bucket link that was sent in the JSON message when we created the deposit.
        - Once we finish uploading the files, we can check the deposit to see if they have been uploaded. I did the test twice and uploaded simple .txt files (zenodotest.txt & zenodotest2.txt) and they are accessible via this URL: https://zenodo.org/api/files/4aefd393-ed38-489c-bc8c-2413d9cb160f/zenodotest2.txt?access_token=WgYPkomVp1HpJniDmwS2ylFBhwsNpntxFzKqo02HGij94nVFlO0tAefbc
2. with Python:
    - LINK: https://developers.zenodo.org/?python#quickstart-upload
    - PROCESS:
        - The process is similar to the last one. We first import the package requests which we will be using to send the HTTP requests to the browser, and then we use a GET request to access the deposit (while giving an authentication key with the right access).

- We then send a POST request to create a new deposit and we get in return a message containing information about this new deposit (id, links, ..).
- Now, we can finally create new uploads. We first fetch the bucket URL by using this command which retrieves the item corresponding to the key "bucket" in the "links" dictionary.

```
bucket_url = r.json()["links"]["bucket"]
```

We use the method PUT to upload out file into the deposit. I did that using the new API.

2.2.2 Uploading through the website:

I also tested how to publish an article and save it in the database. To do this, I simply used the sandbox and uploaded a report I worked on with an other student last year.

2.3 Snakemake.

I created a simple Snakefile in which I wrote a couple rules with shell commands and tried to compile some old projects in order to test this tool.

- LINK: `https://www.youtube.com/watch?v=hPrXcUUp70Y&ab_channel=NCSAatIllinois`

2.4 Docker.

In the case of Docker, I've alredy used this tool during the project of the semester 8 and so I have a grasp on its basic usage. I've already manipulated existing images and deployed containers either separately or in groups using docker-compose. In the case of creating new images, I've tried creating some using simple Dockerfiles.

3 An introduction to Reproducible Research.

It is the practice of having the flow of research documented in all of its steps in order to get the same result even if it's retested by other researchers or in the future. It relies on many components of the scientific study such as: having accessible data, detailed research and analysis, reproducible workflows and evironments, ..

- The main features that need to be present for a scientific research to be reproductibility-friendly:
  - WF specification: connected tools steps of the analysis.
  - WF execution: provenence modules, data management, ..
  - WF environment: companion tools like Virtual machines, containers, ..

4 A look into GUIX and reproducible software environments.

4.1 Link:

`https://link.springer.com/content/pdf/10.1007%2F978-3-319-27308-2_47.pdf` Paper: *Reproducible and User-Controlled Software Environments in HPC with Guix* by Ludovic Courtès and Ricardo Wurmus.

4.2 Notes.

- "GNU GUIX is a general-purpose package manager that implements the

functional package management paradigm pioneered by Nix".

- It's important to handle the software environments when trying to work on reproducible research because the work that is being done mainly focuses on workflows, conventions, ... We need to have the same software environment to be able to reproduce the same results.
- A solution that was first given is to either write down the numbers of the dependencies (insufficient), or to save/download and reuse full system images (the images are large + it's difficult to combine with the software environment of each of the users). -> GUIX is the sollution to these problems.
- Reproducing the exact same software environment on a different HPC system could allow the users to assess the impact of the hardware on the software's performance + it would allow other researchers to reproduce the same experiment on their systems.
- Package managers like APT / RPM suffer from limitations:

  - Package binaries that every user installs, such as .deb files, are actually built on the package maintainer's machine, and details about the host may leak into the binary that is uploaded.
  - While it is possible for users to define their own variant of a package, it's still difficult to do. For instance, even if a user builds a custom .spec file, they can't always register it in the yumdb database (it's only allowed for the administrator) so it's difficult for the user to track down and register the complete graph of dependencies manually.
  - These tools implement an imperative and stateful package management model: imperative in the sense that it modifies the set of available packages in place (ex: switching to an alternative MPI implementation, or upgrading the OpenMP run-time library means that suddenly all the installed applications and libraries start using them). It is stateful because the system state at a given point in time is the result of the series of installation and upgrade operations that have been made over time, and there may be no way to reproduce the exact same state elsewhere.

- Package management is to be seen as functional paradigms where the results only depend on the inputs. So, rerunning a given build with the same input should result in bit-identical files (used by Nix and now by Guix). A tool that is used is chroot which allows them to run in a limited environment with a defined set of env variables, a dedicated user ID, separate name spaces for PIDs, inter-process communication (IPC), networking, ... This is so as to ensure that build can't end up using libraries it's not supposed to use -> this is what allows this process to be seen as pure functions with reproducible results.

- After each build, the created files are stored in the /gnu/store directory. each entry has a name that includes a hash of all the inputs of the build process that led to it (libs, compilers, ..). Therefore, we can fetch the diagram of dependencies easily in this case. After running 'guix build openmpi' for instance, we get as a return the directory name /gnu/store/xx-openmpi-1.8.1. and the daemon spawns the build process in an isolated environment (if the directory doesn't exist). For the normal user, the command 'guix package' is enough to install the packages without typing out the long names. In fact, this command creates a symbolic link to the selected /gnu/store item and the symbolic link is then stored in ~ /.guix-profile.