

## Sommaire :

### Introduction

#### 1)Présentation de l'organisme

##### 1.1)présentation générale

##### 1.2)conditions du stage

#### 2)

##### 2.1)Galaxy

##### 2.2)*Wrappers*

##### 2.2.1)Q'est ce qu'un wrapper ?

##### 2.2.2)Composition d'un wrapper

##### 2.2.3)Le Tool shed

##### 2.3)Test logiciel

#### 3) Développement de tests pour des wrappers dans Galaxy

##### 3.1)L'outil bioinformatique

##### 3.2)Modification du wrapper

##### 3.3)Tests locaux

##### 3.4)Intégration dans le dépôt Github

##### 3.5)Intégration continue

#### 4) Liste des wrappers d'outils testés

#### 5)Cas particuliers traités :

##### 5.1)Paramètres

##### 5.1.1)Multiplicité de données d'entrée

##### 5.1.2)Nombre de fichiers en sortie inconnu

##### 5.1.3)Particularité de humann2\_join\_tables

##### 5.2)Dépendances

# Résumé

ASaiM (Auvergne Sequence analysis of intestinal Microbiota) est un environnement créé pour analyser les microbiotes et plus particulièrement le microbiote intestinal. Afin d'avoir une analyse correcte de ce microbiote, il faut étudier l'ensemble des génomes issus du milieu et donc faire de la métagénomique.

Il existe de nombreux outils bioinformatiques permettant d'analyser des données métagénomiques et présentant un intérêt certain à être intégrés au sein d'ASaiM notamment HumanN2, MetaphLan2 ou encore GraphLan.

Cependant, ces outils, comme la grande majorité des outils bioinformatiques, doivent être lancés en ligne de commande. Et la majorité des biologistes n'ont pas ou peu de connaissances en programmation et ont donc du mal à utiliser ces outils. À cause de cela, ils doivent faire appel à une personne ayant les connaissances suffisantes où se tourner vers un autre moyen d'analyse, comme les tableurs. Ce qui rend ces analyses bien plus longues et fastidieuses.

Afin de venir à bout de ce problème, une instance basée sur la plateforme Galaxy a été intégrée à ASaiM afin de permettre une utilisation facilitée de ces outils. Grâce à des logiciels faisant office d'interfaces web nommés wrappers et adaptées aux outils il est possible de simplifier grandement l'utilisation de ces derniers. Cependant, comme tous logiciels, ces wrappers ont besoin d'être testés afin de garantir leur fiabilité.

L'objectif de mon stage est de produire des données de tests convenables pour les différents wrappers afin de garantir cette fiabilité et permettre de les publier notamment sur l'instance Galaxy d'ASaiM.

# **1)Présentation de l'organisme**

## **1.1)présentation générale**

Fondée en 1976 lors de la scission de l'université de Clermont, l'Université d'Auvergne (UdA) actuellement présidée par Alain Eschalier compte plus de 16 000 étudiants dont 3 000 étrangers rassemblés au sein de sept composantes :

- École de Droit
- École d'Économie
- École Universitaire de Management
- Faculté de Médecine
- Faculté de Pharmacie
- Faculté de Chirurgie Dentaire
- Institut Universitaire Technologique

L'université dispose aussi de 22 laboratoires de recherche

## **1.2)conditions du stage**

J'ai effectué mon stage au sein du laboratoire EA CIDAM (Conception, Ingénierie et Développement de l'Aliment et du Médicament).

Dirigé par le professeur M. Alric, l'équipe de recherche travaille notamment à comprendre, évaluer et analyser, dans l'environnement digestif, différentes situations physio-pathologiques liées au vieillissement, à la présence de bactéries pathogènes (en particulier d'*Escherichia coli* entérohémorragiques), ou encore à celle de produits toxiques, de xénobiotiques, en particulier de polluants.

C'est dans ce cadre-là que Bérénice Batut, ma tutrice de stage, développe un environnement bioinformatique permettant de faciliter l'analyse de données massives issues du microbiote, ASaiM.

## **2) Introduction**

L'objectif d'ASaiM est d'apporter une simplification des analyses de données métagénomiques issues du séquençage de microbiote. Il existe de nombreux outils bioinformatiques permettant ces analyses. Cependant, ils fonctionnent en ligne de commande. La majorité des biologistes maîtrisant que très peu voire pas du tout l'informatique, il est nécessaire de trouver une voie détournée pour utiliser ces outils. Ainsi une instance galaxy a été

développée au sein d'ASaiM dans le but de faciliter l'utilisation de ces outils via des interfaces. Cependant une majorité de ces outils ne possèdent pas d'interface qui les lient à galaxy, il faut donc en développer.

## **2.1) Galaxy :**

La plate forme Galaxy a été privilégiée pour l'intégration des outils, car celle-ci dispose de nombreux avantages. Tout d'abord, le fait qu'elle soit ouverte et open source, cela signifie que ses données sont accessibles librement en ligne, ce qui permet à n'importe qui d'installer cette plate-forme et de profiter de tous les logiciels qui lui sont liés.

Ainsi, il est possible d'utiliser cette instance via les serveurs publics dont l'avantage étant qu'ils disposent le plus souvent d'une importante puissance de calcul. La fréquentation de ces serveurs risque cependant de créer des temps d'attente et il n'y a pas ou peu d'interactions possibles avec les administrateurs. Afin d'avoir plus d'indépendance et plus de liberté au niveau administratif, nombre de laboratoires et d'entreprises ont développé leur propre instance de Galaxy qui disposent généralement d'une bonne puissance de calcul et dont les administrateurs sont disponibles lorsqu'il s'agit d'ajouter des outils. L'objectif d'ASaiM est de devenir l'une de ces instances.

Il est aussi possible de télécharger une instance de Galaxy en local sur n'importe quel ordinateur. Cependant, dans ce cas-là, il faut gérer soit même l'installation des outils et la puissance de calcul est limitée à celle de l'ordinateur. Lors du stage, nous avons téléchargé une instance de Galaxy sur un ordinateur afin d'effectuer les tests des wrappers en développement.

L'interface de Galaxy est claire. Sur la gauche on retrouve la liste des outils disponibles au centre l'interface de l'outil utilisé et sur la droite l'historique et la liste des fichiers utilisés (voir Figure 1). Ces fichiers sont stockés dans l'historique de l'instance et ne peuvent être définitivement supprimés que par un administrateur. Ainsi même plusieurs mois après, il est possible de récupérer à nouveau nos données.

Mais l'intérêt majeur de cette instance est de pouvoir lancer des outils de bioinformatique, qui fonctionnent normalement en ligne de commande afin d'effectuer des analyses complètes, mais surtout reproductibles. Cela grâce aux wrappers, des interfaces faisant le lien entre l'utilisateur et la ligne de commande de l'outil.

### **2.2.1) Le Tool shed**

Ces *wrappers* ne sont pas automatiquement disponibles sur galaxy. Il faut donc les télécharger. Ils sont stockés au sein d'une banque de donnée accessible à tous, le tool-shed. Cette banque de donnée fonctionne selon le

même principe que les plate-formes de téléchargement mobile. La totalité des logiciels présents sont mis à disposition et téléchargeables. Ils sont répartis dans des répertoires selon la fonction de l'outil auquel ils sont liés. Le tool-shed dispose aussi d'une barre de recherche nominale (voir Figure 2).

Il existe deux tool-shed principaux, tous deux maintenus par l'équipe de développement de Galaxy. Tout d'abord, le Main tool-shed où l'on retrouve tous les wrappers utilisables, puis le Test tool-shed qui regroupe tous les wrappers mis en lignes, mais encore en cours de développement.

Il est aussi possible d'avoir son propre tool-shed sur son ordinateur cependant il est principalement utilisé lors de la phase de développement ou lorsque l'on souhaite garder ses outils et wrappers confidentiels.

Lors du dernier recensement effectué au sein du Main tool-shed, le 15 Mai 2016, 3904 wrappers étaient disponibles.

### **2.2.2)Wrappers :**

Les wrappers sont la base de Galaxy. Ils servent d'interface entre l'utilisateur et la ligne de commande qui lance l'outil.

Un wrapper utilise une version de l'outil sélectionnée préalablement. Ainsi même si l'outil est mis à jour, le wrapper continue d'utiliser la version prédéfinie. Cela rend les analyses effectuées reproductibles. En effet, après une mise à jour, l'outil peut analyser les données différemment nécessiter un type de fichiers d'entrée différent ou encore fournir des fichiers de sortie n'ayant rien à voir avec les fichiers fournis avant la mise à jour. Dans ce cas-là il y a un fort risque que les résultats issus de la comparaison de fichiers issus de différentes ne soit pas biaisée et fausse les résultats finaux.

Les wrappers utilisés dans Galaxy sont de logiciels codés en langage .xml qui est un langage qui permet de décrire et de structurer des données à l'aide de balises qu'il est possible de personnaliser. Un wrapper est composé de trois fichiers. D'abord, le fichier principal qui reprend le code du wrapper, c'est ce fichier qui nous intéresse. Puis un fichier tool\_dependencies qui gère l'installation des dépendances nécessaires à l'outil. Et un .shed.yml qui gère l'installation de l'outil dans le tool-shed.

Le fichier principal est composé de la manière suivante (voir Figure 3) :

On retrouve d'abord les informations générales sur l'outil qui regroupent son nom, son id, sa version et sa description. Vient ensuite la section "requirements" qui fournit les informations sur les dépendances nécessaires à leur prise en charge.

Ensuite, vient la partie "command". C'est elle qui fait le lien entre la ligne de commande de l'outil et les paramètres donnés dans Galaxy. Elle reprend la totalité de la ligne de commande : le nom de l'outil, le ou les fichier d'entrée et de sortie ainsi que toutes les options de l'outil. Chaque fichier ou valeur d'option

est fournie par la section input ou la section output.

De plus, les wrappers gèrent les fichiers d'entrée et de sortie, les paramètres nécessaires ainsi que les dépendances de l'outil, tous les programmes nécessaires au bon fonctionnement de l'outil.

Puis la section "inputs", c'est dans cette partie que l'on définit ce qui va être affiché à l'écran lors de l'utilisation de l'outil dans Galaxy notamment les noms que l'on souhaite donner aux différentes entrées afin de faciliter la compréhension ou encore un commentaire d'aide. On y définit les propriétés des fichiers en entrée, notamment leur type (text, FastA, FastQ, etc.), mais aussi celles de chaque paramètre que l'on souhaite afficher. Tout d'abord, le type du paramètre, il peut s'agir d'une valeur à rentrer (entier ou décimal) ou d'un texte, ce paramètre peut aussi être une barre de sélection ou encore une check box. Chaque fichier d'entrée ou paramètre que l'on souhaite utiliser doit être présent dans cette section.

La section "outputs" permet de gérer les sorties de l'outil que l'on veut voir apparaître dans la barre de l'historique de Galaxy. On y définit notamment le nom que l'on veut donner à la sortie. On peut aussi modifier son format. Toutes les sorties de l'outil doivent être référencées dans cette section.

La section "tests" est, comme son nom l'indique, liée à la phase de test du wrapper. On fournit dans cette section les fichiers d'entrée et de sortie nécessaires pour le test du wrapper ainsi que les valeurs des différents paramètres. On peut fournir autant de jeux de données différents afin d'avoir une batterie de tests la plus variée possible. Cependant, la totalité des fichiers et des paramètres doivent être testés au minimum une fois chacun.

On retrouve à la fin la section "help" qui permet d'afficher des informations sur l'outil. Et la section "citations" dans laquelle on liste les différentes citations avec leur DOI.

## **2.2) Test logiciel :**

Tout logiciel ou programme informatique, tels que les wrappers par exemple, a besoin d'être testé afin de garantir une bonne fiabilité au niveau du fonctionnement du programme ainsi que son comportement en présence des différentes entrées.

Le test est une technique de contrôle qui consiste à lancer le logiciel avec des données d'entrée préparées à l'avance et de comparer ce que le programme renvoi avec les sorties attendues.

L'objectif d'un test est d'exécuter un programme dans l'intention d'y trouver des défauts et non pas pour démontrer que le programme ne contient plus d'erreur. Il faut donc que la personne chargée des tests ait pour but de

trouver des erreurs, autrement elle n'en trouvera que peu ou pas.

Cependant, il est impossible d'obtenir un programme sans défauts en effet les tests ne peuvent vérifier qu'une partie des possibilités. Cependant, un objectif réalisable est de corriger les erreurs sévères et récurrentes à l'aide de données de test représentatives.

Un autre problème est que le test logiciel est un processus destructif, à l'opposé de la programmation qui est un processus constructif. En effet le but du programmeur est de créer un logiciel qui fonctionne et rechigne souvent à effectuer les tests.

On peut comparer la programmation à l'orthographe, on a toujours plus de mal à corriger ses propres fautes et cela demande plus d'efforts. Ainsi, il arrive souvent que des logiciels créés au sein d'équipes restreintes, comme les wrappers de Galaxy, soient publiés sans tests et donc sans garantie de fonctionnement. Ce qui peut poser problème lors de l'utilisation.

L'une des meilleures solutions afin de produire un test convenable est de faire appel à des personnes extérieures à l'équipe de développement (ce qui n'est pas forcément possible pour des travaux de faible envergure) pour développer ces tests avec un regard neuf sur le logiciel.

Il existe de nombreux types de tests différents (unitaires, intégration, performance, etc.) cependant deux types de tests sont principalement utilisés. Tout d'abord, les tests unitaires dont le principe est de tester de façon indépendante un seul élément du logiciel.

Le second type de test est le test fonctionnel qui consiste dans le fait de tester la totalité d'un programme en reproduisant les « conditions réelles ». Cela veut dire que lors du test nous effectuons ce que l'utilisateur va potentiellement faire. Cela permet de voir à quelles erreurs ce dernier peut être confronté. C'est ce type de test que nous avons pratiqué pour nos wrappers.

### **3) Développement de tests automatiques des wrappers dans Galaxy**

Actuellement, n'importe qui peut mettre ses wrappers en ligne sur le Main tool-shed. Ainsi une part très importante de ces wrappers ne disposent d'aucun fichier de test. On ne peut pas se fier aux sorties de ces wrappers, ils ne vont donc qu'être très peu utilisés. Leur développement aura donc été une perte de temps. De plus, ils risquent de fausser le travail d'autrui. Il est donc important de produire des fichiers de test corrects pour les wrappers. Voici

#### **3.1) L'outil bioinformatique**

Les wrappers sont créés afin de servir d'interface entre l'utilisateur et la

ligne de commande d'un outil bioinformatique comme GaphLan par exemple. Afin de développer un test correct il faut donc tout d'abord prendre connaissance du fonctionnement de l'outil. Il faut donc effectuer des recherches sur la documentation de l'outil (voir Figure 4).

L'outil GarphLan nécessite un fichier d'entrée et quatre paramètres pour fonctionner. Il produit un fichier en sortie.

Voici par exemple une ligne de command permettant de lancer graphlan :

```
graphlan.py input.txt image.png --format png --dpi 100 --size 7 --pad 2
```

On retrouve tout d'abord le nom du fichier permettant de lancer l'outil : graphlan.py Puis le fichier d'entrée de l'outil : input.txt ainsi que le nom que l'on veut donner à son fichier de sortie : image.png Viennent ensuite les quatre paramètres nécessaires au fonctionnement de l'outil :

"--format png" : tout d'abord le paramètre renseignant le format du fichier de sortie, ici on souhaite qu'elle soit au format png

"--dpi 100" : ce second paramètre renseigne la définition ou qualité de l'image, ici elle devra être de 100 points par pouce (mesure)

"--size 7" : ce paramètre renseigne la taille de l'image en sortie en pouces, ici on souhaite que la taille soit de 7 pouces par 7 pouces

"--pad 2" : ce paramètre renseigne la taille des bordures autour de l'image, elle seront ici de 2 pouces

Il faut ensuite s'intéresser aux fichiers d'entrée et de sortie, notamment les données que ces fichiers contiennent ainsi que leur disposition.

Le fichier d'entrée doit contenir des informations taxonomiques liées aux différents organismes que l'on souhaite étudier. Pour chaque organisme, doivent être fournis les huit niveaux taxonomiques (le domaine, le règne, le phylum (ou l'embranchement), la classe, l'ordre, la famille, le genre et l'espèce). Les données peuvent être présentées de deux façons différentes (voir ci-contre).

Le fichier de sortie, quant à lui, doit être une image sur laquelle est représentée un arbre phylogénétique circulaire.

Généralement les fichiers de test ne sont pas fournis et il faut les générer. Dans le cas que l'on étudie, le fichier d'entrée doit être généré. Il s'agit du fichier de sortie de graphlan\_annotate, un outil lié à GraphLan qui ajoute des annotations au fichier d'entrée de GraphLan. Le fichier de sortie de GraphLan doit lui aussi être généré en lançant l'outil avec le fichier d'entrée de test et les paramètres choisis.

Les outils bioinformatiques sont généralement complexes et effectuent des opérations longues et pénibles à programmer autant qu'à tester. On peut éviter cela en faisant effectuer une partie du travail à un logiciel conçu pour cela que l'on peut appeler grâce au *wrapper*, on les appelle dépendances. Elles n'interviennent en rien dans la création des données de test cependant il faut



s'y intéresser, car si elles ne sont pas gérées correctement, elles peuvent être à l'origine d'erreurs.

### 3.2) Modification du *wrapper*

Après avoir pris connaissance du fonctionnement de l'outil, il faut éditer le *wrapper* en conséquence. Il faut générer ou réécrire un ou plusieurs tests et au besoin modifier le corps du *wrapper*.

Dans le cas de GraphLan, nous avons édité la section tests en fournissant les données d'entrées citées précédemment pour un test afin d'observer la réaction du *wrapper* face à un jeu de données standard.

Cependant, suite à l'apparition d'erreurs lors de la phase de test il a fallu corriger le *wrapper*. L'erreur était due à une dépendance qui n'était pas à la bonne version. Il a donc fallu changer la version de la dépendance "graphlan" en passant de 0.9.7 à 1.0.0 .

### 3.3) Tests locaux

Après avoir modifié le *wrapper*, il faut le tester. En commençant par tester en local, sur l'ordinateur.

Pour tester un *wrapper* localement, il existe deux méthodes. La première est la méthode manuelle, la seconde est d'utiliser un outil nommé planemo. Si l'on veut effectuer les tests manuellement, il faut tout d'abord installer galaxy et un tool-shed sur l'ordinateur, pour lancer le *wrapper* en conditions réelles. Puis, il faut télécharger sur le tool-shed local, un à un, la totalité des fichiers et dépendances nécessaires. Cela peut prendre du temps lorsqu'il y a plus de dix fichiers, ce qui est fréquent. Après cela il faut encore lancer les tests en ligne de commande. Cette opération est à répéter à chaque modification.

La seconde méthode utilise planemo, un outil développé dans le but de faciliter le développement des *wrappers* pour Galaxy. Planemo dispose de trois commandes directement liées au test des *wrappers* :

"*planemo lint*" : Cette commande permet de tester le code .xml. Elle sert notamment à repérer les erreurs de frappe ou les oublis (ex : balise ouverte, mais qui n'a pas été fermée)

"*planemo shed-lint*" : Ce test vérifie que le *wrapper* est correctement formé et peut être déployé sur le tool-shed. Il vérifie le code .xml comme lint mais aussi les autres fichiers, notamment le .shed.yml.

"*planemo test*" : Cette commande lance l'installation d'une instance de Galaxy virtuelle et nue puis installe le *wrapper* et ses dépendances avant de lancer l'outil avec les données de test. Elle permet d'effectuer un test en conditions « réelles » du *wrapper*.

Voici la commande utilisée pour tester l'outil GraphLan :

```
$ planemo test --conda_dependency_resolution --conda_prefix $HOME/conda
--galaxy_branch release_16.04 --galaxy_source $GALAXY_REPO --skip_venv
tools/graphlan/
```

"--conda\_dependency\_resolution" : fait en sorte que Galaxy n'utilise conda que pour gérer les dépendances

"--conda\_prefix \$HOME/conda" : donne le chemin de conda

"--galaxy\_branch release\_16.04" : donne la version de galaxy a utiliser

"--galaxy\_source \$GALAXY\_REPO" : fourni le lien pour télécharger Galaxy

"--skip\_venv" : empêche la création d'un environnement virtuel pour conserver la configuration de l'environnement déjà existant

"tools/graphlan/" : chemin vers le *wrapper*

Utiliser planemo pour effectuer les tests engendre un important gain de temps par rapport à la méthode manuelle. De plus, après avoir lancé la commande de test, tout se fait automatiquement.

Si le test échoue, il faut retourner éditer le *wrapper* pour corriger l'erreur. La commande fournit également des informations sur les raisons de l'échec. Si le test est passé, on peut mettre l'outil en ligne.

### 3.4)Intégration dans le dépôt Github

Lors du développement d'ASaiM, ses données ont été stockées sur Github, une plate-forme en ligne utilisant Git, un logiciel de gestion de versions. <https://github.com/ASaiM/galaxytools>

C'est donc sur cette plate-teforme qu'il a fallu télécharger les *wrappers* pour lesquels il fallait un test. Mais c'est aussi là qu'il a fallu les remettre en ligne après l'ajout des données de test. Cependant, utiliser Github et Git présente de nombreux avantages.

Tout d'abord, la possibilité de travailler avec des branches, des versions parallèles d'un même dossier indépendantes les unes des autres. On peut modifier les données d'un fichier sur une branche tout en préservant l'intégrité du même fichier situé sur les autres branches. On peut ainsi travailler à plusieurs sur un même fichier puis fusionner le travail effectué.

De plus, Git suit l'évolution de tous les fichiers et stocke les anciennes versions de chacun d'eux. Il retient aussi qui a effectué chaque modification de chaque fichier et pourquoi. À chaque modification Git requiert qu'elle soit commentée.

La plate-forme Github quant à elle permet le libre accès à tout ce qui a été stocké dessus. N'importe qui peut regarder le code de n'importe quel logiciel, mais aussi les commentaires fait tout au long du développement. Il peut même participer au développement de ces logiciels, grâce aux Pull Requests (PR, voir ). En effet n'importe qui peut proposer une modification du code d'un

logiciel via une PR. Il s'agit d'une demande faite au développeur de l'outil d'ajouter les modifications proposées. On peut y voir les modifications proposées (ce que l'on enlève et ce que l'on ajoute) mais aussi toutes les modifications apportées à la requête. Il est aussi possible de commenter la PR ou encore de s'en servir pour faire de l'intégration continue.

### 3.5)Intégration continue

L'intégration continue consiste à relancer, à chaque modification apportée, la totalité des tests liés au logiciel de façon automatique. De plus, il est possible de programmer une notification automatique, un envoi de mail ou une notification sur un logiciel de tchat, signalant la fin des tests ainsi que leur résultat. Cela génère un important gain de temps et permet d'éviter des erreurs, notamment lors du lancement des tests.

L'idée est de pousser le plus souvent possible, les plus petites modifications possibles. Ceci afin de minimiser les risques d'erreur. De plus, cela permet au développeur de retrouver et de corriger bien plus facilement les erreurs affichées.

Pour cela il faut tout d'abord configurer un serveur qui va gérer cette intégration continue. Lors de ce stage nous avons utilisé Travis. C'est un logiciel spécialement développé pour l'intégration continue. Il crée un environnement totalement vierge avant de n'y installer que les outils et les fichiers nécessaires aux tests. Cela permet d'éviter toute pollution des tests due à des logiciels tiers ou des versions antérieures des fichiers. De nouvelles erreurs peuvent ainsi être détectées ou d'autres résolues.

Durant ce stage, un serveur Travis à été développé afin d'effectuer l'intégration continue à chaque mise à jour de la Pool Request.

## 4) Liste des *wrappers* d'outils testés

Lors de ce stage, nous avons produit des tests pour de nombreux outils. En voici la liste. Nous verrons aussi les cas particuliers auxquels il a fallu faire face lors de la production des tests .

#### extract\_min\_max\_lines :

Cet outil permet d'extraire les lignes dont la valeur maximale ou minimale se trouve sur la colonne sélectionnée .

L'outil requiert un fichier d'entrée, prend en compte trois paramètres et fournit un fichier en sortie.

Un seul *wrapper* est lié à cet outil, pour lequel nous avons créé quatre tests et généré quatre fichiers de sortie, un pour chaque test.

#### combine\_metaphlan2\_humann2 :

Cet outil permet de combiner les sorties de l'outil HunanN2 et de l'outil metaphlan2.

L'outil requiert deux fichiers d'entrée, prend en compte un paramètre et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé deux tests et généré un fichier d'entrée ainsi que deux fichiers de sortie.

#### plot\_generic\_x\_y\_plot :

L'outil crée un diagramme en nuage de points à partir d'un fichier au format tabular.

L'outil requiert un fichier d'entrée, prend en compte quinze paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé un test et généré un fichier d'entrée et un fichier de sortie.

#### normalize\_dataset :

Cet outil permet de normaliser chaque colonne ou chaque ligne et présente les résultats sous forme de proportion ou de pourcentage.

L'outil requiert un fichier d'entrée, prend en compte deux paramètres et fournit un fichier en sortie.

Il n'y a qu'un *wrapper* lié à cet outil, pour lequel nous avons créé trois tests et généré un fichier d'entrée ainsi que trois fichiers de sortie.

#### fasta\_add\_barcode :

Cet outil ajoute un « code barre » au début de chaque séquence sur un fichier au format FastA.

L'outil requiert deux fichiers en entrée et fournit un fichier en sortie. Cet outil ne demande pas de paramètres. Un seul *wrapper* est lié à cet outil, pour lequel nous avons créé un test et généré les deux fichiers d'entrée ainsi que le fichier de sortie.

#### plot\_grouped\_barplot :

L'outil crée un diagramme de barres à partir d'un fichier au format tabular.

L'outil requiert un fichier d'entrée, prend en compte quinze paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé un test et généré un fichier d'entrée et un fichier de sortie.

Lors du développement de cet outil, nous avons été face à un cas particulier. L'outil requiert plusieurs fois une même donnée d'entrée ou un groupe de données.

Présenté ainsi dans la section "inputs" :

```
<repeat name="samples" title="Add sample information">
  <param name="name" ... />
  <param name="column_id" ... />
  <param name="color" ... />
</repeat>
```

Dans la section "tests" les paramètres doivent être présentés de la façon suivante :

```
<test>
  <param name="samples_0|name" value=" ... "/>
  <param name="samples_0|column_id" value=" ... "/>
  <param name="samples_0|color" value=" ... "/>
  <param name="samples_1|name" value=" ... "/>
  <param name="samples_1|column_id" value=" ... "/>
  <param name="samples_1|color" value=" ... "/>
</test>
```

Afin de trouver la forme correcte pour les tests, il a fallu effectuer des recherches pour trouver des exemples d'outils présentant cette particularité et disposant de tests.

#### plot\_barplot :

Cet outil crée un diagramme de barres à partir d'un fichier au format tabular.

L'outil requiert un fichier d'entrée, prend en compte douze paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé deux tests et généré un fichier d'entrée ainsi que deux fichiers de sortie.

#### compare\_humann2\_output :

L'outil compare le contenu de différents fichiers de sortie de humann2 et renvoie les familles de gènes et voies métaboliques similaires entre les fichiers. Cet outil renvoie aussi les plus abondantes et celles qui sont spécifiques aux différents fichiers.

L'outil prend en compte nombre indéfini de fichiers d'entrée, un paramètre présent autant de fois qu'il y a de fichiers d'entrée, un second paramètre et fournit trois fichiers en sortie ainsi qu'un nombre de fichiers de sortie supplémentaires équivalent au nombre de fichiers d'entrée.

Pour le *wrapper* lié à cet outil, nous avons créé un test et généré deux fichiers d'entrée ainsi que cinq fichiers de sortie.

L'outil renvoie un nombre de fichiers de sortie inconnu à l'avance dépendant des fichiers d'entrée ou de certains paramètres. Ils sont fournis par l'outil dans un dossier. Le problème est que Galaxy ne peut pas traiter de

dossier. Il faut donc passer par une collection :

```
<collection name="specific_files" type="list">
  <discover_datasets pattern="__designation_and_ext__" directory="dossier"/>
</collection>
```

Voici les données de test correspondantes :

```
<output_collection name="specific_files" type="list">
  <element name="nom spécifique" file="sortie1.txt" />
  <element name="nom spécifique" file="sortie2.txt" />
</output_collection>
```

La particularité de ce type de sortie est qu'il faut le gérer différemment, selon l'outil.

Les paramètres de "discover\_datasets pattern" peuvent être modifiés pour changer le fonctionnement de la collection et le nom lié à "element name" dépend totalement de l'outil utilisé.

Il faut donc chercher des exemples et effectuer des tests à l'aveugle.

#### export2graphlan :

Cet outil permet de convertir des fichiers issus de MetaPhlAn, LefSe, ou HUMAnN afin qu'ils soient compatibles avec l'outil graphlan.

L'outil requiert un fichier d'entrée, prend en compte vingt-six paramètres et fournit deux fichiers en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test.

#### format\_metaphlan2\_output :

Génère neuf fichiers à partir d'une sortie de l'outil metaphlan. Les huit premiers reprennent les abondances liées aux différents groupes pour chaque niveau taxonomique (ordre, famille, espèce, etc.) . Le dernier reprend tous les niveaux taxonomiques et les abondances correspondantes.

L'outil requiert un fichier d'entrée et fournit neuf fichiers en sortie. Il ne prend pas en compte de paramètres.

Pour le *wrapper* lié à cet outil, nous avons édité un test et modifié huit fichiers de sortie.

#### cdhit :

Les options de cdhit pour lesquelles il a fallu fournir des données de test permettent de regrouper les séquences fournies afin de former des clusters. Deux *wrappers* sont liés à cet outil. Celui de l'option cd\_hit\_est et celui de cd\_hit\_protein.

##### cd\_hit\_est :

Regroupe des séquences nucléotidiques.

L'outil requiert un fichier d'entrée, prend en compte cinq paramètres et fournit deux fichiers en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré deux fichiers de sortie.

#### cd\_hit\_protein :

Regroupe des séquences protéiques.

L'outil requiert un fichier d'entrée, prend en compte six paramètres et fournit deux fichiers en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré deux fichiers de sortie.

#### extract\_sequence\_file :

Cet outil extrait des informations des fichiers de séquences métagénomiques. Il peut aussi convertir un fichier au format FastQ en un fichier au format FastA.

L'outil requiert un fichier d'entrée, prend en compte jusqu'à quinze paramètres, et fournit deux ou trois fichiers en sortie, selon les paramètres sélectionnés. Les deux fichiers fournis dans un cas sont différents des trois fournis dans l'autre.

Pour le *wrapper* lié à cet outil, nous avons créé deux tests. Nous avons généré cinq fichiers de sortie, deux pour le premier et trois pour le second.

#### format\_cd\_hit\_output :

Cet outil permet de formater les sorties de cd-hit afin de renommer les séquences représentatives avec les noms des clusters mais aussi d'extraire la distribution des catégories au sein des clusters.

L'outil nécessite deux fichiers en entrée, nécessite quatre paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré deux fichiers de sortie.

#### GraphLan :

Cet outil crée un arbre phylogénétique circulaire à partir des données phylogénétiques fournies.

Deux *wrappers* sont liés à cet outil. Le premier correspond à l'outil graphlan et le second est lié à graphlan\_annotate une option de l'outil.

#### Graphlan :

L'outil requiert un fichier d'entrée, prend en compte quatre paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré un fichier d'entrée ainsi qu'un fichier de sortie.

Lors du développement de cet outil, nous avons obtenu une erreur particulière, il s'agissait d'une erreur due à un problème de dépendance. Pour corriger ce genre d'erreur il faut se renseigner sur les dépendances, notamment

via la documentation ou le rapport d'erreur du test, puis installer ou mettre à jour ce qui est nécessaire.

Pour l'outil graphlan, il a fallu mettre à jour la dépendance "graphlan" en passant de 0.9.7 à 1.0.0.

#### graphlan\_annotate :

Permet d'annoter les fichiers d'entrée de graphlan et ajoute des informations supplémentaires modifiant l'aspect structurel ou graphique de l'arbre phylogénétique produit par graphlan.

L'outil requiert deux fichiers en entrée et fournit un fichier en sortie. Cet outil ne demande pas de paramètres.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré l'un des deux fichiers d'entrée ainsi que le fichier de sortie.

#### HumanN2 :

Humann2 permet d'analyser des données de séquençage métagénomique et métatranscriptomique issues d'un microbiote. Cela, afin de déterminer la présence ou l'absence, mais aussi l'abondance, de voies métaboliques ainsi que de familles de gènes au sein de ce microbiote.

Neuf *wrappers* sont liés à cet outil cependant nous n'avons pu produire de données de test que pour six d'entre eux.

#### humann2\_join\_tables :

Permet de fusionner plusieurs fichiers contenant des tables de gènes ou de chemins en un seul.

L'outil prend un nombre indéfini de fichiers en entrée et fournit un fichier en sortie. Il ne demande pas de paramètres.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré deux fichiers d'entrée et un fichier de sortie.

Lors du développement du test de cet outil, nous avons fait face à un problème particulier. En effet, lorsque l'on utilise d'un outil à travers un *wrapper*, on observe que les fichiers d'entrée sont automatiquement renommés data# (ex : data1, data65, etc.). Le nombre après "data" dépend de la position que tient le fichier, ce qui peut varier entre les tests. Or l'outil humann2\_join\_tables fusionne différents fichiers afin d'obtenir un tableau, comme le Tableau 2. Dans notre exemple l'outil a fusionné un fichier "humann2\_Abundance" et un fichier "humann2\_Coverage".

Cependant, si les fichiers ne disposent pas d'en-tête comme celle présente sur le tableau 3 (« humann2\_Abundance ») alors l'outil va prendre le nom du fichier (un data#) pour nommer la colonne correspondante. Ce qui entraîne une erreur, étant donné que les data# varient à chaque lancement de test et donc que l'on observe une différence au niveau des en-têtes des fichiers.

Ne disposant pas, à l'origine, de fichier avec un en-tête, il a fallu pratiquer de nombreux tests à l'aveugle avant de trouver l'origine de l'erreur.



#### humann2\_reduce\_table :

Diminue la taille d'une table (contenue dans un fichier) en triant les lignes, selon la fonction sélectionnée.

L'outil requiert un fichier d'entrée, prend en compte deux paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé un test et généré un fichier d'entrée et un fichier de sortie.

#### humann2\_regroup\_table :

Fusionne les tables d'un même fichier afin de n'en avoir plus qu'une.

L'outil requiert un fichier d'entrée, prend en compte trois paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test, écrit un second test et généré un fichier d'entrée ainsi que deux fichiers de sortie.

#### humann2\_rename\_table :

Renomme les différentes tables d'un fichier.

L'outil requiert deux fichiers d'entrée, prend en compte deux paramètres et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons créé un test et généré deux fichiers d'entrée et un fichier de sortie.

#### humann2\_renorm\_table :

Normalise le contenu du fichier qui lui est confié, soit en copies par millions, soit en abondance relative.

L'outil requiert un fichier d'entrée, prend en compte un paramètre et fournit un fichier en sortie.

Pour le *wrapper* lié à cet outil, nous avons généré un fichier d'entrée et un fichier de sortie.

#### humann2\_split\_table :

Sépare les tables rassemblées dans un seul fichier avec "*humann2\_join\_tables*".

L'outil requiert un fichier en entrée et fournit un nombre indéfini de fichiers en sortie. Il ne demande pas de paramètres.

Pour le *wrapper* lié à cet outil, nous avons réécrit un test et généré deux fichiers d'entrée et un fichier de sortie.

On retrouve pour cet outil la même particularité que pour "*compare\_humann2\_output*". Il a donc fallu effectuer des tests en aveugle pour déterminer les paramètres corrects de la "collection".