rapport stage

Oumaima Hajji

July 19, 2021

Contents

1 Plan (17.5 / 20 pages).

1.1 Focalisation (2.5 / 3 pages).

1.1.1 points (2 pages).

- En France, il y a un mouvement autour de la science ouverte (def v) `https://www.enseignementsup-recherche.gouv.fr/cid132529/`

le-plan-national-pour-la-science-ouverte-les-resultats-de-la-recherche-scientifi
html

- la science ouverte: "c'est la diffusion sans entrave des publications et des données de la recherche. Son objectif : faire sortir la recherche financée sur fonds publics du cadre confiné des bases de données fermées. Elle réduit les efforts dupliqués dans la collecte, la création, le transfert et la réutilisation du matériel scientifique. Elle augmente ainsi l'efficacité de la recherche."

- Le plan national pour la science ouverte: "rend obligatoire l'accès ouvert pour les publications et pour les données issues de recherches financées sur projets. Il met en place un Comité pour la science ouverte et soutient des initiatives majeures de structuration du paysage concernant les publications et les données."

- open science, open peer review, open protocol (mooc 1: focalisé sur la prise des notes, le document computationnal), open data. ("FAIR is not equal to Open: The 'A' in FAIR stands for 'Accessible under well defined conditions'. ... As such, while FAIR data does not need to be open, in order to comply with the condition of reusability, FAIR data are required to have a clear, preferably machine readable, license.": https://www.go-fair.org/resources/faq/ask-question-difference-fair-data-open-data

- FAIR data: https://www.go-fair.org/fair-principles/

- on sait qu'il faut faire ça (respecter les principes de FAIR data) mais comment le faire -> mooc (grâce à les outils donnés).

- Cette version du mooc se focalise sur la partie archivage data pour s'assurer de sa disponibilité, ...

1.1.2 liens.

- les principes de FAIR data: https://www.go-fair.org/fair-principles/

- open science: https://fr.wikipedia.org/wiki/Science_ouverte "La science ouverte (open science ou open research pour les anglophones) est un mouvement qui cherche à rendre la recherche scientifique et les données qu'elle produit accessibles à tous et dans tous les niveaux de la société."

- open peer review: https://openscience.pasteur.fr/2020/06/04/open-peer-review-un-mouvement-qui-prend-de-lampleur/
- open protocol.
- open data: https://fr.wikipedia.org/wiki/Donn%C3%A9es_ouvertes

1.2 Etat de l'art (3.5 / 4 pages).

1.2.1 historique / gros fichiers (1.5 / 2 pages).

1. points importants à inclure.
   - Dans la science moderne (science compitationnelle / data science) , on travaille sur des milliers de fichiers de données qui évoluent au cours du temps. Il faut donc garder une trace de chaque version de données et du code (*) exécuté à un instant pour pouvoir avoir une empreinte continue du développement de la recherche et donc pour comprendre et suivre tous les pas de l'expérience. C'est pour cette raison que l'on se sert de Git qui est le logiciel le plus utilisé pour la gestion de versions. changer l'ordre.
   - Puisque c'est difficile de gérer les fichiers de grandes tailles sur Git on va se servir de git-annex qui fait ça sans enregistrer le contenu des fichiers sur Git. En effet, blablabla.
   - Mais git-annex n'est pas le seul outil existant qui permet la gestion des fichiers des gros fichiers. En effet, Git a une extension qui permet de faire juste ça: Git lfs. (https://www.atlassian.com/git/tutorials/git-lfs). Mais il y a des problèmes avec cet outil qui gênent son utilisation: expliquer les deux problèmes..
   - Le choix entre les deux outils est donc fait: git-annex!
2. extra notes.
   (a) Importance of Version Controlling Data. The reality is that data is only rarely invariant. For example, throughout a scientific project, datasets can be extended with new data, adapted to new naming schemes, reorganised into different file hierarchies, updated with new data points or modified to fix any errors. If a dataset that is the basis for computing a scientific result changes without version control, reproducibility can be threatened: results may become invalid, or scripts that are based on file names that

change between versions can break. Therefore, version controlling data and other large files in a similar way to version controlling code or manuscripts can help ensure the reproducibility of a project and capture the provenance of results. (*) L'importance de 'Data Version Control' est évidente puisque dans le domaine de la recherche on gère des données de différents types et de grande taille et ces données peuvent changer et évoluer rapidement et donc au final on aura plusieurs version de chaque fichier et quand on compile les fichiers sources on peut avoir des différents résultats si on prend les mauvaises versions des fichiers. Mais on peut pas toujours tout mettre sur Git (surtout dans le cas des fichiers de grande taille qui évoluent exponentiellement). On gère ça alors with git-annex qui crée un directory annex où sont stockés les noms et les métadata des fichiers. Donc quand le dépôt est push sur Github, seuls les métadata sont transmises et alors les dépôts ne sont plus lourds. Les données peuvent être transmises sur des autres depôts (figshare, amazon, ...) et on peut facilement les récupérer avec une commande.

(b) Challenges in version controlling data. When you work, share, and collaborate on large, potentially binary files (such as many scientific data formats), you need to think about ways to version control this data with specialised tools. This is because most version control tools - such as Git - are not well suited to handle large binary data. As a Git repository stores every version of every file that is added to it, large files that undergo regular modifications can inflate the size of a project significantly. If others try to clone your repository or fetch/pull to update it locally, it will take longer to do this if it contains larger files that have been versioned and modified.

(c) Version controlling data with git-annex. The git-annex tool is a distributed system that can manage and share large files independent from a central service or server. git-annex manages all file content in a separate directory in the repository (.git/annex/objects, the so-called annex) and only places file names with some metadata into version control by Git. When a Git repository with an annex is pushed to a web-hosting service such as GitHub, the contents stored in the annex are not uploaded. Instead, they can be pushed to a storage system. If a repository with an annex is cloned, the clone will not contain the contents of all annexed files

by default, but display only file names. This makes the repository small, even if it tracks hundreds of gigabytes of data, and cloning fast, while file contents are stored in one or more free or commercial external storage solutions. `https://the-turing-way.netlify.app/reproducible-research/vcs/vcs-data.html`

1.2.2 archivage (2 pages).

1. points importants à inclure.

   - archivage: pourquoi (pérénité, mise à disposition à une communauté à définir) et ce que ça fourni (référencement pérène)
   - Exemples: zenodo, nakala, figshare, ...
   - Chainon manquant dans le workflow de gestion de données: l'archivage est une opération manuelle.
   - Ébauches de solution: github-zenodo, gitlab-zenodo
   - Ma proposition: proposer un remote zenodo
   - Maintenant que l'on sait comment gérer les fichiers il faut passer à l'autre étape importante dans ce procès qui est l'étape de l'archivage !!
   - Expliquer que le principe de special remotes est intéressant puisque la gestion est déjà faite par git-annex et qu'il faut juste choisir l'un des remotes qui nous est pratique et après on peut stocker les données dessus.
   - Expliquer pourquoi on n'a pas choisi les special remotes qui sont déjà implémenté par git-annex: `https://git-annex.branchable.com/special_remotes/`
   - Expliquer que la solution la plus évidente est de se servir de ce principe pour implémenter un special remote qui répond à nos attentes. Mais pour faire cela il y a plusieurs plateformes d'archivage: zenodo(cern), figshare, nakala, .. Il faut faire une comparaison de ces outils pour arriver à la conclusion que Zenodo est l'outil le plus intéressant pour nous.
   - Mais quand on s'appuie sur Zenodo pour faire de l'archivage, on remarque que il y a un shortcut entre Zenodo et github où les deux comptes de l'utilisateur sont connectés pour lui permettre d'upload ses projets github directement sur Zenodo et de les

archiver facilement. Pourquoi pas juste utiliser ce shortcut au lieu de passer par git et git-annex? Le problème c'est que ce mechanisme est personnalisé juste pour github et donc on ne peut pas faire cela avec des autres plateformes comme gitlab sans devoir passer par des biblio. Et même quand on fait ça, il y a toujours un problème avec le lien Zenodo-Gitlab car cette méthode permet juste d'upload des fichiers sur Zenodo en utilisant l'API et ne permet pas de faire plus que ça. Donc la solution la plus évidente est de commencer par git et de construire un chemin vers Zenodo.

- Parler de datalad et comme quoi il y a aussi un problem là car a seule solution d'archivage de ce type proposée par datalad est d'upload des archive zip sur figshare. donc on a implémenté le remote zenodo pour faire ça.

2. extra notes.

   (a) zenodo. 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.

   (b) datalad.

   - 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.
   - Converting an existing Git or git-annex repository into a DataLad dataset: $ datalad create -f
   - DataLad only cares (knows) about two things: Datasets and files. A DataLad dataset is a collection of files in folders. And

a file is the smallest unit any dataset can contain. Thus, a DataLad dataset has the same structure as any directory on your computer, and DataLad itself can be conceptualized as a content-management system that operates on the units of files.

- 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.

1.2.3 liens.

- comparing the archiving platforms: `https://espacechercheurs.enpc.fr/fr/donnees-recherche-aspects-techniques`
- git-annex vs lfs: `https://stackoverflow.com/questions/39337586/how-do-git-lfs-and-git-annex-differ`
- nakala: `https://documentation.huma-num.fr/nakala/#introduction-et-presentation`
- mendeley: `https://data.mendeley.com/archive-process`
- datalad.
- figshare.
- github to zenodo: we know that there is alink between the two which allows to archive a github repository on zenodo (this is especially useful in the case of when a researcher wants to cite the findings they have on github but they don't have the doi, so the next step to do is to use zenodo to archive the files that are on this repository and so we get at the end the doi number which allows us to cite): `https://guides.github.com/activities/citable-code/`
- l'archivage gitlab -> zenodo ne gère pas les fichiers dans git LFS: `https://gitlab.com/lnesi/icpp21/-/jobs/1430800588`
- library allowing to archive from gitlab to zenodo. It's still in beta stages and has just been developped since there isn't one that is already there like the github direct link: `https://pypi.org/project/gitlab2zenodo/`

https://gitlab.com/gitlab-org/gitlab/-/issues/25587 https://github.com/zenodo/zenodo/issues/1404 !! https://gitlab.com/gitlab-org/gitlab/-/issues/18763

1.3 Contributions (6.5 / 8 pages).

1.3.1 modele de donnees (1 / 1.5 page).

- Même si Zenodo paraît comme une la plateforme parfaite à utiliser comme un special remote de git-annex, il y a toujours un problème architecturel qui nous a gêné quand on a commencé la réfléxion de comment structurer notre remote. Quand on fait un upload Zenodo, puisque son infrastructure ne permet pas d'avoir des directory, alors le stockage se fait dans une liste des fichiers mal structurée.

- Un autre problème est aussi le fait que dans Zenodo quand on crée un nouveau upload, c'est toujours un dépôt où on va mettre tous nos fichiers. Donc, on peut choisir des différents modèles d'implémentation du remote Zenodo et pour chaque modèle, la fréquence de création des dépôts et les fichiers qui sont dedans changent. Si un dépôt est créé au moment de l'initialisation du remote, alors toutes les opérations qui viennent seront appliquées sur ce dépôt, et on aura ainsi un seul dépôt pour chaque remote créé. Mais on fait un autre choix, où l'initialisation du remote ne déclenche pas la création du dépôt, et au lieu faire cela après, alors c'est possible d'avoir plusieurs dépôts, et alors plusieurs identificateurs de dépôts pour un seul remote git-annex. Or, cela n'est pas nécéssaire puisque l'on peut choisir les fichiers à mettre dans un dépôt et ceux à laisser en local, et donc avoir un seul dépôt par directory (l'endroit où on initialise le remote) est suffisant puisqu'on peut l'utiliser quand on veut pour manipuler les fichiers que l'on veut sans avoir des problèmes de confusion. Si l'utilisateur veut créer un autre dépôt Zenodo avec un remote git-annex dans le même endroit que le premier remote, c'est toujours possible d'initialiser plusieurs remotes git-annex dans la même directory.

- On a aussi fait des tests pour voir s'il y a des limites imposées sur le nombre de fichiers possibles à mattre dans un dépôt mais il n'y avait pas des problèmes avec ces tests et donc c'est possible d'uplad des milliers de fichiers mais la taille du dépôt ne doit pas atteindre 50GB. C'est la seule limite imposée par Zenodo. On les a contacté pour s'assurer de cette hypothèse et on a eu une réponse positive.

1.3.2 Implem de remote zenodo (3.5 / 4.5 pages)

L'implémentation du remote Zenodo s'est faite en plusieurs étapes:

1. api rest (0.5 page). `new_version` ou `publish`

   un *record* ou un truc important

   nouveau dépot

   publish

   ou alors $\mathbf{new_{record}}$ et **publish**

   la première partie du procès est de comprendre comment fonctionne l'API Zenodo et de tester les fonctionnalités possibles de cette API. Il fallait faire des tests pour chacune des requêtes HTTP pour tester les opérations possibles Zenodo. Les opérations les plus importantes comme la création du dépôt, l'envoi des fichiers sur le dépôt, la suppression, et l'obtention des informations sur le dépôt et les fichiers stockés dedans. Il y a aussi des autres opérations pour publier le dépôt pour archiver les données (une fois un dépôt est publié, il devient un record qui a un doi et que l'on peut citer alors quand on veut, on ne peut donc pas supprimer un record une fois publié. C'est comme ça que l'on garantie son existence et on le rend accessible et trouvable depuis le doi). On peut aussi créer des nouvelles versions d'un record avec une simple requête post vers l'API, et c'est grâce à cette opération que l'on veut faire évoluer ses données en gardant des traces (chaque version publiée d'un record a son doi, et puisque l'on peut juste avoir une seule nouvelle version à la fois, on peut s'assurer du développement des données).

2. biblio python qui implemente deja le protocol (0.5 page).

   - Afin d'implémenter un remote git-annex[1] il faut d'abord être sûr que son. [**?**]

   program implémente bien le protocole 'external special remote' de git-annex qui fait le lien entre git-annex et son remote externe. Les deux bout de la communication échangent des requêtes et des réponses durant la période de l'exécution du programme (celui qui implémente le remote X: git-annex-remote-X). Pour ne pas avoir des soucis de

[1] Le protocole `git-annex` `http://git-annex.org/protocol`

confusion des intéractions, à chaque fois l'une des deux parties prend l'initiative en n'envoyant que des requêtes et l'autre partie répond alors avec des reponses à ces requêtes.

- On utilise la bibliothèque **AnnexRemote** de python qui implémente la totalité du protocole et respecte toutes ses spécifications. Il faut donc juste importer cette les modèles de cette bibiliothèque que l'on veut utiliser dans notre programme. On définit alors une classe pour notre remote qui extend la classe SpecialRemote de la bibliothèque. Ensuite, il nous reste à implémenter les fonctions que l'on va utiliser pour déposer les données sur le remote et les manipuler.

3. operations principales (1.5 / 2 pages). Chaque remote Zenodo doit être capable d'exécuter des opérations principales qui servent à envoyer les fichiers sur le remote, les manipuler, et les récupérer en local. Tout cela se fait avec les fonctions du programme principal git-annex-remote-zenodo avec creation d'un depot, upload, check, remove, get.

4. tests (0.5 page). avec les exceptions du protocol pour s'assurer que les pb de l'api passent bien à git-annex et qu'il y a une coherence en les deux . debug mode

1.3.3 archiver avec disableremote (1 page)

Quand la première partie de la gestion des données finit, et on stocke tous les fichiers qui nous intéressent dans le remote, il faut maintenant passer à la deuxième partie de l'archivage qui se fait indépendamment de la première, et où on finalise son dépôt avec toutes les méta-données nécessaires avant de le publier.

- les options pour par exemple publier le fichier .json ou

1. newversion.

1.3.4 restaurer une archive (1 page)

1.4 Evaluation (1 page).

rédaction d'un tutorial pour tester et comprendre la totalité du projet (0.5 page).

- walkthrough, intégration au MOOC 2 en cours

1.5 Méthodologie et Compétences développées (3 pages).

1.5.1 comppétences développées (1 page).

- Bilan des connaissances et expériences acquises ou approfondies au

cours de ce stage.

- Description sur une page d'une ou deux compétences développées pendant le stage. Cela peut être des compétences du métier d'ingénieur en informatique ou aussi des compétences transversales au métier d'ingénieur (voir les deux fichiers excel attachés).
- MOOC 1 (prise des notes! rendre mon travail compréhensible par quelqu'un d'autre)

1.5.2 méthodologie (2 page).

1. documentaion de l'ensemble du processus (0.5 page).
    - ex: parler du journal (application directe des éléments de la RR).
        - parler des tutos faits au début / des petits programmes écrits pour tester les outils (API Zenodo, tuto git-annex, tuto snakemake?)
    - tests
        - doc
2. Gestion du projet: Description de la gestion de votre projet (cycle de vie, structuration en taches, durées estimées et réelles, gestion de risques ...)
    - reunions hebdomadaires, agile,
    - gantt chart, a posteriori, planification légère
    - identifier mes propositions personnelles (que ce soit des trouvailles ou des directions à ne pas emprunter)

1.6 Conclusion (1 - 2 page).

ce j'ai pas pu faire: nakala - datalad (submodules) voir comment ça peut s'integrer avec zenodo (ex de figshare par opposition) - (snakemake <-> git-annex) : pb: où integrer les commandes git annex simples (ex get) dans un workflow snakemake.

1.7 Bibliographie.

liens à mettre après.