Projet Reprohackathon

Lucas Bernigaud, Charlotte Dubec, Guillaume Studer, Elodie Yan

Table des matières

1 Introduction

1.1 Reproductibilité

Depuis une dizaine d'années, on observe une sensibilité particulière du monde académique à reproduire des expériences pour retrouver les résultats qui ont été obtenus par l'expérience d'origine. Or, plusieurs études ont été menées montrant qu'il est souvent difficile, si ce n'est impossible, de savoir comment ont été obtenus les résultats décrits dans une publication.

Par exemple, Nekrutenko et Taylor, dans une étude publiée en 2012 dans *Nature Genetics*, se sont intéressés à 50 publications de 2011 utilisant le Transformateur de Burrows-Wheeler pour mapper des séquences Illumina. Sur ces 50 publications, seules 7 donnaient tous les détails nécessaires pour reproduire l'expérience. 31 ne donnaient aucune information sur les versions d'outils utilisés, les paramètres utilisés et les références génomiques exactes des séquences. Cela signifie donc que dans 62% des cas, il est impossible de savoir comment les résultats ont été obtenus.

Dans les années 2010, de telles publications ont renforcé ce que l'on appelle la "Crise de la Reproductibilité", ou autrement dit la crise méthodologique touchant de nombreux domaines scientifiques et décrivant la quasi-impossibilité de reproduire des résultats obtenus par d'autres (voire par soi-même !).

Il existe trois grands types de reproductibilité, qui expliquent les nombreuses causes possible de l'irreproductibilité.

- La **reproductibilité empirique**, qui concerne les sciences expérimentales et qui est classiquement difficile à obtenir puisque ces dernières reposent sur des phénomènes et sujets biologiques avec une forte variabilité (phénomènes biologiques aléatoires, différents sujets d'étude à différentes échelles). Les techniques utilisées en sciences expérimentales sont également sujettes à des variations selon les opérateur.ice.s (préparation des échantillons, mesures...) et les conditions expérimentales "externes" qui ne peuvent pas toujours être contrôlées avec précision (température de la pièce qui peut jouer sur certaines expériences par exemple).
- La **reproductibilité statistique** concerne les choix de tests statistiques, les paramètres des modèles, les valeurs seuils...
- La **reproductibilité computationnelle** concerne quant à elle les informations sur les codes, logiciels, systèmes, machines... Avec une simple mise à jour de logiciel, il est possible d'obtenir des résultats tout à fait différents pour un même code.

Pour garantir un maximum de reproductibilité, il est important de garder trace de tous les paramètres et conditions utilisés pour réaliser l'expérience. Par exemple, pour la reproductibilité empirique, il peut être pertinent de tenir un "cahier de laboratoire" contenant le plus de détails possibles sur les conditions de réalisation de chaque expérience. Pour la reproductibilité statistique, il faudrait noter les tests, paramètres etc, et pour la reproductibilité computationnelle, noter les versions des logiciels et systèmes. Enfin, le tout est de rendre disponibles et accessibles toutes ces informations.

Il existe trois niveaux de reproductibilité (Cohen-Boulakia et al., FGCS, 2017) :

- **La répétition** : il s'agit de reproduire l'expérience de manière exactement identique, autrement dit, on suit le même protocole à la lettre. On s'attend donc à avoir exactement les mêmes résultats. Ce niveau de reproductibilité est notamment utile pour les relecteur.ice.s de publications.
- **La réplication** : on reproduit le protocole mais avec des outils légèrement différents (les versions des logiciels par exemple). En général, on reproduit l'expérience avec des outils améliorés. Les résultats ne sont donc pas identiques en tout point mais restent très similaires et les interprétations restent les mêmes. Ce niveau permet de tester la robustesse du protocole.
- **La reproduction** : on reproduit l'expérience avec un protocole différent ce qui donne des résultats différents mais les interprétations restent les mêmes.

1.2 Objectifs du projet

Dans ce projet centré sur la reproductibilité, nous allons reproduire une partie des expériences présentées dans deux publications :

- Harbour JW, Roberson ED, Anbunathan H, Onken MD, Worley LA, Bowcock AM. Recurrent mutations at codon 625 of the splicing factor SF3B1 in uveal melanoma. *Nat Genet.* 2013;45(2):133-135. doi:10.1038/ng.2523
- Furney SJ, Pedersen M, Gentien D, et al. SF3B1 mutations are associated with alternative splicing in uveal melanoma. *Cancer Discov.* 2013;3(10):1122-1129. doi:10.1158/2159-8290.CD-13-0330

La seconde publication reprend les expériences de la première ; il s'agit de séquençage aléatoire du transcriptome entier sur des échantillons d'ARN de patients atteints de mélanome uvéal, dont certains présentent des mutations sur le gène SF3B1.

Nous partirons du même jeu de données afin de mettre en évidence l'expression différentielle des gènes selon qu'ils appartiennent à un sujet avec la mutation SFB3 ou non. Pour cela nous allons travailler sur 8 échantillons de génome de patients avec une forme de mélanome. Ces patients sont séparés en deux catégories : les wild-types (au nombre de 5) et ceux présentant une mutation sur sur le facteur d'épissage alternatif SF3B1 (au nombre de 3).

1.3 Choix des outils

Pour que notre travail soit le plus reproductible possible, nous avons choisi d'utiliser les outils Snakemake, Singularity et Github. Snakemake est un système de gestion de workflow qui utilise le langage Python avec lequel nous étions déjà familier.e.s, ce qui explique en partie pourquoi nous l'avons choisi par rapport à Nextflow.

Concernant la documentation utilisée pour comprendre et exploiter ces outils nous avons utilisé les documents explicatifs fournis par les encadrants et avons en plus fait des recherches qui sont résumées ci-dessous.

Pour l'utilisation de Snakemake et Singularity nous avons utilisé les manuels d'installation et d'utilisation, que nous avons résumés en début de projet dans des fiches pour mieux nous familiariser avec ces outils, étant donné que nous les utilisions pour la première fois.
Pour les différents containers que nous avons utilisés, en plus du document récapitulant les commandes shell à écrire, nous avons suivi les instructions données sur les Github ou Dockerhub contenant les containers.

2 Organisation du travail

2.1 Répartition des tâches et organisation

Concernant le codage du pipeline (qui consiste en un seul fichier appelé Snakefile, détaillé dans les sections suivantes), nous avons décidé de travailler à chaque fois en binôme (les binômes n'étant pas toujours les mêmes, en fonction des disponibilités et envies de chacun), un binôme s'occupait donc d'une des étapes ou sous-étapes, demandant de l'aide au reste du groupe lorsqu'ils/elles étaient bloqué.e.s. A la fin de l'étape, le binôme présentait et expliquait au reste du groupe le code écrit pour faire d'éventuelles modifications ensemble. Pour cela, nous organisions 2 ou 3 réunions hebdomadaires en appel visio. Au sein du binôme, le travail se faisait souvent grâce à un appel visio et un partage d'écran.

Chaque étape n'étant pas forcément écrite par les mêmes personnes nous mettions les morceaux de code correspondant à une étape ou sous-étape sur notre repository Github. Nous avons assemblé le tout dans un "Snakefile global" (que nous avons commencé début novembre), complété ensuite au fil du projet.

Pour préparer les rendez-vous hebdomadaires avec les encadrants , l'un.e de nous quatre se chargeait de la trame du diaporama. Celle-ci comportait notamment les scripts de nos codes et les éventuelles erreurs renvoyées. Chacun.e commentait ensuite les diapositives avec ses questions éventuelles. Lors des rendez-vous, une personne prenait la main sur la présentation, et une autre personne notait les remarques et conseils de l'enseignant. Celle-ci rédigeait ensuite un compte-rendu pour garder trace de tout ce qui avait été dit, et surtout des réponses précises à nos questions - cela a été d'une grande aide pour nous "débloquer" sur les scripts. Tout le monde pouvait évidemment intervenir pour poser des questions complémentaires ou demander des clarifications sur certains points. Après la réunion, nous faisions un court débriefing sur les commentaires donnés et organisions notre travail sur la semaine suivante (cahier des charges et répartition au sein du groupe).

2.2 Outil de partage des codes : GitHub

Afin de pouvoir facilement partager nos codes et ainsi avancer en parallèle, sans avoir à se transmettre les codes "manuellement" chaque fois qu'un membre de l'équipe avançait, nous avons utilisé l'outil d'hébergement et de gestion de développement de codes et logiciels GitHub, qui se base sur la gestion du versionning de Git et permet la mise en commun facile des codes développés par différent.e.s collaborateur.ice.s, tout en mémorisant les anciennes versions des codes.

Cet outil nous a permis d'essayer différents codes en parallèle, de reprendre la base d'un code écrit par un.e autre membre et également de pouvoir facilement mutualiser les scripts, et les corrections apportées, utiles pour la construction du pipeline complet.

Ainsi, les codes que nous avons utilisés sont accessibles à l'adresse suivante :
https://github.com/elodieyan/hackathon_groupe6

3 Résultats

3.1 ACP

Nous avons réalisé une Analyse en Composantes Principales (ACP) sur les données d'expression des gènes (Figure 1).

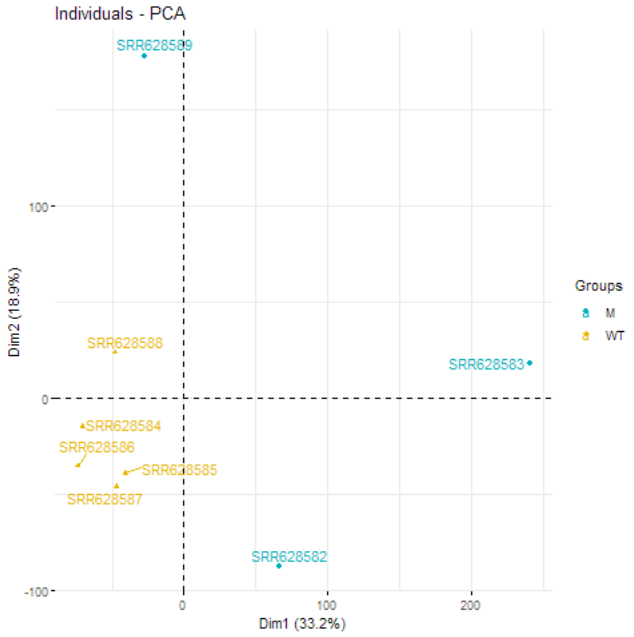

Figure 1: Résultats de l'ACP : graphe des individus

Ce graphique nous montre bien que dans l'espace des gènes, les échantillons WT sont groupés et forment un ensemble cohérents, tandis que les échantillons mutés sont à l'écart de ce groupe. On peut déjà supposer à partir de cette observation que les organismes mutés ont une expression génétique différente des organismes WT.

3.2 Expression différencielle des gènes

Nous avons utilisé la fonction Enhanced Volcano sur R pour construire un graphique de nos données (figure 2). Nous avions initialement plus de 60 000 données, mais en enlevant celles avec une valeur "NA" (correspondant aux gènes qui ne présentaient aucun reads), nous avons finalement 20 861 variables.

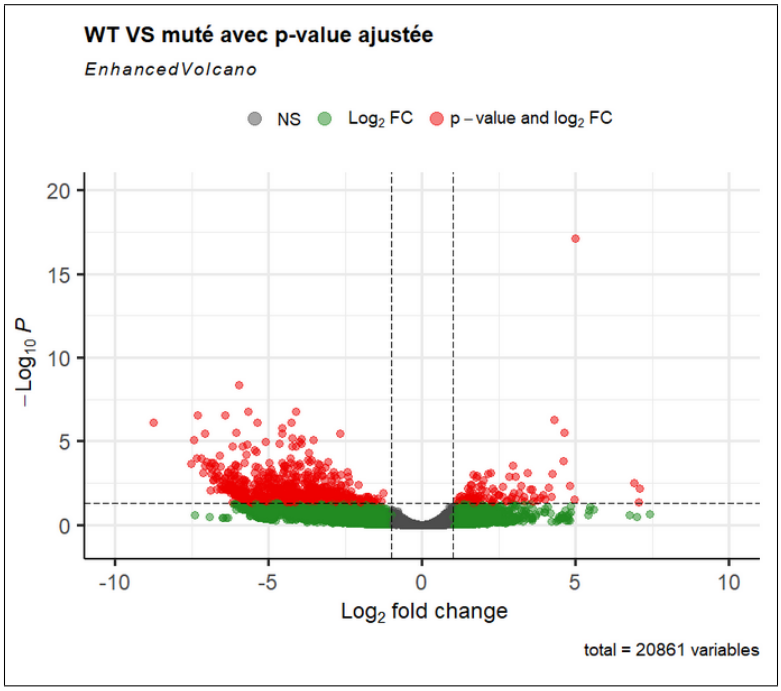

Figure 2: Volcano plot

Sur la figure 2, les points en rouge sont ceux correspondant à une p-value inférieure à 0.05 (et donc un résultat significatif statistiquement). Tous les points ayant une p-value ajustée inférieure à 0.05 ont également un log2(fold change) inférieur à -1 (expression au moins 2 fois inférieure chez les patients mutés) ou supérieur à 1 (expression au moins 2 fois supérieure chez les patients mutés). Plus on se déplace vers la gauche du graphique, plus les gènes sont sous-exprimés chez les mutés et plus on se déplace à droite, plus ils sont sur-exprimés chez les mutés.

On remarque également que certains gènes ont un log2(foldchange) allant jusqu'à +7 ce qui signifie que certains gènes sont 2^7 fois plus exprimés chez les mutés que chez les wild- types. Nous avons donc de très fortes différences d'expression entre les deux conditions.

En faisant une sélection dans le tableau des résultas de l'analyse statistique, 860 gènes sont sélectionnés comme ayant une epxression significativement différente entre les sujets mutés et wild-types. Cette valeur est assez éloignée de celles trouvées dans les deux articles. En effet dans l'article de *Furney et al.*, 325 gènes sont ressortis comme différentiellement exprimés, et seulement 10 dans l'article de *Harbour et al.* (ils se disent d'ailleurs surpris de ce résultat).
Dans l'article de *Furney et al.*, sur les 325 gènes différentiellement exprimés, 46 (soit 15%) sont sur-exprimés chez les mutants et 279 (soit 85 %) sous-exprimés. Nous retrouvons dans nos résultats à peu près les mêmes proportions avec 11 % (97 gènes) sur-exprimés et 89% sous-exprimés. Idéalement, il aurait fallu comparer si les gènes différentiellement exprimés étaient en partie les mêmes dans l'article et dans nos résultats mais nous n'en avons pas eu le temps.

3.3 Reproductibilité de notre travail

L'objectif principal de ce projet était de construire un travail reproductible. Nous avons vu dans l'introduction que la reproductibilité était divisée en trois grands types : la reproductibilité empirique, la reproductibilité statistique et la reproductibilité computationnelle. La reproductibilté empirique n'est pas du ressort de ce projet comme nous ne faisons pas d'expériences biologiques. Pour la reproductibilité statistique, les tests et seuils sont implémentés dans notre code R et ne sont pas modifiés lors de l'utilisation de notre code. Pour la reproductibilité computationelle cela relève de trois points :

- Cet aspect de la reproductibilité est très influencé par les éventuelles mises à jour de logiciels : comme spécifié dans l'introduction, une simple mise à jour peut engendrer des résultats tout à fait différents. L'utilisation de containers permet de pallier cela. En effet, une fois construites, les fonctions dans un container ne sont plus changées, sauf si mise à jour il y a, auquel cas il faudrait changer les appels de containers dans notre pipeline pour avoir les nouvelles versions. Sans ces changements, nos résultats sont reproductibles sur cet aspect-là.

- La construction d'un workflow permet à toutes les étapes de tourner toujours de la même manière, dans le même ordre et avec les mêmes entrées et sorties. Cela garantit une reproductibilité car l'utilisateur.ice n'intervient pas dans le déroulement des étapes.

- Lors de la construction de notre pipeline, nous avons annoté notre code et écrit une "notice d'utilisation" pour que l'utilisateur.ice n'ait pas de choix à faire et que la procédure se déroule toujours de la même façon. Ces annotations précises permettent également de garantir la reproductibilité de notre projet.

4 Matériels et Méthodes

4.1 Outils utilisés et leurs options

Dans cette section nous détaillerons les outils mentionnés dans la section 1.3 (Choix des outils) en expliquant leur fonctionnement général et en explicitant certaines options que nous avons utilisées pour le fonctionnement de notre pipeline.

4.1.1 Singularity

Singularity est un outil permettant la création et l'utilisation d'environnement virtuels encapsulés. Ces environnements sont appellés des containers. L'objectif d'un tel outil est de s'affranchir de la variabilité générée par les différents systèmes d'exploitations et versions de logiciels de chaque machine. En utilisant Singularity, on peut faire tourner une étape de pipeline ou bien un pipeline

complet avec une configuration système pré-définie dans un ou plusieurs containers. Ainsi, on installe les configurations, outils et packages nécessaires à la réalisation d'une étape d'analyse dans un unique container qui peut être partagé via des plateformes web comme DockerHub. Toute personne souhaitant réaliser une expérience avec les configurations identiques peut choisir de faire tourner son analyse à l'intérieur même du container récupéré sur une plateforme en s'affranchissant des configurations de sa propre machine. Cet outil représente l'élément principal de reproductibilité en ce qui concerne le niveau de reproductibilité « Replicabilité » explicité dans la section 1.1. Singularity ayant également plusieurs versions il est important de préciser que nous avons utilisé, dans le cadre de ce projet, la version 3.6.3 de Singularity. Il aurait également été possible d'utiliser l'outil Docker qui permet d'effectuer des opérations similaires.

4.1.2 Snakemake

Comme mentionné précédemment Snakemake est un outil de gestion de workflow. C'est l'outil que nous utilisons pour expliciter et réaliser les étapes d'analyses selon un ordre précis et contrôlé. Il constitue un protocole d'analyse et nous permet donc de respecter le niveau de reproductiblité "Reproduction" (section 1.1). L'utilisation basique de Snakemake consiste en l'écriture d'un unique script que l'on nomme un SnakeFile. Snakemake comprend comme langage principal Python, il est donc possible d'écrire tel quel des commandes python à l'intérieur du Snakefile. L'objet principal de snakemake est une "règle" (`rule`). C'est une organisation d'informations spécifique à Snakemake qui permet d'écrire une étape d'analyse. Cet objet comprend comme éléments nécessaires :

- une section `output` qui précise les fichiers que nous souhaitons obtenir à la fin de cette étape d'analyse (fichiers de sortie),
- une section "commande" qui se traduit par trois mots-clefs si on utilise Singularity (`run`, `shell` ou `script`) donnant les commandes à réaliser pour obtenir les fichiers d'analyses.

Sans être obligatoire, les règles contiennent généralement une section préalable `input` très importante qui permet d'indiquer les fichiers utilisés par la règle pour obtenir les fichiers de sortie. C'est nottament cette section qui nous permet d'éxecuter chaque étape dans le bon ordre.

Snakemake utilise, via le *Snakefile*, les fichiers comme éléments principaux de gestion du pipeline. Ceux-ci lui permettent de contrôler la bonne réalisation des étapes et de les lier. Un Snakefile contient obligatoirement comme première règle une "règle cible" (communément appelée `rule all`) qui explicite dans ses inputs les outputs finaux du Snakefile, c'est-à-dire les fichiers que l'on veut obtenir à la fin du protocole. Snakemake arrive à faire se succéder toute les étapes en les liant selon l'enchaînement des fichiers input et outputs. Pour cela, Snakemake repère l'étape qui permet d'obtenir les fichiers outputs finaux (en utilisant la règle cible) et analyse les inputs nécessaires à la réalisation de cette étape, puis fait de même pour l'étape qui génère ces inputs

jusqu'à identifier la ou les premières étapes du pipeline et entamer sa réalisation. Un avantage considérable de Snakemake est la capacité à parralléliser les branches indépendantes du Snakefile. En effet, certaines règles nécessitent plusieurs inputs provenant de règles différentes. Si ces règles sont indépendantes, alors elles sont sur des branches différentes et Snakemake va automatiquement parralléliser l'avancée des deux branches dont l'avancée se fera simultanément.

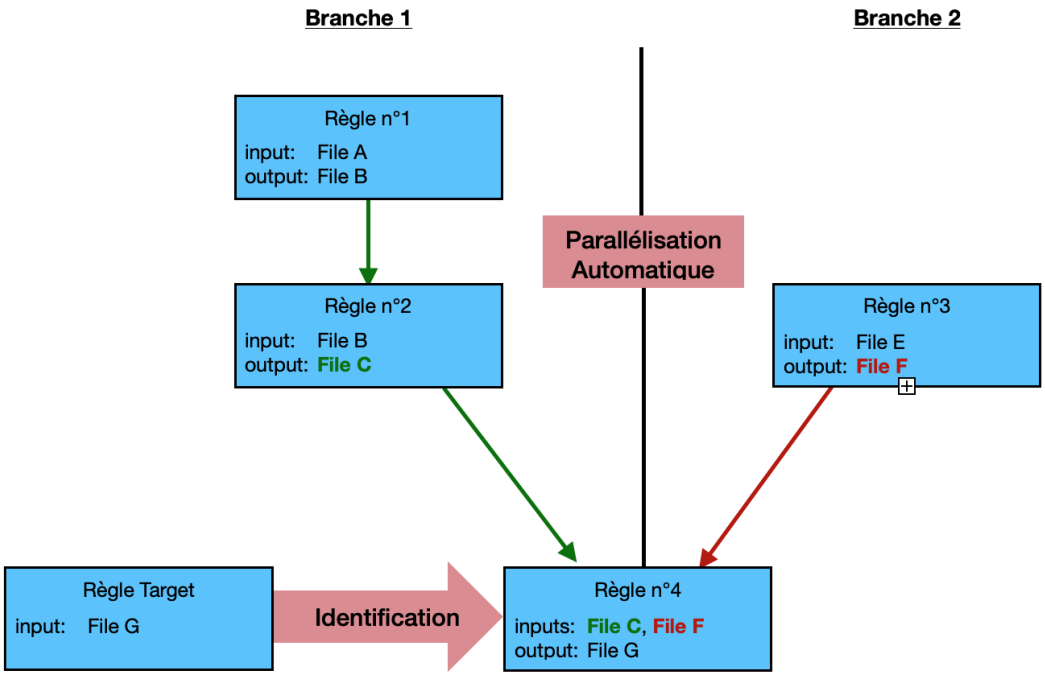

Figure 3: Parallélisation des étapes du pipeline

Snakemake admet un grand nombre d'options qui permettent d'optimiser le pipeline. Dans notre projet nous avons essentiellement utilisé celles qui nous permettent de gérer la capacité du processeur en répartissant les coeurs, et donc de parralléliser les tâches au sein d'une même étape ou pour différentes étapes ainsi que celle qui nous permet de réaliser nos commandes au sein d'un container en utilisant Singularity. Ces options se traduisent par des sections spécifiques dans les règles du Snakefile:

- La section `threads` permet de spécifier le nombre de coeurs que l'on souhaite allouer à la réalisation de cette étape. Voici un exemple simple : si nous lançons notre pipeline complet en allouant N coeurs et qu'une étape du pipeline demande n threads alors snakemake comprendra automatiquement qu'on peut réaliser t = N/n (arrondi à l'inférieur) fois cette même étape en parallèle (si besoin de traiter différents fichiers de même type par exemple). En réalité, Snakemake ne va pas forcément réaliser uniquement cette étape si d'autres étapes indépendantes du pipeline peuvent être réalisées en même temps, il effectue donc une répartition automatique et optimale des coeurs entre les différentes étapes.

- La section `singularity` permet de préciser le container dans lequel on souhaite réaliser la commande de cette étape. On fournit par exemple dans cette section l'adresse web du container souhaité sur une plateforme de partage de container (comme DockerHub).

Enfin, Snakemake met à disposition deux syntaxes : les wildcards et la fonction `expand`, que nous avons utilisées pour permettre respectivement la parrallélisation des tâches et l'allégement du script.

- **Les wildcards** : dans le cas ou nous aurions la même étape à réaliser sur un grand nombre de fichiers stockés dans le même répertoire, on peut utiliser la syntaxe suivante dans les sections `output` et `input` : `"repertory/{file"}`. Snakemake va alors comprendre que pour chaque fichier contenu dans ce repertoire il faut réaliser l'étape définie dans la règle et stocker ensuite les outputs de la même manière. Cet outil est intéressant car il précise à snakemake que cette étape n'a pas besoin de tous les fichiers du répertoire pour démarrer l'étape mais bien que chaque fichier doit être utilisé indépendamment et donner un output. C'est cette syntaxe qui permet, en fonction du nombre de coeurs disponibles et du nombre de coeurs nécessaires à cette étape, de parralléliser la tâche. De cette manière, plusieurs fichiers du répertoire d'inputs peuvent être utilisés indépendamment pour cette étape.

- **La fonction `expand`** utilise la structure de données `list` de Python. Dans les sections `input` et/ou `output`, on peut faire appel à cette fonction si on a besoin de beaucoup de fichiers inputs. Par exemple, si on traite plusieurs échantillons en même temps, on peut définir une liste Python au début du SnakeFile contenant le nom de chaque fichier et écrire dans les sections `input` ou `output` : `expand("repertory/input_file.ech", input_file = Notre_Liste)`, ce qui est équivalent à écrire une ligne d'input pour chaque fichier d'échantillon.

Ci-après un exemple d'une règle avec une syntaxe similaire à ce que nous avons utilisé dans l'écriture de notre Snakefile.

```
rule statistic:

        input:
                “repertory_1{sample}.txt“
        output:
                “repertory_2/{sample}.csv“

        threads: 2

        singularity:
                “docker://web_path_to_docker_image/statistical_tool/v2.3“

        shell:
                “““
                statistical_tool_1 {input} > {output} —using-threads {threads}
                “““
```

Figure 4: Exemple de règle comportant des options

Snakemake fournit un grand nombre d'autres d'options que nous n'avons pas utilisées lors de ce projet mais qui sont disponibles dans la documentation: https://snakemake.readthedocs.io/en/stable/.

4.2 Données utilisées

Nous avons d'abord téléchargé les 8 séquences que nous allions analyser dans le projet, via le portail du NCBI (https://www.ncbi.nlm.nih.gov/Traces/study/?acc=SRP017413&o=acc_s%3Aa). Ces 8 séquences sont nommées "SRR628582", "SRR628583", "SRR628584", "SRR628585", "SRR628586", "SRR628587", "SRR628588" et "SRR628589". Nous les avons obtenues en format SRA. Nous avons ensuite téléchargé les séquences ADN du génome humain (chromosomes) sur le site Ensembl (https://www.ensembl.org/index.html), en format FASTA. Sur ce même site, nous avons pu télécharger les annotations du génome humain, en format GTF. Nous donnons ici le lien des sites mais le téléchargment de ces données est inclus dans notre pipeline. Il n'est donc pas nécessaire de les télécharger manuellement.

4.3 Généralités sur notre pipeline

4.3.1 Construction du pipeline

Pour rendre plus facile le travail en parallèle sur les différentes étapes du pipeline, nous avons initialement réfléchi à des règles, correspondant à des étapes, exécutées seules dans des scripts Snakemake. Il a alors fallu les assembler et rendre exécutables ces règles indépendantes dans un seul fichier à exécuter avec Snakemake en respectant l'ordre d'enchainement des étapes. Nous avons donc suivi la logique de Snakemake, qui est de regarder l'input de la "règle cible" (`rule`

`all`) qui correspond au fichier final de sortie du pipeline, puis de remonter dans les autres règles les outputs et inputs successifs afin d'établir un ordre d'exécution des règles.

Pour cela, nous avons donc lié les règles entre elles, enchaînant les outputs des premières règles en inputs des règles à exécuter ensuite. Nous avons envisagé 2 formes de pipeline :

- Une forme **linéaire**, Snakemake exécutant une à une les règles avec un enchaînement parfois absurde des règles mais pour que les fichiers à utiliser dans les règles de fin soient présent et qu'il n'y ait pas de problème d'exécution.
- Une forme avec exécutions en **parallèle**, où Snakemake doit attendre l'exécution de certaines règles avant de lancer celles de la fin.

Nous avons retenu cette dernière option, Snakemake étant optimisé pour paralléliser l'exécution des règles, et sachant attendre que tous les inputs soient créés avant de lancer une règle qui en aurait besoin, cette forme est donc plus adaptée. Ainsi, à titre d'exemple, dans la règle `mapping`, nous avons précisé en input les outputs des règles `makefastq` et `index` pour que Snakemake les éxecute avant de lancer le mapping.

Voici un graphique produit par Snakemake permettant d'illustrer notre pipeline.

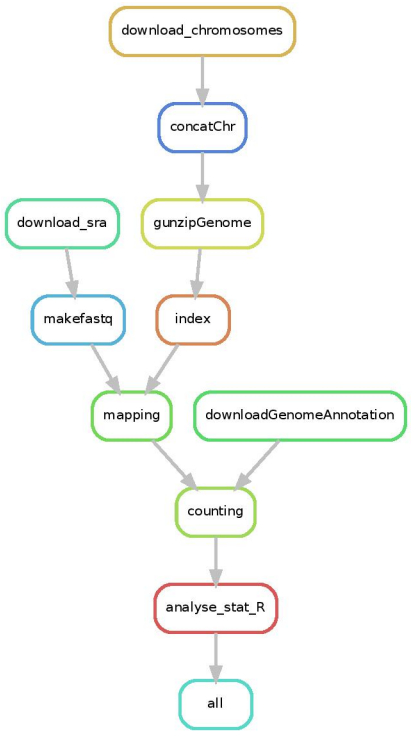

Figure 5: Illustration du pipeline

Un autre graphique plus détaillé - également produit par Snakemake - se trouve en annexe, et permet de bien voir comment les règles sont liées par les inputs et les outputs.

Pour rendre le pipeline utilisable pour un nombre de coeurs alloués variable nous avons réfléchi à un moyen de calculer le nombre de threads pour chaque étape en fonction du nombre de coeurs donné lors de l'exécution du pipeline. Pour cela nous avons écrit quelques commandes en Python dans le haut de notre pipeline pour récupérer le nombre de coeurs passés en paramètre et subdiviser ces coeurs pour chaque étape (partie « ## Répartition du nombre de threads total » dans le Snakefile). La répartition des threads s'est faite en respectant les proportions de threads que nous avons utilisées lors de nos expériences en allouant 16 coeurs au pipeline complet car c'est avec cette répartition que notre pipeline semblait optimal.

4.3.2 Execution du pipeline sur une machine distante

Dans cette partie nous détaillons l'ensemble des étapes et commandes nécessaires pour utiliser notre pipeline. On précise ici que nous avons utilisé un environnement linux pour lancer notre pipeline donc les commandes utilisées sont en langage bash.

- Création du répertoire dans lequel on récupère l'ensemble des fichiers stockés sur le répertoire GitHub: pour cela il faut entrer depuis le terminal la commande suivante: « `git clone` https://github.com/elodieyan/hackathon_groupe6.git »
- Activation de l'environnement conda (il faut s'assurer au préalable de posséder un environnement conda), avec la commande suivante :
  « `conda activate` »
- Installation de Singularity, version 3.6.3, en exécutant la commande suivante :
  « `conda install singularity=3.6.3 >>`
- On va ensuite se placer dans un répertoire où l'on sait que l'on peut stocker de grandes quantités de données. Pour cela, on tape simplement la commande :
  « `cd path/to/your/data/repertory` » (dans notre cas c'était le répertoire `mydatalocal`)
- **Attention : En raison d'un problème de fichier de configuration lors de l'utilisation du docker `evolbioinfo/sratoolkit:v2.10.8` (voire section suivante), pour la première utilisation du pipeline il faut lancer préalablement un autre Snakefile également contenu dans le répértoire GitHub, permettant uniquement la configuration de la machine**. Pour cela vous pouvez lancer la commande suivante: « `snakemake --use-singularity --cores 1 -s path/to/gitrepository/sraConfigurator` ». Un panneau bleu de configuration apparaît ; il ne faut pas modifier les configurations mais juste sauver et quitter avec les boutons « save » et « exit ».

- On va ensuite lancer le pipeline en faisant appel au Snakefile contenu dans le répertoire GitHub en effectuant la commande suivante :
  « `screen -d -m snakemake --use-singularity --cores 16 -s /path/to/Snakefile` »

Sur cette dernière commande il faut préciser certaines options. Le pipeline peut avoir un temps d'éxécution très long (de l'ordre de la journée par exemple) ; pour ne pas perdre la main sur le terminal et éviter les ruptures de connexions ssh qui entraineront l'arrêt inopiné du pipeline, on fait tourner ce dernier en "tâche de fond". Pour cela, on utilise la commande « `screen -d -m` » qui permet de détacher le run de la machine personnelle. Pour se reconnecter au run qui a été détaché on peut utiliser les commandes « `screen -list` » qui donnent la liste des différents run en tâche de fond - dont celui du pipeline qui a été lancé - et ensuite « `screen -r screen_du_pipeline` » qui permet de raccrocher le pipeline. Si on veut détacher à nouveau le pipeline du terminal on peut effectuer les commandes suivantes « ctrl » + A puis D (en maintenant « ctrl »). L'option `--use-singularity` indique à Snakemake que l'on va utiliser des containers importés en utilisant Singularity. L'option `--cores` permet d'indiquer le nombre de coeurs que l'on souhaite allouer au pipeline en général (voir section Snakemake pour la répartition des coeurs au sein même du pipeline, à savoir que « `threads` » et « `cores` » signifient la même chose). Enfin l'option `-s` permet simplement de préciser le chemin et le nom du Snakefile qu'on appelle.
Le temps de run du pipeline est de 3h17 pour un nombre de coeurs alloués de 16. Pour obtenir ce temps nous regardons le fichier log de snakemake qui contient toutes les étapes du pipeline avec les heures de débuts et de fin et qui est enregistré dans le répertoire où on lance la commande. Pour cela il faut taper `cat .snakemake/log/le_nom_dulog`, le nom contient la date de lancement pour se repérer.

Nous avons décidé de lancer l'analyse statistique indépendamment de notre pipeline dont les résultats finaux sont les données d'entrée de l'analyse. Pour avoir plus de précisions sur cette étape, nous vous conseillons de regarder le document « Guide_Utilisation.txt » sur notre GitHub.

4.4 Etapes du projet et containers utilisés

4.4.1 Téléchargement des données et conversion en fichier FastQ (SRA Toolkit)

Cette étape a servi à récupérer les 8 séquences que nous avons analysées et à les convertir en fichier FastQ. La récupération des séquences s'est faite à l'aide d'une commande `wget` vers le portail du NCBI (Snakefile: `rule download_sra`). Les séquences récupérées sont celles que notre programme va pouvoir détranscrire en ADN. Pour cela, on utilise le container SRA Toolkit (`evolbioinfo/sratoolkit:v2.10.8`) avec la fonction `fastq-dump` et l'option `--split-files` qui va générer, pour chaque séquence d'ARN, deux fichiers FastQ selon la méthode du "Paired-end" :

un correspondant à une lecture forward de la séquence d'ADN et un correspondant à une lecture backward (Snakefile: `rule makefastq`).

4.4.2 Mapping des fichiers FastQ sur un génome de référence (STAR)

L'étape précédente permet d'obtenir, pour chacun des 8 fichiers de séquences ARN, 2 fichiers FastQ (séquences ADN). Dans cette étape, l'objectif est de mapper les séquences sur un génome humain de référence.

Téléchargement et indexation du génome humain de référence

Pour obtenir le génome humain de référence, nous avons récupéré les 25 séquences de chromosomes humains disponibles sur Ensembl (chromosomes 1 à 22 et séquences MT, X et Y) à l'aide d'une commande `wget` (Snakefile: `rule download_chromosomes`). Nous avons concaténé ces 25 séquences en un seul fichier pour avoir le génome humain de référence (Snakefile: `rule concatChr` puis `rule gunzipGenome`).

Le génome humain étant très volumineux, il est préférable de l'indexer pour pouvoir, lors de l'étape du mapping, se rendre directement à un endroit souhaité du génome. Pour cela, nous utilisons le container STAR (`evolbioinfo/star:v2.7.6a`) avec l'option `--runMode genomeGenerate` (Snakefile: `rule index`). Cette étape est assez longue mais l'indexation de chaque séquence est indépendante donc on utilise l'option `--runThreadN` pour aligner plusieurs séquences en parrallèle.

Mapping des fichiers FastQ

Maintenant que le génome humain de référence a été obtenu et indexé, nous pouvons mapper les fichiers FastQ sur ce génome de référence. Pour cette étape, nous utilisons également le container STAR (`evolbioinfo/star:v2.7.6a`). Nous donnons en entrée les 16 fichiers FastQ (`--readFilesIn {input.i1} {input.i2}`) et indiquons l'endroit où se trouve l'index du génome (`--genomeDir genome_index`). En sortie, nous obtenons des alignements sous forme de fichiers BAM triés (`--outSAMtype BAM SortedByCoordinate`) en allouant une mémoire maximale de 50 Gb pour ce tri (`--limitBAMsortRAM 50000000000`). Nous ne demandons pas de fichiers en sortie pour les séquences non mappées (`--outSAMunmapped None`).

Avec les options `--outFilterMismatchNmax 4` et `--outFilterMultimapNmax 10`, nous indiquons que nous souhaitons garder les alignements qui ont moins de 4 mauvais appariements et ont mappé moins de 10 locus (Snakefile: `rule mapping`).

4.4.3 Calcul du niveau d'expression des gènes (Subread)

Pour calculer le niveau d'expression des gènes, nous avons besoin des annotations du génome humain de référence. Nous les téléchargeons via une commande `wget` (`rule downloadGenomeAnnotation`). Le fichier GTF téléchargé contient des informations sur la localisation des exons, les positions de départ et de fin. Nous utilisons ensuite le container Subread (`evolbioinfo/subread:v2.0.1`) avec la fonction `featureCounts` pour calculer les niveaux d'expression des gènes : la fonction prend en entrée les fichiers BAM générés à l'étape de mapping et le génome annoté et retourne une matrice contenant les niveaux d'expression des gènes pour chacune des 8 séquences ARN.

L'option `-t` de la fonction `featureCounts` indique les types de séquence que l'on recherche. Ici, comme nous nous intéressons aux gènes, nous avons choisi `-t gene`. L'option `-s` de la fonction `featureCounts` peut prendre trois valeurs : 0 pour "unstranded", 1 pour "stranded" et 2 pour "reversely stranded". Cette option permet de donner des informations sur la manière dont la librairie de séquençage a été produite. L'option "stranded" utilise les régions lues en lecture forward et l'option "reverse stranded" utilise les régions lues en lecture backward. Nous avons choisi ici l'option "unstranded" : cette option permet de compter à la fois les régions des brins positifs (forward), les régions des brins négatifs (backward) et les régions overlap (superposition) lorsqu'elles sont alignées avec le génome de référence. Ici, nous avons des fichiers de lecture forward et de lecture backward : il est donc logique d'utiliser l'option "unstranded".

4.4.4 Analyse statistique de résultats (DESeq2)

Dans cette étape, nous réalisons une analyse statistique des niveaux d'expresison des gènes calculés précédemment. L'objectif est de mettre en évidence les gènes pour lesquels les niveaux d'expressions dans le génome wild-type et dans le génome muté sont significativement différents. Pour réaliser cette analyse statistique, nous utilisons un script R que nous appelons dans le Snakefile avec une section `script : "scriptR.R"`. En R, nous importons le package DESeq2, et dans le Snakefile, nous utilisons le container DESeq2 (`evolbioinfo/deseq2:1.28.1`).

Avant de faire l'analyse de l'expression différentielle des gènes, on réalise une Analyse en Composantes Principales (ACP) pour regarder comment les échantillons sont distribués dans l'espace des gènes. Cette analyse est un procédé courant en statistique qui permet de translater un nuage de points en grandes dimensions, après une normalisation pour centrer et réduire les données (intégrée à la fonction utilisée), dans un espace de dimensions réduites. Cela nous permet de grouper les gènes en dimensions porteuses d'inertie du nuage de points (les 8 échantillons) pour avoir une première visualisation de leur distribution et de leurs similarités. Ici, on ne cherche pas à savoir quels gènes sont les plus porteurs de variance pour les données des échantillons (ce qui n'aurait aucun sens au vu du cercle de corrélation obtenu, et du faible nombre de données à analyser),

mais bel et bien à représenter les données dans un espace réduit pour les comparer plus facilement. Pour ce faire, nous utilisons les packages `FactoMineR` et `factoextra`, qui permettent de réaliser une ACP et de visualiser les résultats sur le graphe de représentation des individus. Les résultats sont présentés dans la partie Résultats de ce rapport (figure 1).

En sortie de l'étape du calcul du niveau d'expression des gènes pour chaque échantillon, nous obtenons des matrices de comptage contenant les informations suivantes : l'identifiant du gène, le chromosome sur lequel il est situé, le nucléotide de départ, le nucléotide de fin, le sens de lecture du brin (reverse ou forward), la taille du gène en nombre de nucléotides, le comptage du nombre de fois où il y a correspondance entre le gène et l'échantillon, ce qui correspond au niveau d'expression du gène dans l'échantillon. Cette dernière donnée est celle qui nous intéresse pour faire l'analyse différentielle de l'expression des gènes selon que les échantillons proviennent d'organismes wild-types (WT) ou mutés. La fonction `DESeq` réalise l'analyse en prenant en entrée un objet de type DESeqDataSet, qui peut être obtenu de différentes manières. Dans notre cas, nous utilisons la fonction `DESeqDataSetFromMatrix`, qui prend en entrée :

- **Une matrice de comptage** : cette matrice ne doit contenir en valeurs que les comptages correspondant aux niveaux d'expressions des gènes. Pour l'obtenir, nous avons réuni en une seule matrice les 8 dernières colonnes des 8 fichiers issus de l'étape de l'analyse d'expression des gènes en une seule matrice, dont nous avons gardé en information les noms des gènes correspondants aux noms des lignes de cette matrice. Nous avons également précisé les noms des échantillons en noms de colonnes pour ne pas perdre la correspondance entre les comptages et les échantillons.

- **Un objet type `data.frame`** contenant des colonnes dont le nombre de valeurs correspond au nombre de colonnes de la matrice de comptage. Ces données ont un sens biologique et sont les piliers de l'analyse différentielle puisque c'est selon leurs valeurs que l'on effectue le test statistique. Ainsi, nous avons entré le nom des échantillons dans une première colonne, et le statut "WT" ou "M" (respectivement pour wild-type et muté), qui est le critère selon lequel l'analyse différentielle est réalisée.

- **Un schéma pour l'analyse différentielle** : on spécifie ici selon quelle liste d'attributs `DESeq` devra effectuer l'analyse différentielle, à savoir dans notre étude la liste "mutations" qui contient le statut "WT" ou "M" correspondant à chaque échantillon.

Pour pousuivre l'analyse, l'objet `DESeqDataSet` obtenu doit subir l'analyse par la fonction *DESeq* qui fournit les résultats, accessibles par l'intermédiaire de la fonction `results`.

Initialement intrigué.e.s par le nombre conséquent de gènes différentiellement exprimés, et par la très forte sur-expression ou sous-expression de certains, nous nous sommes interrogé.e.s sur la

normalisation des données. Il s'avère que la fonction `DESeq` intègre à son analyse une première phase de normalisation des données selon une méthode particulière qui est adaptée à ce type de données. En effet, les données proviennent d'échantillons différents dont on a assemblé artificiellement les résultats d'expression des gènes. Pour rendre les données des différents échantillons comparables, il est important de prendre en compte la profondeur de séquençage propre à chaque échantillon, qui correspond au nombre de reads alignés sur un gène dans un échantillon. Pour un même gène exprimé de la même manière, un échantillon dont la profondeur de séquençage est deux fois plus élevée qu'un autre va produire un résultat d'expression du gène deux fois plus élevé, alors que ce n'est pas la réalité biologique.

Il est donc important de normaliser, ce que fait naturellement la fonction `DESeq` par la méthode suivante : on détermine un facteur de taille spécifique à chaque échantillon qui correspond à la médiane du ratio entre l'expression des gènes et la moyenne géométrique d'expression par gène pour tous les gènes : pour un échantillon i sur une étude sur J gènes, le facteur f_i est le suivant : $f_i = mediane(\{\frac{counts(i,j)}{M_g(j)}\}_{1 \leq j \leq J})$ où $M_g(j)$ est la moyenne géométrique de l'expression du gène j à travers tous les échantillons. La moyenne géométrique a été préférée à la moyenne arithmétique car moins sensible aux valeurs extrêmes. Les données issues de l'analyse ont donc été préalablement normalisées et peuvent être interprétées dès leur sortie de la fonction `DESeq`. Nous avons donc réalisé des graphiques pour pouvoir les interpréter (figure 2).

Pour la construction de nos graphiques, nous avons choisi d'utiliser le package `EnhancedVolcano`, qui est un package R permettant de construire des volcano plot. Comme nous importons un package dans ce code nous avons choisi d'indiquer à l'utilisateur de lancer ce code après avoir exécuté le pipeline sur Snakemake. Les graphiques sont construits à partir des données sorties du pipeline donc cela ne pose pas de problème au niveau de la reproductibilité. Nous avons choisi d'utiliser la p-value ajustée qui prend en compte le nombre de tests effectués jusque là, ce qui n'est pas le cas de la p-value. L'ajustement permet de recalibrer la p-value et donc d'avoir une valeur plus interprétable.

Le foldchange est le ratio obtenu par le niveau d'expression d'un gène chez les sujets mutés divisé par le niveau d'expression de ce même gène chez les patients wild-types. Sur les volcanos plot nous avons choisi de fixer deux seuils : un pour la p-value ajustée et un pour le foldchange. Pour la p-value nous avons fixé un seuil de 0.05 (une p-value inférieure à 0.05 signifie que le résultat du test est statistiquement significatif). L'axe des ordonnées est un -log10 pour plus de visbilité. Sur le graphique, cela se traduit par la barre à 1.3 (car -log10(0.05) = 1.3) séparant les points verts des points rouges.

Un foldchange de 1 signifie que le degré d'expression du gène est le même entre les sujets mutés et wild-types. Sur le graphique cela correspond au zéro (car -log2(1) = 0). Il est plus logique que l'abscisse soit en log2 car, par exemple, un ratio de 0.5 et de 2 signifient tous les deux une

expression du gène variant du simple au double mais cela se lirait beaucoup moins facilement avec un axe non logarithmique (sur un axe logarithmique ces deux valeurs seraient symétriques par rapport au zéro). Nous avons choisi de représenter une délimitation à 1 et -1 signifiant que tous les points respectivement à droite et à gauche de ces droites indiquent un niveau d'expression au moins 2 fois plus important dans l'une des conditions.

4.5 Machines virtuelles utilisées

Nous avons utilisé des machines virtuelles de l'appliance BioPipes, que nous avons lancées sur le cloud de l'Institut Français de Bioinformatique (IFB). Les machines virtuelles BioPipes possèdent un environnement conda comportant les dépendances nécessaires à l'utilisation de **snakemake**. Au débuut du projet nous avons utilisé des machines peu puissantes : 4 coeurs, 16 Go de RAM, 220 Go d'espace mémoire (ifb.m4.xlarge). Cela était suffisant pour commencer à implémenter l'étape 1, c'est-à-dire télécharger une ou deux séquences SRRXXXXXX et les convertir en fichiers FastQ. Cependant, nous sommes rapidement passé.e.s à de plus grosses machines avec au moins 16 coeurs, 64 Go de RAM et 920 Go d'espace mémoire (ifb.m4.4xlarge). En effet la généralisation des règles à toutes les séquences ainsi que les règles d'indexation et de mapping nécessitaient d'avoir avec au moins 64 Go de RAM et un nombre conséquent de coeurs étaient utiles à la parralléisation des tâches pour augmenter la rapidité d'acquisition des résultats de chaque expériences.

5 Conclusion

Lors de ce projet nous avons tenter de reproduire une partie des traitements des données mis en place dans deux articles scientifiques. Le traitement de ces données s'est effectué en quatre étapes principales : une étape de téléchargement des données et de transformation de ces dernières au format FastQ, une étape de "mapping" de ces données sur un génome humain de référence (préalablement télécharg2 et indexé), une étape de calcul du niveau d'expression des gènes grâce aux données du mapping, et une étape d'analyse statistique de ce niveau d'expression pour identifier les gènes différentiellement exprimés entre les sujets mutés et les sujets wild-types.

Pour cela, nous avons créé un workflow avec Snakemake en utilisant Singularity pour faire appel à des containers pré-existants comportant les dépendances nécessaires à notre pipeline. En addition de ce workflow, nous avons créé un répertoire GitHub comportant notre pipeline principal annoté mais également les ressources et informations nécessaires à sa bonne compréhension et réutilisation. L'ensemble de ces éléments permettent de rendre nos analyses reproductibles.

Les résultats de nos analyses ne sont pas les mêmes que ceux des articles, avec un nombre de gènes différentiellment exprimés considérablement différent, malgré des proportions de gènes

sous-exprimés et sur-exprimés proche de l'un des articles. Cette différence est sûrement due au fait que les outils utilisés ne sont pas les mêmes. De plus, nous pouvons noter que les résultats entre les deux articles sont eux aussi éloignés. Si nous avions eu plus de temps il aurait pu être intéressant de regarder quels gènes sont différentiellement exprimés, de regarder leur(s) fonction(s) et de comparer avec les gènes décrits dans les deux articles.

Concernant le workflow, nous avons pensé à quelques pistes d'amélioration que nous n'avons pas eu le temps de finaliser. Par exemple la construction d'un nouveau container pour l'analyse statistique nous permettant d'inclure la réalisation de nos graphiques réalisés avec des librairies R non présentes dans le container d'origine (explications plus détaillées en annexe). Nous avons également réfléchi à l'utilisation de l'option de Snakemake `--report` permettant d'obtenir un rapport d'analyse de l'éxécution de notre pipeline (temps de run, répartition des tâches dans le temps, graphiques) mais nous n'avons pas réussi à l'intégrer correctement à notre pipeline avant la fin du projet.

Ce projet nous aura permis de nous familiariser avec des outils permettant la construction de workflow, ce qui nous sera sans nul doute très utile professionnellement. Nous tenons à remercier Monsieur Lemoine et Monsieur Cokelaer pour ce projet enrichissant et pour leur accompagnement.

Annexe 1 : Schéma de notre pipeline Snakemake

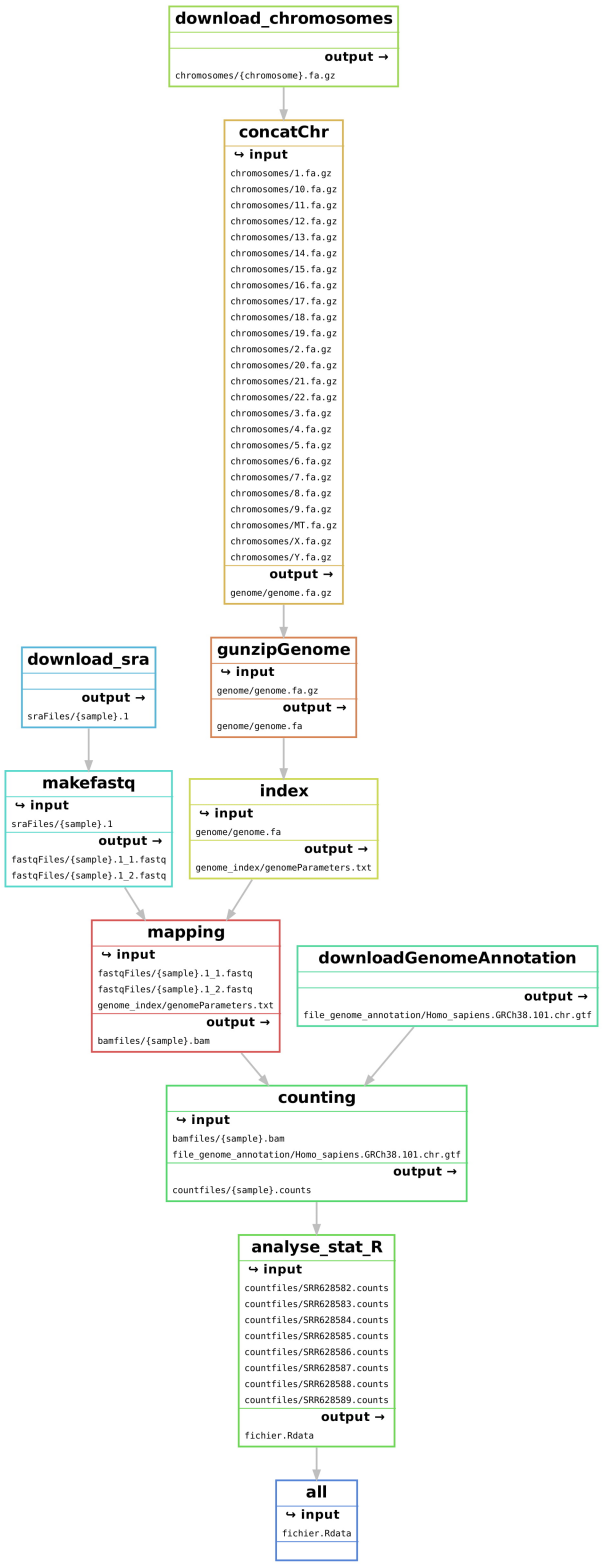

Figure 6: Rulegraph Snakemake expliquant notre pipeline

Annexe 2 : Explication sur la création d'un nouveau container

La dernière étape de notre pipeline produit un environnement R contenant les données issues du traitement statistique et les données brutes à utiliser pour l'ACP. Utilisant des packages R qui ne sont pas disponibles dans le container qui nous était mis à disposition pour la visualisation graphique des résultats de l'ACP et de l'analyse de l'expression différentielle, nous vons voulu créer un container Singularity en utilisant la plateforme Sigularity Hub couplée à un répertoire GitHub dédié à cet essai. Le répertoire GitHub est le suivant : https://github.com/GuillaumeStuder/essai_hackathon

Vous trouverez sur ce répertoire :

- Un fichier *Singularity* qui est la "recette" utilisée par Singularity Hub pour créer le container.
- Un dossier *Workflow_and_results* contenant les fichiers pour l'analyse : le fichier de sortie du pipeline, *fichier.Rdata*, le script R à exécuter via snakemake, *script_graphs.R* et le fichier snakemake *script_graphs_snakemake*.
  On y trouve également le fichier *graph_individuals_pca.pgn* obtenu en exécutant la commande `snakemake --use-singularity --cores 1 -s script_graph_snakemake` dans la console. En effet, le container ne contient pas le package EnhancedVolcano permettant la construction du graphique de résultats de l'analyse différentielle, mais il fonctionne pour le résultat de l'ACP. Nous avons donc commenté ce qui concernait l'expression différentielle pour rendre le script R exécutable, et retiré en output de la règle Snakemake le graphe correspondant à l'analyse différentielle.
- Un dossier *dock* contenant un fichier *Dockerfile* car nous avons égelament essayé avec Docker Hub de créer un container, ce qui n'a pas fonctionné.

Nous aurions aimé poursuivre cette piste d'amélioration de la reproductibilité de l'analyse, mais nous n'avons pas eu le temps.