Rapport de stage

Développement de tests automatiques pour des outils dans l'environnement Galaxy

Stage du 11 avril 2016 au 17 juin 2016

Etudiant : Thomas Eymard
Tuteur de stage : Bérénice Batut

Université d'Auvergne, IUT de Clermont Ferrand, Campus universitaire d'Aurillac,
Département génie biologique, 2nd année de DUT Bio-informatique

Équipe de recherche EA CIDAM
Facultés de Médecine et de Pharmacie, CBRV - RdC et 5ème étage BP 38
28, Place Henri-Dunant 63001 Clermont-Ferrand

Sommaire

Remerciements

Résumé

J'ai effectué mon stage au sein de l'Université d'Auvergne, dans le laboratoire de recherche EA CIDAM. Le laboratoire étudie, le microbiote intestinal, au travers d'analyses **métagénomiques**.

Pour traiter ce type de données, de nombreux **outils bioinformatiques** sont développés par exemple QIIM ou BLAST. Cependant, nombre de ces outils doivent être lancés en **ligne de commande**. Cela peut poser problème à la majorité des biologistes.

C'est dans ce cadre-là qu'ASaiM est développé. Il s'agit d'un ensemble d'outils basé sur **Galaxy**, une application web fournissant une interface à l'utilisation d'outils en ligne de commande. Cette interface se fait grâce à des ***wrappers***, des outils codés en XML permettant d'exécuter n'importe quel outil en ligne de commande. Des outils qu'il faut **tester** pour certifier leur **qualité**.

Lors de ce stage, nous avons produit un protocole permettant de développer des **tests automatiques** pour les *wrappers* des différents outils d'ASaiM. Nous avons ainsi fourni des tests valides pour les *wrappers* de quinze outils sur les dix-huit outils initialement prévu. La majorité des outils a donc pu être publiée.

Abstract

I have performed my internship in the Auvergne University, in the research laboratory EA CIDAM. The laboratory is studying the intestinal microbiota with **metagenomics** analysies.

To process this datas, lots of **bioinformatics tools** are developed, like QIIM or BLAST. However lots of this tools must be used with **command line**. This could be problematic for a large majority of biologists.

That is in that context, ASaiM is developed. There is a framework based on **Galaxy**, a web application providing an interface for use command line tools. Which it is due to **wrappers**, XML tools allowing to execute all command line tools. We must **test** this tools to certify their **quality**.

During this internship, we produced protocol to developped **automatic tests** for ASaiM tools' wrappers. Thus, we provided valid wrappers for fifteen tools on the heighteen expected tools. So we have published most of the tests.

Introduction

Le laboratoire EA CIDAM est l'une des unités de recherche de l'Université d'Auvergne. Il a pour objectif de comprendre, évaluer et analyser, dans l'environnement digestif, différentes situations physio-pathologiques. Le travail s'oriente principalement autour de l'analyse du microbiote intestinal, l'ensemble des micro-organismes peuplant le tube digestif. La majorité de ces micro-organismes n'étant pas cultivable. Afin d'étudier le comportement et l'évolution de ce microbiote, il faut pratiquer une étude générique des gènes issus de l'ensemble des génomes des populations de micro-organismes du milieu. Ce type d'étude est appelé métagénomique.

La métagénomique fournit une quantité très importantes de données à analyser. Ces données sont impossible à analyser manuellement. Pour traiter ce type de données, de nombreux outils bioinformatiques sont développés. Parmi les outils les plus connus, on retrouve QIIM ou encore BLAST mais il en existe des milliers. Cependant, ces outils, la grande majorité de ces outils bioinformatiques, ne disposent pas d'interface graphique et doivent être lancés en ligne de commande. Il est donc nécessaire de disposer de quelques notions d'informatiques pour les utiliser. Ce qui n'est pas le cas de la majorité des biologistes.

C'est dans ce cadre-là qu'ASaiM (Auvergne Sequence analysis of intestinal Microbiota) est développé. Il s'agit d'un framework (un ensemble d'outils) basé sur Galaxy, une application web fournissant une interface graphique intuitive pour l'utilisation d'outils en ligne de commande. Cependant, cette interface se fait grâce à des wrappers, des outils codés en XML permettant d'exécuter n'importe quel outil en ligne de commande. Ces outils ne sont généralement pas fournis avec l'outil bioinformatique auquel ils sont liés. Ils doivent donc être développés puis testés pour pouvoir intégrer l'outil bioinformatique dans Galaxy.

C'est dans ce cadre-là que s'inscrit ce stage. Son objectif est de mettre en place des tests automatiques des wrappers développés dans le cadre d'AsaiM. Ces tests doivent être effectués afin de garantir que tout changement dans le wrapper n'impacte pas son comportement et donc de garantir une certaine qualité des wrappers. Dans l'objectif d'optimiser le travail, il faudra mettre en place un protocole de développement de test.

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. Elle rassemble en son sein sept composantes, trois écoles doctorales trois facultées et Institut Universitaire Technologique (IUT) disposant de quatre départements.

L'université dispose aussi de 22 laboratoires de recherche dont l'EA CIDAM (Conception, Ingénierie et Développement de l'Aliment et du Médicament) où j'ai effectué mon stage.

1.2 Conditions du stage

L'EA CIDAM est localisée sur le site de la faculté de Médecine et de Pharmacie de Clermont-Ferrand. Le laboratoire est réparti entre le rez-de-chaussée et le 5e étage du Centre Biomédical de Recherche et de Valorisation (CBRV), associant des équipes impliquées dans la biopharmacie et dans la physiologie et la simulation de l'appareil digestif. Il dépend administrativement de l'Université d'Auvergne et plus précisément de la Faculté de Pharmacie de Clermont-Ferrand. Il est issu de la réunion de deux équipes de recherche, l'équipe de Recherche Technologique CIDAM (ERT-CIDAM) et l'équipe Génomique Intégrée des Interactions Microbiennes (GIIM) issue du Laboratoire Microorganismes : Génome et Environnement (LMGE).

L'équipe de recherche actuelle est composée, tout d'abord, de treize enseignants chercheurs, dont l'activité d'enseignement est répartie entre la faculté de Pharmacie, le cursus « Nutrition » de l‘UdA et l'IUT de Clermont-Ferrand, sur les sites de Clermont-Ferrand et d'Aurillac. Le reste du personnel est formé d'une secrétaire à mi-temps, d'un ingénieur de recherche et d'une technicienne. Le laboratoire abrite trois post-doctorants, dont Bérénice Batut, ma tutrice de stage et plusieurs étudiants réalisant leur doctorat de sciences. Enfin, des étudiants en formation sont également présents au moment de ce stage, provenant de différentes filières (M2, M1 et DUT Génie Biologique).

Actuellement dirigé par le professeur M. Alric, le laboratoire travaille notamment sur la compréhension, l'évaluation et l'analyse, dans l'environnement digestif, de différentes situations physio-pathologiques. Ces situation sont liées au vieillissement, à la présence de bactéries pathogènes (en particulier d'Escherichia coli entérohémorragiques), ou encore à celle de produits toxiques et de xénobiotiques.

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

2 Cadre du projet

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, la plupart fonctionnent en ligne de commande, ce qui peu poser problème à de nombreux biologistes. Il est nécessaire de trouver une voie détournée pour utiliser ces outils. Ainsi ASaiM est développé afin de faciliter l'utilisation de ces outils grâce à une interface. C'est un framework basé sur Galaxy, une application web fournissant une interface à ces outils d'analyse.

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, la possibilité de lancer des outils de bioinformatique, qui fonctionnent normalement en ligne de commande. En effet, l'interface graphique de Galaxy est intuitive et facilement utilisable (Figure 1).

Ensuite, Galaxy est une application ouverte et *open source*, cela signifie que le code source est distribué sous une licence, considérée libre de diffusion par l'OSI, permettant, si le développeur principal le souhaite, à quiconque de lire, modifier ou redistribuer ce logiciel.

L'intérêt principal de Galaxy est d'effectuer des analyses complètes et reproductibles grâce à son gestionnaire de *workflows*. Il exécute automatiquement des séries d'opérations répétitives en associant graphiquement des outils entre eux. Nous pouvons aussi exécuter les outils manuellement, puis extraire le *workflow* à partir de l'historique des opérations effectuées.

Cette liberté d'utilisation et d'installation ainsi que le gestionnaire de *wrokflow* ont favorisés la formation d'une communauté très active autour de Galaxy et de nombreux outils sont soumis par des personnes extérieures à l'équipe de développement .

Cependant, les outils bioinformatiques ne sont pas directement utilisables par Galaxy. En effet l'interface se fait grâce à des wrappers. Ce sont des outils qui permettent d'éxécuter n'importe que outil en ligne de commande. Ils servent d'interface entre l'utilisateur et la ligne de commande qui lance l'outil. Les wrappers gèrent les

fichiers d'entrée et de sortie et les paramètres nécessaires au fonctionnement de l'outil bioinformatique.

Les wrappers utilisés dans Galaxy sont des outils codés en langage XML, un langage qui permet de décrire et de structurer des données à l'aide de balises. 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.xml qui gère l'installation des dépendances nécessaires à l'outil. Et un .shed.yml qui gère les métadonnées de l'outil dans le ToolShed

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

- Les informations générales sur l'outil : son nom, son identifiant, sa version et sa description.
- La section `<requirements>` fournit les informations sur les dépendances nécessaires à l'outil (nom et version).
- La partie `<command>` fait le lien 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.
- C'est dans la section `<input>` que l'on définit les paramètres d'entrée de l'outil qui sont affichés via l'interface graphique. On y précise par exemple le nom que l'on veut afficher ou encore le type de ces paramètres.
- 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. Il est aussi possible de modifier son format. Toutes les sorties de l'outil doivent être référencées dans cette section.
- La partie `<tests>` permet de faire du test automatique et garantir la fiabilité du wrapper en fournissant les données d'entrée. C'est la section sur laquelle se concentre la partie principale du travail effectué durant le stage.
- La section `<help>` permet d'afficher des informations sur l'outil.
- C'est dans la section `<citations>` qu'on liste les différentes citations de l'outil

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 ToolShed. Cette banque de donnée fonctionne selon le même principe que les plate-formes de téléchargement mobile. La totalité des logiciels présents sont mis a disposition et téléchargeables (Figure 2). C'est là que les développeurs extérieurs à l'équipe de développement de Galaxy soumettent les wrappers qu'ils ont développés. Lors du dernier recensement effectué au sein du ToolShed principal, le 15 Mai 2016,

3904 wrappers étaient disponibles.

Cependant, avant d'être publié dans le ToolShed et afin de garantir son comportement, un wrapper, comme tout logiciel ou programme informatique a besoin d'être testé.

2.2 Test logiciel

L'objectif du test logiciel est d'exécuter le programme concerné dans l'intention d'y trouver des défauts. L'objectif n'est pas de démontrer que le programme ne contient plus d'erreur. En effet si nous développons des tests dans l'objectif de démontrer le fonctionnement de l'outil, nous risquons de fournir des fichiers d'entrée, dont nous savons qu'ils fonctionnent, et ne pas produire d'autre tests afin de vraiment vérifier le fonctionnement de l'outil.

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 à des sorties attendues.

Il existe de nombreux types de tests différents (unitaires, intégration, performance, etc.). 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. Il consiste dans le fait de tester la totalité d'un programme en reproduisant des « conditions réelles d'utilisation ». C'est ce type de test que nous avons pratiqué pour nos wrappers.

3 Protocole de développement de tests automatiques des wrappers dans Galaxy

Afin de garantir la qualité des données fournies par un wrapper, il est fortement conseillé de lui fournir des tests garantissant son fonctionnement et ce de façon automatique. Nous avons donc mis en place, durant ce stage, le protocole suivant (Figure 4). Tout au long de cette partie, nous citerons l'outil GraPhlan comme exemple.

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. Avant toute chose,pour développer correctement un wrapper et ses test, il faut prendre connaissance du fonctionnement de l'outil. Il faut donc effectuer des recherches sur la documentation de l'outil. Nous devons également installer localement l'outil. Premièrement pour avoir accès à la documentation fournie avec l'outil via la ligne de commande. Puis pour pouvoir produire les fichiers de tests.

Pour l'outil pris en exemple, nous nous sommes principalement documentés sur la plate-forme où se trouve le code source de GraPhlan (Figure 5). Ainsi, l'outil nécessite un fichier d'entrée et quatre paramètres pour fonctionner. Il produit un fichier en sortie.

Voici par exemple une ligne de commande permettant de lancer GraPhlan :

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

Il est possible de scinder cette ligne de commande en trois parties distinctes qui sont les suivantes.

On retrouve tout d'abord le nom du fichier permettant de lancer l'outil : `graphlan.py`

Puis les fichier d'entrée er de sortie de l'outil : `input.txt` et image.png

Viennent ensuite les quatre paramètres nécessaires au fonctionnement de l'outil :

- Le paramètre `--format png` reseigne le format du fichier de sortie. Ici nous souhaitons qu'elle soit au format pnj.
- Le second paramètre, `--dpi 100`, renseigne la définition ou qualité de l'image. Ici elle devra être de 100 points par pouce(mesure).
- `--size 7` quand à lui renseigne la taille de l'image en sortie en pouces. Ici nous souhaitons que la taille soit de 7 pouces par 7 pouces.
- Enfin `--pad 2` 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.

Dans le cas de GraPhlan, 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, la classe, l'ordre, la famille, le genre et l'espèce). Le fichier de sortie, quant à lui, est une image sur laquelle est représentée un arbre phylogénétique circulaire.

Généralement les données de test (entrées, paramètres et sorties) ne sont pas fournies. Il faut alors générer les fichiers et les paramètres correspondants. Dans le cas de GraPlan, le fichier d'entrée correspond à un fichier de sortie de graphlan_annotate,

venant avec GraPhlan. Le fichier de sortie de GraPhlan doit aussi être généré en lançant l'outil avec le fichier d'entrée de test et les paramètres choisis.

3.2 Modification du *wrapper*

Après avoir pris en main l'outil et son fonctionnement, il faut éditer le *wrapper* pour générer ou réécrire un ou plusieurs tests. Parfois, lors de la mise en place des tests, il faut même modifier le corps du *wrapper* pour corriger les erreurs détectées.

Dans le cas de GraPhlan, nous avons édité la section `<tests>` pour fournir les données d'entrées citées précédemment (voir Figure 7).

Cependant, suite à l'apparition d'erreurs lors de la phase de test, le *wrapper* a dû être corrigé en chageant la version de la dépendance *"graphlan"* (de 0.9.7 à 1.0.0) .

3.3 Tests locaux

Avant l'intégration pour l'automatisation des tests, les tests mis en place sont testés localement. Pour tester un *wrapper* localement, la meilleure méthode est d'utiliser un outil nommé *Planemo*. Il s'agit d'un outil développé par la communauté Galaxy pour faciliter le développement des *wrappers* pour Galaxy. *Planemo* dispose de trois commandes directement liées au test des *wrappers* :

- La commande `$ planemo lint` permet de tester le fichier XML. Elle sert notamment à repérer les erreurs de frappe ou les oublis (par exemple, une balise ouverte, mais qui n'a pas été fermée).
- Lancer le test `$ planemo shed-lint` permet de vérifier que le *wrapper* est correct et prêt à être déployé sur le ToolShed.
- Effectuer la commande `$ planemo test` lance l'installation d'une instance de Galaxy virtuelle et nue, installe le *wrapper* et ses dépendances avant de lancer les tests fonctionnels décrits dans la section <*test*>.

Pour tester l'outil GraphLan, la commande utilisée est :

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

Dans cette commande, nous utilisons plusieurs options. Nous allons présenter leur

utilité.

- `--conda_dependency_resolution` fait en sorte que Galaxy n'utilise *conda*, un gestionnaire d'outil, que pour gérer les dépendances
- `--conda_prefix $HOME/conda` donne le chemin de conda
- `--galaxy_branch release_16.04` donne la version de Galaxy à utiliser
- `--galaxy_source $GALAXY_REPO` fourni le lien pour télécharger Galaxy
- `--skip_venv` empèche la création d'un environnement virtuel pour conserver la configuration de l'environnement déjà existant
- `tools/graphlan/` est le chemin vers le *wrapper*

Utiliser Planemo pour effectuer les tests engendre un important gain de temps. En effet, la commande de test gère tout le test automatiquement :

- déploiement d'une instance de Galaxy
- installation de l'outil et du wrapper à tester
- lancement du test, comparaison entre les sorties générées et celles attendues
- retour des erreurs détectées

Si le test échoue, il faut retourner éditer le *wrapper* pour corriger la ou les erreur(s). La commande fournit également des informations sur les raisons de l'échec. Si le test est passé, l'outil pourra être publiable.

3.4 Intégration dans le répertoir de travail Git via Github

Le gestionnaire de version Git et la plate-forme GitHub sont utilisés pour stocker et distribuer le code source d'ASaiM et des wrappers associés (https://github.com/AsaiM/galaxytools).

Git est un gestionnaire de version, cela signifie qu'il 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 (commentaire de toute modification).

De plus, il est possible de travailler avec des branches, c'est-à-dire, des versions parallèles du code source. Il est possible de modifier les données d'un fichier sur une branche tout en préservant l'intégrité du même fichier situé sur les autres branches, en particulier la branche principale. Il est ainsi possible de travailler à plusieurs sur une même portion de code pour le développement de fonctionnalités différentes. Il faut

ensuite fusionner le travail.

Git est un outil très utile pour le développement open source qui favorise le travail collaboratif entre programmateurs, car il permet de voir clairement le travail fait par chacun. Le code source pouvant être relu et amélioré par tout le monde. Cependant, le système traditionnel open source oblige chaque contributeur à télécharger les sources du projet et à proposer ensuite ses modifications à l'équipe du projet.

GitHub, le service web d'hébergement et de gestion de développement de logiciels utilisant Git, permet de contourner le problème grâce au principe de fork (embranchement), sur lequel il repose. En effet, toute personne « forkant » un projet devient, de facto, le leader de son projet portant le même nom que l'original (Figure 8). Grâce à cela les deux projets peuvent évoluer séparément après coup sur le même principe que les branches de Git. De plus, GitHub est centré vers l'aspect social du développement. Il offre de nombreuses fonctionnalités habituellement retrouvées sur les réseaux sociaux comme les flux ou la possibilité de suivre des personnes ou des projets.

Ensuite il est possible de créer des issues, des commentaires permettant de signaler la présence d'une erreur au sein du code source. Il est possible de commenter ces issues et de faire des échanges autour, par exemple pour discuter de la façon de corriger cette erreur..

La principale fonctionnalité de GitHub, est la Pull Request (PR ; Figure 9). Toutes les autres fonctionnalités vues auparavant gravitent autour de celle-ci. En effet, elle permet à n'importe qui, ayant auparavant fait un fork du projet, de proposer des modifications à apporter au code source. On voit sur ces PR la totalité des modifications apportées au code. Tout comme avec les issues, il est possible de commenter les modifications proposées, de faire des échanges autour. De plus, GitHub permet la mise en place d'une intégration continue pour n'importe quelle PR, cela afin de vérifier les modifications proposées.

3.5 Intégration continue

L'intégration continue est un ensemble de pratiques consistant à vérifier, à chaque modification du code source, que les modifications proposées n'entraient pas de régression de l'outil. Le principal objectif de l'intégration continue est de détecter les problèmes d'intégration le plus tôt possible lors du développement.

Pour cela, il faut tout d'abord que le code source soit partagé comme par exemple avec les fork de GitHUb. Ensuite, il faut mettre en ligne les modifications le plus fréquemment possible. Nous l'avons fait avec les Pull Requests, actualisées après les modifications. Enfin il faut développer des tests d'intégration afin de valider ou non les

modifications apportées. Lors du stage, les tests d'intégration sont ceux qui ont été développés et testés manuellement auparavant.

Pour le bon fonctionnement de l'intégration continue, un logiciel ,permettant de gérer automatiquement cette dernière, est requis. Lors du stage nous avons développé une instance de TRAVIS CI pour gérer l'intégration continue. TRAVIS CI est un service développé spécifiquement pour gérer l'intégration continue des projets GitHub. En effet la plate-forme l'informe à chaque mise à jour des Pull Request et TRAVIS CI lance automatiquement les tests. Il est configuré par un fichier .travis.yml. Ce fichier spécifie notamment le langage de programmation à utiliser, l'environnement à mettre en place et les tests souhaités. Après les tests il averti automatiquement le développeur (de la façon pour laquelle il a été programmé).

L'intégration continue permet notamment de tester immédiatement et automatiquement les modifications. Elle envoie une notification, dès la fin des tests, en cas d'erreur. Elle est toujours disponible pour un test, une démonstration ou une distribution. Ainsi elle permet de gagner beaucoup de temps lors de la phase de test.

Après avoir passé les tests de l'intégration continue, les wrappers sont « mergés », c'est-à-dire fusionnés avec le code source. Ils ont ensuite subi d'autres vérifications ainsi qu'une mise en forme. Cependant, les actions effectuées après l'intégration continue ne rentraient pas dans le cadre du travail effectué lors du stage.

4 *wrappers* d'outils testés

Lors de ce stage, nous avons produit des tests pour des wrappers. Un certain nombre de cas particuliers ont été rencontrés lors de la production des tests. Nous avons ainsi intégré de nombreux outils à ASaiM (Tableau I).

4.1 Outils de Manipulation

L'outil *extract_min_max_lines* va extraire les lignes dont la valeur maximale (ou minimale suivant le choix de l'utilisateur) se trouve sur la colonne sélectionnée. 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, quatre tests ont été créés et quatre fichiers de sortie (un pour chaque test) ont été générés.

L'outil *normalize_dataset* permet de normaliser chaque colonne ou chaque ligne et présente les résultats sous forme de proportion ou de pourcentage. Il requiert un fichier

d'entrée, prend en compte deux paramètres et fournit un fichier en sortie. Pour le wrapper lié à cet outil, trois tests ont été créés et fichier d'entrée ainsi que trois fichiers de sortie ont été générés.

L'outil *convert_extract_sequence_file* extrait des informations des fichiers de séquences métagénomiques. Il peut aussi convertir un fichier au format FastQ en un fichier au format FastA. Il 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, deux tests ont été créés. Cinq fichiers de sortie, deux pour le premier et trois pour le second ont été générés.

L'outil *fasta_add_barcode* ajoute un « code barre » au début de chaque séquence sur un fichier au format FastA. Il 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, un test a été créé et deux fichiers d'entrée ainsi qu'un fichier de sortie ont été générés.

4.2 Outils de prétraitement des données issues du séquençage

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.Il requiert un fichier d'entrée, prend en compte cinq paramètres et fournit deux fichiers en sortie.Pour le *wrapper* lié à cet outil, un test a été réécrit et deux fichiers de sortie ont été générés.

cd_hit_protein regroupe des séquences protéiques. Il 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 deux fichiers de sortie ont été générés.

L'outil *format_cd_hit_output* 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. Il nécessite deux fichiers en entrée, nécessite quatre paramètres et fournit un fichier en sortie. Un test a été réécrit et deux fichiers de sortie générés, pour le *wrapper* lié à cet outil.

4.3 Outils pour les anamyses métagénomiques

L'outil *MetaPhlan2* permet de déterminer la composition des communautés microbiennes (bactéries, archaes, eukaryotes et virus) à partir de données métagénomiques. Deux *wrappers* dépendent de cet outil, metaphlan2 et *metaphlan2krona*.

metaphlan2 est le *wrapper* de l'outil *MetaPhlan2*. Il nécessite trois fichiers en entrée, nécessite 10 paramètres et fournit un fichier en sortie.

metaphlan2krona permet de formater les données issues de *metaphlan2* afin qu'elles soient utilisables par un autre outil, *Krona*. Il requiert un fichier d'entrée et fournit un fichier en sortie. Il ne prend pas en compte de paramètres.

Les tests pour cet outil n'ont pas été produits, car des données de tests convenables n'ont pas pu être produites.

L'outil *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. Il requiert un fichier d'entrée et fournit neuf fichiers en sortie. Il ne nécessite pas de paramètres. Pour le *wrapper* lié à cet outil, un test a été édité et huit fichiers de sortie modifiés.

L'outil *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 du 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, un test a été réécrit et deux fichiers d'entrée et un fichier de sortie ont été générés. 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*, nous observons 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 différent fichiers afin d'obtenir un tableau, comme le Tableau II. 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 III ("*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 a 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. Il requiert un fichier d'entrée, prend en compte deux paramètres et fournit un fichier en sortie. Pour le *wrapper* lié à cet outil, un test a été créé et un fichier d'entrée et un fichier de sortie ont été générés.

humann2_regroup_table fusionne les tables d'un même fichier afin de n'en avoir plus qu'une. Il 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. Il 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, un fichier d'entrée a été généré ainsi qu'un fichier de sortie.

humann2_split_table sépare les tables rassemblées dans un seul fichier avec *hunann2_join_tables*. Il 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, un test a été réécrit et deux fichiers d'entrée et un fichier de sortie ont été générés. 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.

L'outil *compare_humann2_output* 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, un test a été créé et deux fichiers d'entrée ainsi que cinq fichiers de sortie ont été générés.

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.

L'outil *group_humann2_uniref_abundances_to_go* compare un fichier issu de *HumanN2* avec la base de données *UniRef50* afin d'obtenir les abondances des fonctions moléculaires, des processus biologiques et des comportements cellulaires. L'outil requiert trois fichiers d'entrée, prend en compte trois paramètres et fournit trois fichiers en sortie. Nous n'avons pas pu produire de tests pour cet outil suite à l'appartion d'une erreur incompréhensible lors de la phase de test manuelle.

L'outil *combine_metaphlan2_humann2* 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.

4.4 Outils pour la visualisation et les statistiques

L'outil *compute_wilcoxon_test* effectue le test de wilcoxon (statistiques) sur les données fournies. L'outil requiert un fichier d'entrée, prend en compte neuf paramètres et fournit un fichier en sortie. Nous n'avons pas pu produire de tests pour cet outil suite à l'appartion d'une erreur incompréhensible lors de la phase de test manuelle.

L'outil *export2graphlan* 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 vinght-six paramètres et fournit deux fichiers en sortie. Pour le *wrapper* lié à cet outil, nous avons réécrit un test.

L'outil *GraPhlan* 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 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. Il a fallu mettre à jour la dépendance *"graphlan"* en passant de 0.9.7 à 1.0.0.

graphlan_annotate annote 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 réécri

L'outil *plot_generic_x_y_plot* créé un diagramme en nuage de points à partir d'un fichier au format tabular. Il 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.

L'outil *plot_barplot* 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.

L'outil *plot_grouped_barplot* 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.

Conclusion

L'objectif du stage était de générer des données de test afin de finaliser la mise en place de tests « automatiques ». Cela, dans l'objectif d'apporter un gage de qualité pour les wrappers produit dans le cadre d'ASaiM. ASaiM étant développé afin de simplifier les analyses des biologistes dans le domaine de la métagénomique.

Le protocole de développement mise en place lors du stage a permis de fournir au moins un test valide pour chacun des vingt-deux wrappers validés. Cela a permis de publier quinze outils avec des wrappers testés sur le ToolShed principal.

Cependant, pour diverses raisons, nous n'avons pas pu produire de test convenable pour trois outils, qui sont *MetaPhlan2*, *group_humann2_uniref_abundances_to_go* et *compute_wilcoxon_test*. Ces tests ont dû être produits par Bérénice Batut pour pouvoir publier ces outils avec les autres. De plus, une majorité des *wrappers* ne disposent que d'un seul test. Si l'on souhaite encore améliorer la fiabilité des *wrappers*, il faudrait développer plus de tests pour ces *wrappers*.

Ce stage fut riche en enseignements. Il m'a tout d'abord permis de découvrir ce qu'était un véritable projet de développement. Il m'a aussi permis d'apprendre à gérer un développent notamment avec l'utilisation d'un logiciel de gestion de version (Git durant le stage) ou l'importance du travail collaboratif dans ce genre de projet. Ensuite, ce stage m'a fait découvrir l'importance du test logiciel et de l'intégration continue. J'ai aussi appris à développer et mettre en place des tests. Lors de ce stage j'ai découvert ce qu'était vraiment la programmation et le développement et j'ai vraiment apprécié. Cela m'a d'ailleurs convaincu de continuer dans la voie de l'informatique.

Bibliographie

Table des sigles et abréviations

CIDAM : Conception, Ingenierie et Developpement de l'Aliment et du Medicament

EA : Equipe d'Accueil

IUT: Institut universitaire de technologie

OSI : Open Source Initiative

Glossaire

Code source:

Interface:

ligne de commande:

Métadonnée: Type de donnée qui permet de caractériser et structurer des ressources numériques, telles que celles contenues dans une page web.

Métagénomique :

Microbiote :

Situations physio-pathologiques :

Anglicismes

framework : Ensemble d'outils constituant les fondations d'un logiciel informatique ou d'applications web, et destiné autant à faciliter le travail qu'à augmenter la productivité du programmateur qui l'utilisera.

Open Source : programme informatique dont le code source est distribué sous une licence permettant à quiconque de lire, modifier ou redistribuer ce logiciel. Davantage tourné vers un objectif de développement collaboratif. Open Source ne signifie pas « gratuit ». Il existe de nombreux logiciels dont le code source est propriétaire (il n’est pas permis d’y accéder, de le modifier ou de le redistribuer).

Workflows :

Wrapper :

Table des illustrations