Rapport de stage
Publication du package crisprbuilder_tb

Stephane Robin sous la direction de Christophe Guyeux et Jean-Claude Charr

2 juillet 2020

Table des matières

1

Préambule

1. Remerciements

Je tiens à exprimer mes plus sincères remerciements à Messieurs Christophe Guyeux et Jean-Claude Charr, professeurs à l'université de Franche Comté pour m'avoir guidé, conseillé et soutenu durant le déroulement de ce stage. Leurs explcations et leur disponibilité malgré un emploi du temps chargé ont largement contribué à ma compréhension du sujet.

Je voudrais également souligner la patience dont ils ont fait preuve pour m'expliquer les notions de bioinformatique que j'ai été amené à découvrir durant ce projet. Il en ressort que j'ai particulièrement apprécié de travailler sous leur direction.

2. Présentation du projet

Dans le cadre d'un travail de recherche en bio-informatique, Christophe Guyeux, Jean-Claude Charr, Christophe Sola et Guislaine Refrégier ont créé du code "brut" leur permettant d'afficher et de stocker des données relatives à une SRA [1] particulière. Ce code doit être rassemblé, réorganisé, nettoyé pour être conforme aux critères PEP [2]. Il doit être rendu fonctionnel puis plus efficace si possible. Il doit s'exécuter en utilisant un *Command Line Interface*, et doit finalement être empaqueté pour qu'un utilisateur puisse l'utiliser quelque soit sa plateforme (Linux, MacOS ou Windows) une fois installé. Pour cela, il est nécessaire de publier ce package sur PyPI [3] et de le documenter clairement en anglais pour en faciliter l'utilisation.

Lorsque nous avons défini les objectifs du stage, il est immédiatement ressorti l'importance de lui donner un caractère pratique, qui pourrait éventuellement être transposable dans le milieu professionnel. La reconstruction du programme à partir d'éléments de code, la nécessité de créer localement un environnement spécifique de travail et d'installer automatiquement cet environnement sur les ordinateurs des utilisateurs, la possibilité de faire fonctionner le package sur différentes plateformes, les différents tests effectués, la création et publication du package sont autant de compétences pratiques utiles, auxquelles il faut se confronter pour bien en comprendre les difficultés. Par ailleurs, la construction de ce package a nécessité de comprendre le fonctionnement de termes clés en bioinformatique afin de pouvoir travailler dans ce cadre métier particulier. Ceci a conduit à une synthèse de l'état de l'art fortement orientée vers la biologie et plus particulièrement le *CRISPR*.

1. **SRA :** Sequence Read Archive
2. **PEP :** Python Enhancement Proposals
3. **PyPI :** Python Package Index

2

Etat de l'art

1. Evolution de la branche humaine de la tuberculose

1.1. Diversité génétique de la tuberculose

Une bactérie survit d'autant mieux qu'elle est capable de s'adapter à son environnement au travers de mutations génétiques. Le polymorphisme génétique est à l'origine de la diversité génétique et correspond, dans le cas de notre étude, à des variations de séquences d'ADN entre différentes souches de *M. tuberculosis*. Ces variations sont dues à des mutations successives au cours de l'évolution de la bactérie, et elles permettent l'analyse phylogénique de *M. tuberculosis*. Il existe plusieurs formes de polymorphisme, le polymorphisme chromosomique lié à un changement du nombre de chromosomes ou de leurs structures, le polymorphisme d'insertion, de délétion et d'inversion qui provoquent un changement spécifique de certaines séquences du génome, et le polymorphisme nucléotidique SNP *Single Nucleotide Polymorphism* lié au changement d'une seule paire de bases [1] du génome de *M. tuberculosis*. Nous allons détailler plus particulièrement ce dernier cas.

Certaines mutations n'ont aucun impact évolutif sur *M. tuberculosis*. En revanche, des changements fonctionnels peuvent avoir lieu lorsque ces mutations entraînent des modifications d'acides aminés dans les régions codantes, cela peut-être le cas lors d'une adaptation à l'environnement ou lors d'une nouvelle forme de résistance aux antibiotiques. Les SNPs synonymes ne changent pas la séquence de protéine, ainsi la substitution d'un codon [2] par un autre codon peut engendrer le même acide aminé. Au contraire, les SNPs non-synonymes changent la séquence de protéine, et engendrent donc l'incorporation d'un acide aminé différent. Chez *M. tuberculosis*, les SNPs sont peu sujets à des phénomènes d'homoplasie [3](seuls 1,1 % des SNPs sont homoplasiques), ce qui suggère que la structure de *M. tuberculosis* favorise les clonages plutôt que les recombinaisons entre branches. Pour de tels organismes clonaux, l'identification de mutations homoplasiques est un excellent moyen de déterminer les différentes souches bactériennes, et ainsi de procéder à des études phylogéniques [4] et de classification.

1. **Paire de bases** : appariement de 2 bases nucléiques situées sur 2 brins complémentaires d'ADN, reliées par des ponts d'hydrogène.
2. **Codon** : ensemble composé de trois nucléotides consécutifs spécifiant l'incorporation d'un acide aminé déterminé. Le code génétique est ainsi lu trois nucléotides par trois nucléotides.
3. **Homoplasie** : similitude de caractères chez différentes espèces, qui ne provient pas d'un ancêtre commun, mais peut par exemple provenir d'une adaptation à l'environnement. Diffère de l'homologie qui est une similitude de caractères observée chez deux espèces différentes, provenant de l'héritage d'un ancêtre commun.
4. **Phylogénie** : étude des liens entre espèces apparentées, permettant de retracer les principales étapes de l'évolution des organismes depuis un ancêtre commun.

1.2. Co-évolution de la tuberculose avec l'homme moderne

Le développement des maladies s'adapte à la densité de population concernée. En effet, auprès d'une foule dense, les infections se répandent plus largement et deviennent plus virulentes, alors qu'auprès d'une population moins importante, elles ont une croissance plus faible, laissant parfois place à des périodes où les infections restent latentes.

Une période charnière dans l'histoire de l'humanité est la transition démographique du Néolithique, qui a vu il y a 10 000 ans, suite à l'apparition de l'agriculture et de l'élevage, un accroissement de la population, favorisant la naissance de nombreuses maladies. Les maladies humaines plus anciennes se développaient auprès de populations moins denses et produisaient des phases chroniques de latence et de réactivation permettant aux populations infectées de survivre.

Nous allons voir que la tuberculose conjugue ces deux modèles de maladie.

L'étude phylogénique de Comas et al.[1] se base exclusivement sur l'étude du génome [5] complet de toutes les lignées connues de *M. tuberculosis* en utilisant les SNPs comme marqueurs pour construire les relations entre les différentes branches. Les résultats obtenus rejoignent de précédentes études effectuées à partir d'autres marqueurs, et confirment l'existence de sept principales lignées de tuberculose. On remarque en particulier que plusieurs branches d'origine animale se sont regroupées avec la lignée 6 d'Afrique de l'Ouest, et que les lignées modernes 2 d'Asie de l'Est, 3 d'Asie Centrale et 4 d'Europe ont des origines proches.

L'analyse phylogénique de Comas et al. corrobore la conjecture selon laquelle la tuberculose est originaire d'Afrique. Par ailleurs, s'appuyant sur les origines africaines de l'espèce humaine, cette étude cherche également à déterminer l'ancienneté de l'association entre la tuberculose et son hôte humain. Pour cela, l'analyse des divergences des génomes de la tuberculose est comparée à celle d'une arborescence génétique déjà établie à partir de mitochondries [6] de l'être humain.

Les similitudes relevées montrent que la tuberculose a infecté les premiers hommes d'Afrique. Pour aller plus loin, l'étude de Comas et al. a tenu compte de trois dates importantes dans l'évolution biologique de l'être humain qui ont été reportées sur l'analyse phylogénique de *M. tuberculosis* des lignées 5 et 6 d'Afrique de l'Ouest : l'émergence de l'homo sapiens correspondant au MTBC-185 [7], l'émergence de l'haplogroupe [8] mitochondrial de la lignée 3 chez l'homme correspondant au MTBC-70, et le début de la transition démographique du Néolithique correspondant au MTBC-10.

La branche MTBC-185 suggère l'apparition de mutations à partir de lignées africaines il y a 174 000 ans, c'est à dire que la dispersion de la tuberculose précèderait celle de l'homo sapiens.

La branche MTBC-70 révèle des corrélations avec l'histoire de l'humanité telle qu'elle a pu être décrite par l'archéologie, en montrant l'apparition des sept différentes lignées de tuberculose :

- il y a 73 000 ans, apparition des lignées 5 et 6 correspondant à une première migration humaine importante vers l'Afrique de l'Ouest,
- il y a 67 000 ans, apparition de la lignée 1 correspondant à une migration humaine importante autour de l'Océan Indien,
- il y a 64 000 ans, apparition de la lignée 7 concernant une population qui est restée en Afrique ou

5. **Génome** : ensemble de l'information génétique d'un organisme. Par extension, le génome se réfère aussi au support physique de cette information génétique, la macromolécule d'ADN. L'annotation des gènes est le processus permettant d'identifier l'emplacement des gènes dans l'ADN, de déterminer leurs fonctions et leurs possibles interactions.

6. **Mitochondrie** : centrale énergétique des cellules qui contribue à la production d'ATP.

7. **MTBC** : Mycobacterium Tuberculosis Complex.

8. **Haplogroupe** : groupe possédant les mêmes caractères génétiques et partageant un ancêtre commun suivant une mutation SNP.

est revenue en Afrique après une première migration,
- il y a 46 000 ans, apparition de la lignée 4 correspondant à une migration humaine importante vers l'Europe,
- il y a 42 000 ans, apparition des lignées 2 et 3 correspondant à une migration humaine importante vers l'Asie de l'Est et l'Asie Centrale.

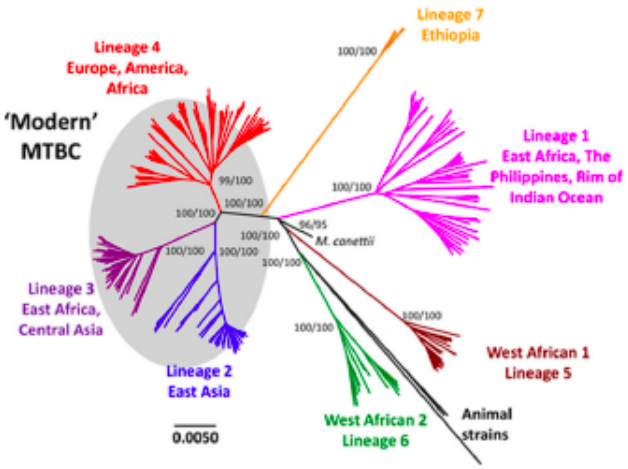


Figure 1 – Phylogénie du génome complet de MTBC, d'après *Out-of-Africa migration and Neolithic coexpansion of Mycobacterium tuberculosis with modern humans*

Dans tous les cas, la tuberculose aurait infecté l'espèce humaine et évolué conjointement avec elle depuis 70 000 ans, mais son apparition serait antérieure à la transition démographique du Néolithique.

La base de données de tuberculose étudiée de façon probabiliste par Comas et al.[1] montre que le Néolithique a fortement contribué à l'expansion de la maladie il y a 10 000 ans grâce à l'augmentation de la densité de population et à la probabilité de co-infection avec d'autres maladies également dépendantes de la densité de population. La possibilité pour la tuberculose de muter d'une variété animale vers une variété humaine n'est en revanche pas retenue par Comas et al. En effet, l'analyse phylogénique de la tuberculose montre que les branches humaines ont divergé des branches animales avant le Néolithique.

Le Néolithique n'était pas la seule période où l'augmentation de la population fut importante, toutefois la concentration de population qui s'en est suivie a permis l'apparition, auprès de la tuberculose, de caractères fortement dépendants de la densité de population qu'elle affecte. Le Néolithique a donc marqué un tournant dans l'histoire de la tuberculose, qui a alors commencé à conjuguer les deux principaux modèles de maladie, d'une part dépendant de la densité de population et d'autre part s'apparentant à une infection chronique. En effet, le mode de transmission aérosol de la tuberculose s'est parfaitement adapté aux foules, et elle a montré à travers les âges des périodes de latence et de réactivation.

Il faut donc considérer que la co-existence de la tuberculose avec l'espèce humaine depuis des milliers d'années a conduit la maladie à s'adapter aux changements du génome humain et inversement. Les prochaines études sur la tuberculose devraient donc se concentrer sur des génomes complets de la tuberculose et de l'être humain choisis en rapport à leurs associations.

En particulier, la tuberculose a dû s'adapter aux autres infections ayant touché l'espèce humaine, avec plus ou moins de succès. Dans cet ordre d'idée, une étude récente de Perry S. et al.[7, 8] suggère que l'infection d'un organisme par *Helicobacter Pylori* pourrait protéger de la tuberculose sous sa forme active. A contrario, nous ne savons pas si la tuberculose latente pourrait protéger contre les

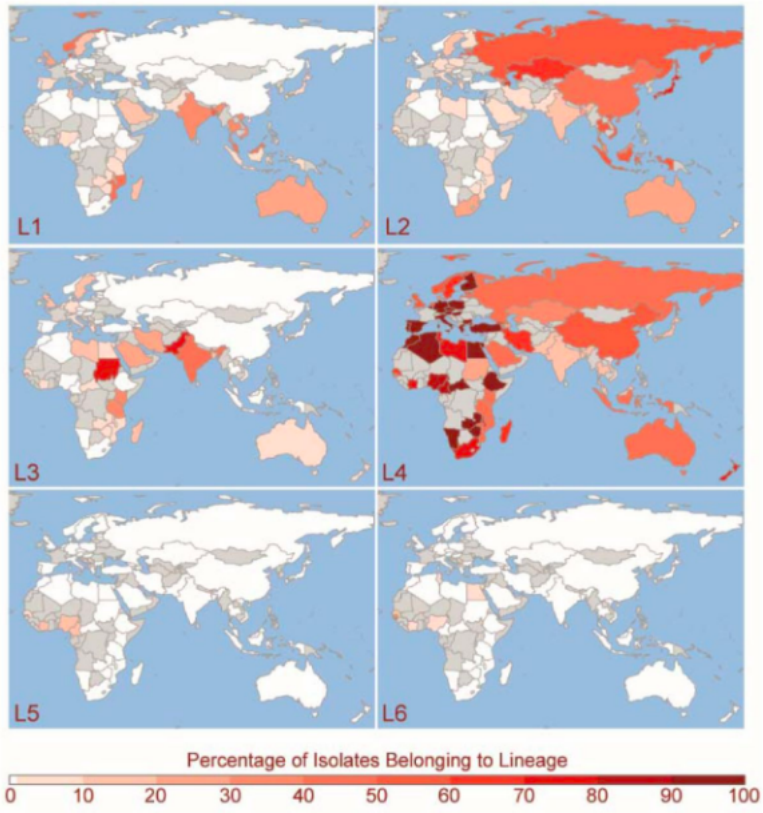


Figure 2 – Distribution géographique des lignées 1 à 6, d'après *Lineage specific histories of Mycobacterium tuberculosis dispersal in Africa and Eurasia*

ulcères et les cancers de l'estomac causés par *Helicobacter Pylori*.

2. Le développement de souches résistantes aux antibiotiques

2.1. L'expansion de la lignée 4 de *M. tuberculosis*

La lignée 4 de *M. tuberculosis* est la plus répandue de par le monde et pour cette raison a fait l'objet de nombreuses publications. Brynildsrud O.B. et al.[3] utilisent des méthodes d'analyse discrète et une approche bayésienne[9] en phylogénie moléculaire pour obtenir de manière formelle l'évolution phylogéographique de la lignée 4 de *M. tuberculosis*. Ils estiment que le plus récent ancêtre commun de la lignée 4 est apparu en Europe en 1096 après JC. Si on considère l'Europe en tant que continent au sens large, cela ne contredit pas les résultats de O'Neil M.B. et al.[14] qui estiment l'origine de la lignée 4 autour de la méditerranée.

L'analyse phylogéographique de Brynildsrud O.B. suggère que les premières vagues de migration de la lignée 4 hors d'Europe se sont déroulées au début du 13ème siècle vers l'Asie du Sud-Est. Il est également possible d'établir une correspondance entre la structure des isolats[10] Vietnamiens et l'époque de l'expansion coloniale française en Indochine au 19ème siècle.

Les vagues suivantes de migration de la lignée 4 se sont dirigées vers l'Afrique de l'Ouest au 15ème

9. **Approche bayésienne** : méthode probabiliste basée sur le calcul des probabilités postérieures des arbres phylogéniques par la combinaison d'une probabilité antérieure avec la fonction de vraisemblance
10. **Isolat** : fragment d'organisme qu'on a isolé à des fins d'examen histo-pathologiques ou pour être cultivé in vitro.

siècle, puis vers Afrique de l'Est et du Sud au 17ème siècle. Les échanges continus avec le Portugal dès le 15ème siècle ont favorisé la dispersion de la maladie, ce qui a été renforcé plus tard par la colonisation française de l'Afrique de l'Ouest. Ces échanges avec les populations européennes ont prévalu à une transmission locale de la tuberculose jusqu'au 19ème siècle.

La transmission de la maladie en Amérique date, elle aussi, du 15ème siècle avec la colonisation du continent, mais il faudra attendre le 17ème siècle pour voir l'explosion de la maladie en Amérique du Sud. Ce retard dans l'évolution de la maladie par rapport à la branche africaine peut s'expliquer par le taux de mortalité élevé des populations aborigènes au contact des européens.

La première migration interne de la maladie en Afrique date de l'Empire Zulu au 19ème siècle et se dirigeait vers le Nord et l'Est africain.

Ainsi, Brynildsrud O.B. et al. montrent que la dispersion de la lignée 4 est essentiellement liée à l'expansion coloniale européenne en Afrique et en Amérique entre le 17ème et le 19ème siècle.

2.2. L'adaptation de la lignée 4 pour devenir résistante aux antibiotiques

Nous avons déjà vu que la tuberculose a su s'adapter à l'évolution géographique de l'humanité en suivant les différentes migrations humaines pour créer de nouvelles lignées ou de nouvelles souches. Il apparaît que la tuberculose est également capable de suivre l'évolution médicale de l'humanité. L'étude de Brynildsrud O.B. et al.[3] constate chez *M. tuberculosis* l'émergence croissante d'une résistance à de multiples antibiotiques entre 1960 et 2000 au travers de la phylogénie de la lignée 4.

Des mutations spontanées dans le génome de la tuberculose peuvent altérer les protéines qui sont la cible des médicaments, ce qui rend les bactéries résistantes à ces médicaments. Prenons comme exemple une mutation du gène rpoB de *M. tuberculosis*, qui code pour la sous-unité β[11] de l'ARN polymérase[12] de la bactérie. Dans la tuberculose non résistante, la rifampicine se lie à cette sous-unité β et perturbe l'élongation de la transcription de l'ARN. La mutation dans le gène rpoB modifie la séquence des acides aminés et donc de la sous-unité β. Dans ce cas, la rifampicine ne peut plus se lier à la sous-unité β de l'ARN et empêcher la transcription. La bactérie est devenue résistante. C'est bien le cas de la tuberculose, qui est considérée aujourd'hui comme une maladie résistante aux antibiotiques.

Une souche de *M. tuberculosis* est appelée MDR-TB *Multi-Drug-Resistant Tuberculosis* si elle est résistante au deux anti-tuberculeux de première intention les plus puissants, l'isoniazide et le rifampicine. Dans ce cas, certaines régions du génome de *M. tuberculosis* sont impliquées dans la résistance à plus d'un médicament. La découverte de nouvelles cibles moléculaires s'avère essentielle pour lutter contre ce développement de la résistance chez *M. tuberculosis*. Une souche de *M. tuberculosis* est appelée XDR-TB si elle est de surcroît résistante aux anti-tuberculeux de seconde intention tels que le fluoroquinolone et l'aminoglycoside.

Les causes de la résistance de *M. tuberculosis* aux antibiotiques sont multiples, mais il s'agit principalement de l'utilisation inappropriée ou incorrecte d'antibiotiques, et de l'interruption précoce des traitements. Dans ce cas, les souches résistantes se transmettent génétiquement de générations en générations. Toutefois, ces souches résistantes de *M. tuberculosis* peuvent aussi se transmettre directement à une personne saine, qui dans ce cas, se retrouve infectée avec une souche MDR-TB sans avoir pris de traitement inapproprié contre la tuberculose.

11. **Sous-unité** β : élément de l'ARN polymérase des bactéries qui est composé de la structure suivante $\alpha_2\beta\beta'\omega$
12. **ARN polymérase** : complexe enzymatique responsable de la synthèse de l'ARN à partir d'ADN.

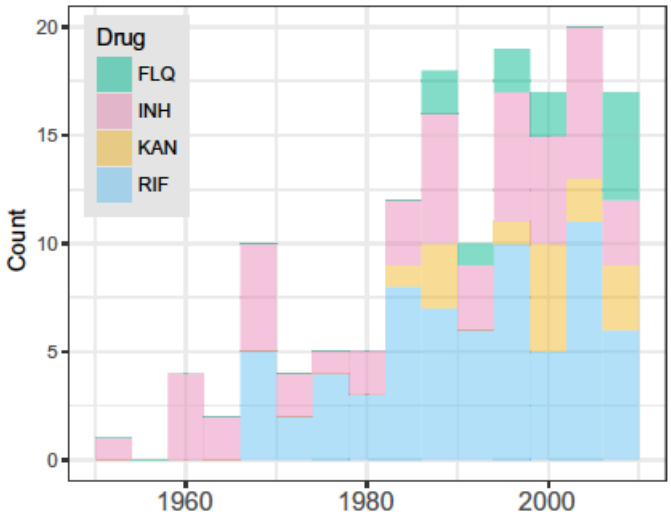


Figure 3 – Transmission de la résistance ces dernières années à l'échelle mondiale, d'après *Global expansion of Mycobacterium tuberculosis lineage 4 shaped by colonial migration and local adaptation*. FLQ=fluoroquinolone, INH=isoniazide, KAN=aminoglycoside, RIF=rifampicine

Brynildsrud O.B. et al. étudient également le gène lldD2 impliqué dans la réplication de *M. tuberculosis* au sein des macrophages [13] humains. Ils identifient au niveau des codons 3 et 253 la présence de nombreux promoteurs [14] et mutations non-synonymes qui ont évolué indépendamment.

Une recherche au sein d'une base de données recouvrant les lignées 1 à 6 a révélé que la mutation du codon 3 a émergé indépendamment dans les lignées 1, 2 et 4, alors que la mutation du codon 253 est apparue à plusieurs reprises dans la lignée 4 et est présente dans pratiquement tous les isolats de la lignée 2. Brynildsrud O.B. et al. constatent que les mutations de lldD2 ont commencé à apparaître bien avant l'utilisation des antibiotiques sur tous les continents. Ceci suppose une adaptation locale de *M. tuberculosis* à de profonds changements chez l'hôte humain, qui s'est opérée en parallèle sur les différents continents. Par ailleurs, les souches hébergeant des mutations du promoteur lldD2 présentent un avantage significatif en terme de transmissibilité.

Il ne fait aucun doute que des souches MDR-TB peuvent traverser les frontières, comme cela a déjà été observé avec la lignée 2 entre l'Europe de l'Est et l'Europe de l'Ouest. Toutefois, le jeune âge relatif des souches résistantes pourrait expliquer le manque de migrations observées de ces souches. Brynildsrud O.B. et al. démontrent que, d'un point de vue mondial, la migration humaine a joué un rôle négligeable dans l'élaboration des modèles de résistance aux antimicrobiens. En effet, la migration des souches résistantes s'est avérée marginale. Il s'agit plutôt d'un phénomène local. La restriction géographique de souches résistantes suggère même de lutter contre ce type de mutation de *M. tuberculosis* de façon nationale plutôt que de recourir à une politique globale de traitement antibiotique.

13. **Macrophage** : cellule appartenant aux globules blancs qui infiltre les tissus et est capable de phagocytose.
14. **Promoteur** : région de l'ADN située à proximité d'un gène et indispensable à la transcription de l'ADN.

3. Le locus CRISPR-Cas

3.1. Quelques caractéristiques du génome de *M. tuberculosis*

La souche H37Rv de *M. tuberculosis* est la souche de tuberculose la plus étudiée en laboratoire, depuis sa découverte en 1905. Elle sert aujourd'hui de référence pour le séquençage et l'annotation du génome de *M. tuberculosis*. Constitué d'environ 4 millions de paires de base et 3959 gènes, ce génome se caractérise par un taux élevé de guanine G et de cytosine C (65,6%), et un codon GTG qui sert de codon d'initiation dans 35% des gènes.

Parmi les marqueurs génétiques utilisés pour des études phylogéniques ou d'épidémiologie moléculaire, on retrouve les SNPs, les loci CRISPR, les MIRU [15], et les VNTR [16]. L'association des résultats obtenus par ces marqueurs génère un profil allélique [17] utile pour l'étude du complexe *M. tuberculosis*. La base de données mondiale de marqueurs moléculaires de la tuberculose SITVIT [18] présentée par Demay C. et al.[12] contient les génotypes de *M. tuberculosis* obtenus à partir des marqueurs moléculaires MIRU et VNTR.

3.2. Description du locus CRISPR-Cas

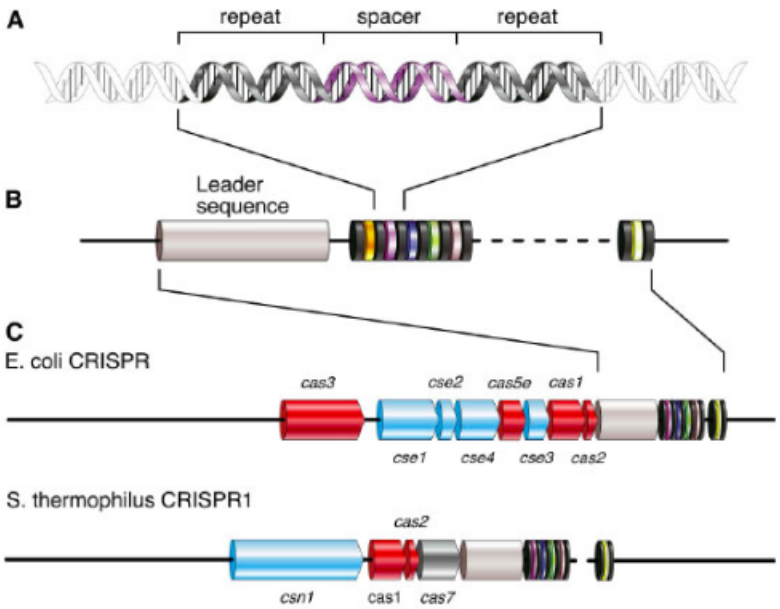


Figure 4 – Structure du locus CRISPR-Cas, d'après https://www.sinobiological.com/crispr-locus.html

15. **Marqueur MIRU Mycobacterial Interspaced Repetitive Units** : séquences nucléotidiques courtes répétitives en tandem entrecoupées de mycobactéries. La méthode MIRU actuellement utilisée sur *M. tuberculosis* est composée de 12 loci MIRU différents. Un mirutype est un modèle à 12 chiffres représentant le nombre de répétitions de chacun de ces 12 loci spécifiques.

16. **Marqueur VNTR Variable Number of Tandem DNA Repeats** : séquences nucléotidiques courtes en tandem à nombre variable. Cinq répétitions en tandem exactes (locus ETR) sont utilisées pour l'analyse VNTR du complexe *M. tuberculosis*.

17. **Allèle** : version variable d'un même gène.

18. **Base de données SITVIT** : base de données de l'Institut Pasteur de Guadeloupe consultable en ligne http://www.pasteur-guadeloupe.fr:8081/SITVIT_ONLINE/query, permettant n'analyser des data liées au MTBC. Elle comprend les spoligotypes de *M. tuberculosis*, ainsi que les marqueurs utilisés pour les détecter MIRU12, VNTR, SIT, MIT, VIT, les différentes branches de MTBC, les pays d'origine et l'année de découverte.

Le locus CRISPR *Clustered Regularly Interspaced Short Palindromic Repeats* est une famille de séquences répétées (DR pour *Direct Repeat*) dans l'ADN formant un palindrome, qui se trouve à l'état naturel chez 40% des bactéries (dont le *M. tuberculosis*) et la plupart des archées. CRISPR est héritable par transmission aux cellules filles et se conserve donc pour une même espèce. Chez *M. tuberculosis*, chaque série de répétitions contient 36 bp [19] ; les répétitions étant régulièrement espacées par des espaceurs de 34 à 41 bp. A l'heure actuelle, 104 espaceurs ont été identifiés dans toutes les souches de *M. tuberculosis*. Les loci CRISPR sont généralement adjacents aux gènes Cas, dont ils sont séparés par une séquence de 300 à 500 bp, appelée leader qui contrôle à la fois l'acquisition de l'ADN viral par les espaceurs et la fabrication des protéines. Les gènes Cas produisent des protéines aux fonctionnalités multiples et notamment les enzymes [20] capables de couper l'ADN en vue de sa réparation.

Ces séquences CRISPR incorporent dans les espaceurs des fragments d'ADN de bactériophages qui ont déjà infecté la bactérie, et sont stockés pour détecter et détruire l'ADN de bactériophages similaires en cas de nouvelle infection. Par conséquent, CRISPR-Cas est un système immunitaire naturel utilisé par les bactéries pour se protéger des infections virales.

3.3. Fonctionnement du système CRISPR-Cas

Les systèmes CRISPR-Cas sont de trois types et utilisent les différents gènes Cas pour intégrer des fragments de gènes étrangers dans les espaceurs de CRISPR. Par exemple, dans le cas d'une bactérie qui détecte la présence d'ADN ou d'ARN d'un virus, elle produit une enzyme nucléase appelée Cas9 capable de couper l'ADN viral, puis une séquence d'ARN CRISPR notée crARN correspondant à celle de l'ADN du virus et servant de guide ARN, et finalement une séquence d'ARN traceur notée trARN. Lorsque trARN trouve sa cible parmi le génome du virus, Cas9 sectionne l'ADN viral puis en incorpore un fragment dans un espaceur du génome de la bactérie, conservant ainsi en mémoire une trace de ce virus en vue d'une éventuelle infection future. Les espaceurs servent donc de banque de mémoire en conservant l'ADN des virus qui ont attaqué la bactérie. Cette fonctionnalité va être exploitée de différentes manières par les biologistes.

La technologie CRISPR-Cas9, s'inspirant du système du même nom, a d'abord été utilisée pour typer les souches bactériennes, suivant une technique appelée spoligotypage. CRISPR-Cas9 est actuellement principalement employé comme ciseau moléculaire afin d'éditer le génome et d'introduire localement des modifications génétiques.

4. Le spoligotypage

La région DR du locus CRISPR-Cas présente un niveau de polymorphisme suffisant pour pouvoir classer phylogéographiquement les souches de *M. tuberculosis*. Le polymorphisme entre les différentes souches résulte des variations et de l'identité des espaceurs. C'est ce polymorphisme qui est exploité en 1997 par Kamerbeek et al. et expliqué dans [16] comme technique de génotypage spécifique de *M. tuberculosis*. Le *Spacer Oligonucleotide Typing*, repose sur la détection de séquences répétitives trouvées entre les gènes d'un agent infectieux au sein d'un locus CRISPR-Cas. Pour ce faire, la région DR d'un isolat à tester subit un traitement par amplification PCR [21] ou celui d'une puce à

19. **bp** : une paire de base.
20. **Enzyme de restriction** : protéine capable de couper un fragment d'ADN au niveau d'une séquence de nucléotides caractéristique appelée site de restriction. Chaque enzyme de restriction reconnaît ainsi un site spécifique.
21. **PCR Polymerase Chain Reaction** : méthode de réaction en chaîne utilisant un polymère pour dupliquer en grand nombre une séquence d'ADN spécifique. La méthode PCR repose sur le cycle thermique, qui expose les séquences à des cycles répétés de chauffage et de refroidissement pour permettre différentes réactions dépendantes de

ADN [22], pour dévoiler un motif de taches correspondant aux espaceurs.

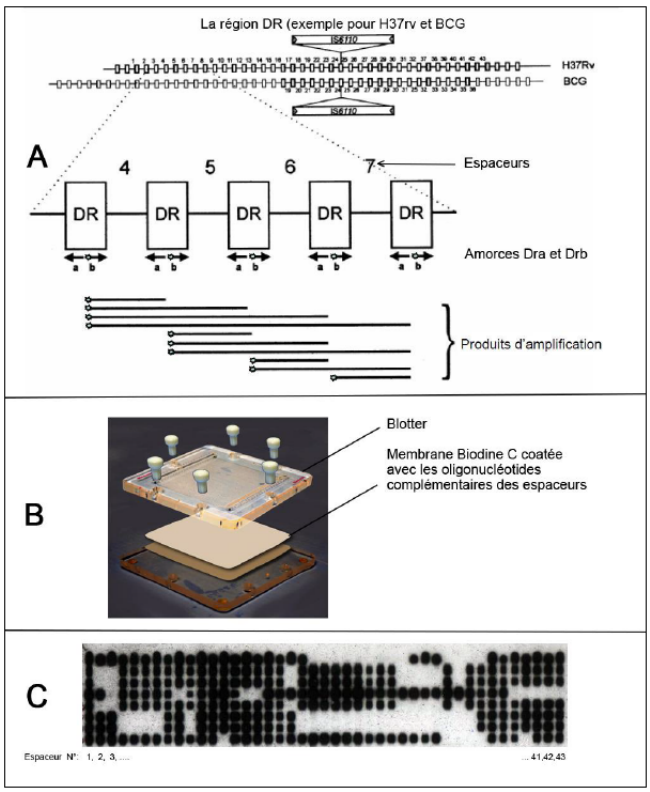


Figure 5 – Les différentes étapes du spoligotypage d'après *Etudes descriptive, épidémiologique, moléculaire et spatiale des souches Mycobacterium tuberculosis circulant à Antananarivo, Madagascar*

La comparaison de ces motifs permet la différentiation des souches. Quarante-trois espaceurs les plus polymorphes ont été utilisés pour le typage des mycobactéries suivant Kamerbeek et al. La classification classique de MTBC utilise donc un groupe de 43 bits représentant la présence ou l'absence d'espaceurs dans le locus CRISPR, qu'on appelle spoligotype. Des études pour augmenter le niveau de discrimination du spoligotypage ont été faites en 2010 utilisant 68 espaceurs. A l'heure actuelle l'équipe AND de l'université de Franche Comté utilise 98 espaceurs pour ce génotypage.

La technique ne nécessite pas une importante quantité d'ADN car elle est basée sur une amplification de la région DR par PCR. Les spoligotypes ainsi obtenus peuvent être partagées entre laboratoires et corroborent les résultats recueillis à partir d'autres marqueurs génétiques. Ces données numériques permettent de bien différencier les souches de *M. tuberculosis* et sont de moindre coût comparativement à d'autres méthodes. Cependant, le spoligotypage éprouve des difficultés à bien différencier les souches au sein de grandes familles de *M. tuberculosis* telles que la lignée 2 par exemple.

Jusqu'à présent, le spoligotypage a permis de fournir une image globale de la diversité des souches de *M. tuberculosis*.

Une nouvelle technologie permettant de combiner le spoligotypage avec des tests moléculaires de sensibilité aux anti-tuberculeux, appelée spoligoriftypage, a été développée pour aboutir à la version

la température comme la fusion de l'ADN et la réplication de l'ADN par les enzymes. La méthode PCR utilise deux agents principaux : les polymères d'ADN i.e. des macromolécules répétant un même motif structural d'ADN et les amorces de séquençage.

22. **Puce à ADN** : ensemble de molécules d'ADN fixées sur une petite surface solide permettant de mesurer le niveau d'expression d'un grand nombre de gènes simultanément, ou de déterminer le génotype de plusieurs régions d'un génome.

Numéro	Numéro espaceur en spoligotypage	Numéro espaceur dans le génome	séquence des oligonucléotides (5'→3')	Numéro	Numéro espaceur en spoligotypage	Numéro espaceur dans le génome	séquence des oligonucléotides (5'→3')
1	1	2	CGCTCCCCTAGTCGT	53	-	17	TCTTGAGCAACGCCATC
2	2	3	TGGGCGACAGCTTTTGA	54	-	45	AAGTTGGCGCTGGGG
3	3	4	CTTCCAGTGATCGCCTT	55	-	48	AACCGTCCCACCTG
4	4	12	TCATACGCCGACCAATC	56	-	49	AACACTTTTTTTGAGCGTGG
5	5	13	TTCTGACCACTTGTGCG	57	-	50	CGGAAACGCAGCACC
6	6	14	TCATTTCCGGCTT	58	-	54	CGATCATGAGAGTTGCG
7	7	15	TGAGGAGAGCGAGTACT	59	-	55	TTTTCGCTGTTGTGGTTCT
8	8	18	TGAAACCGCCCCCAG	60	-	56e	AGCACCTCCCTTGACAA
9	9	19	ACTCGGAATCCCATGTG	61	-	57	TGCTGACTTCGCCTGTA
10	10	20	CTCTAGTTGACTTCCGG	62	-	58	CGAGCAGCGGCATAC
11	11	21	CAGGTGAGCAACGGC	63	-	59	GCATCCACTCGTCGC
12	12	22	ATGGGATATCTGCTGCC	64	-	60	TGGTAATTGCGTCACGG
13	13	23	ATTGCCATTCCCTCTCC	65	-	61	ACCATCCGACGCAGG
14	14	24	TTTCGGTGTGATGCGGA	66	-	66	CCACGCTACTGCTCC
15	15	25	TGAATAACGCGCAGTGAAT	67	-	67	CACCGCCGATGACAG
16	16	26	TCGCACGAGTTCCCG	68	-	68	GTGTTTCGGCCGTGC
17	17	27	CCGGCAACAATCGCG	69	-	69	GTTGCATTCGTCGACTG
18	18	28	TGCAGATGGTCCGGG	70	-	70	GGCGGCGCCGAGAA
19	19	29	TTGCGCTAACTGGCTTG	71	-	71	TTCCATGACTTGACGCC
20	20	30	ATTTCCTTGACCTCGCC	72	-	72	CGATGCGGCCACTAG
21	21	31	CGATGTCGATGTCCCAA	73	-	73	GCTGACCCCATGGATG
22	22	32	ACGGCACGATTGAGACA	74	-	74	CAACAAGGTCTACGCGT
23	23	33	GTCCAGCTCGTCCGT	75	-	75	GATCAGGCGAAGGCG
24	24	34	GCCTGCTGGGTGAGA	76	-	76	ATTGCAGCGACGGGC
25	25	35	GGAGCCGATCAGCGA	77	-	77	CAACGACGCTGTATTGG
26	26	36	CTTCAGCACCACCATCA	78	-	78	AGCAGCATGGACGGTTT
27	27	37	TTCGTGATCTCTTCCCG	79	-	79	GCGGATGTGGTGGTC
28	28	38	GATCACAACACCAACTAATG	80	-	80	GTACATAGCGAGCTG
29	29	39	GAAATACAGGCTCCACG	81	-	81	GCCGCGGGTTTCGTT
30	30	40	TCTTGACGATGCGGTTG	82	-	82	GGGGCGTGTGTTCGT
31	31	41	TTCGCGTCAGACAGGTT	83	-	83	CTGGTGTGCTTATGCCT
32	32	42	ACTCCCGACCAAATAGG	84	-	84	CAAATGTTTGGACTGTGATC
33	33	43	TCGACACCGACATGAC	85	-	85	TTGTCGCGCGCCTTTTT
34	34	44	GAAGTCACCTCGCCC	86	-	86	GTTTCAGTTTTCTTGTCCC
35	35	46	AGTCCGTACGCTCGAAA	87	-	87	CTGGTTGTTGCCCGG
36	36	47	CGAAATCCAGCACCACA	88	-	88	TGTTCGGTGTTCTCCTG
37	37	51	TTTGAGCGCGAACTCGT	89	-	89	TCATGACGAGCCCGCA
38	38	52	TGGATGGCGGATGCG	90	-	90	ACACGGCCTGATCGGT
39	39	53	AAATCGGCGTGGGTAAC	91	-	91	CGGATTGTCTGGCCC
40	40	62	TCATACAGGTCCAGTGC	92	-	92	TAAGCACGCGTCTGTCA
41	41	63	GCTTTCCGGCTTCTATC	93	-	93	GACCACCGAATCACCAT
42	42	64	GACATGGACGAGCGC	94	-	94	TCTGGTAGTGGGCTTCT
43	43	65	CAGAATCGCACCGGG	95	-	11	ACATGCCGTGGCTCA
44	-	1	CAACCCGGAATTCTTGC	96	-	16	CACGACGTTAGGGCA
45	-	5	CAGGCGTGGCTAGG	97	-	5	CGGCAGGCGTGGCTA
46	-	6	GTCGCCGTAAGTGCC	98	-	6	CCGTCGCCGTAAGTG
47	-	7	GTTGACCACGAATTTTCAG	99	-	17	GAGCAACGCCATCAT
48	-	8	GCTGGCGCGCATCAT	100	-	11	TGAGCCACGGCATGT*
49	-	9	CCATATCGGGGACGG	101	-	16	ATGCCCTAACGTCGTG*
50	-	10	GCGTCGTGCCATCAG	102	-	5	TAGCCACGCCTGCCG*
51	-	11	CCGTGCACATGCCGT	103	-	6	CACTTACGGCGACGG*
52	-	16	ACGTTAGGGCATGCAG	104	-	17	GATGATGGCGTTGCT*

Figure 6 – Espaceurs connus chez MTBC, d'après *Etudes descriptive, épidémiologique, moléculaire et spatiale des souches Mycobacterium tuberculosis circulant à Antananarivo, Madagascar*

TB-SPRINT qui a été décrite en 2013 par Gomgnimbou et al. dans leur article [18]. Elle consiste au typage tuberculose-spoligo-rifampicine-isoniazide fonctionnant sur des systèmes à base de microbilles, à partir notamment de 43 espaceurs, 11 SNPs présents sur rpoB aux positions 516, 526 et 531. Cette nouvelle génération de spoligotypage fournit donc, en plus des données classiques de génotypage, une prédiction basée sur la mutation des profils de résistance aux médicaments.

4.1. Vers une normalisation des spoligotypes

Au début du spoligotypage, il n'existait pas de norme pour décrire les motifs formés par les espaceurs ou simplement les numéroter. Chaque laboratoire utilisait son propre système de numérotation accompagné d'un schéma descriptif du motif. Ce manque de normalisation entravait les possibilités

de comparaison des résultats obtenus et le développement d'une vision mondiale de l'évolution de *M. tuberculosis*. Une méthode standardisée de description des spoligotypes a été proposée en 2001 par Dale JW dans son article [10].

Tout d'abord, une base de données centralisée regroupant tous les motifs connus et de leurs numérotations associées existe au RIVM Rijksinstituut voor Volksgezondheid en Milieuhygiene, Bilthoven, Netherlands. Elle peut être consultée au `http://www.caontb.rivm.nl`. A partir de 2001, les nouveaux motifs doivent prendre un unique format de numérotation pour être répertoriés dans cette base de données. Toutefois, cela nécessite l'interrogation systématique de la base de données et la comparaison avec les éléments déjà existants pour chaque nouveau spoligotype. Pour éviter cette perte de temps, de nombreux laboratoires utilisent des systèmes rationnels avec des codes descriptifs assignés à chaque isolat.

Dale JW et al.[10] proposent d'utiliser exclusivement un système rationnel octal ou hexadécimal, sachant qu'il est aisé de passer de l'un à l'autre et qu'il est également facile de retrouver l'état initial binaire. Ainsi, les motifs de spoligotype comprenant 43 bits seraient réduits dans le système octal en 14 groupes de 3 bits auquel s'ajouterait un unique bit, ce qui donnerait finalement un ensemble de 15 chiffres en écriture octale. En ce qui concerne le système hexadécimal, les motifs de 43 bits seraient réduits en 6 groupes de 8 bits avec un dernier groupe ne comprenant que 3 bits, soit 6 groupes de 2 chiffres hexadécimaux. Notons qu'un bit symbolise dans ce cas la présence ou l'absence d'une espaceur dans le locus étudié.

Figure 7 – Exemple de système rationnel octal et hexadécimal, d'après `https://www.mbovis.org/spoligotype-nomenclature.php`

Le site Mbovis.org, `https://www.mbovis.org/database.php`, bien que dédié aux souches animales de MTBC, fournit une application pratique permettant de transformer rapidement les spoligotypes binaires en système octal ou hexadécimal.

4.2. Quel outil informatique pour le spoligotypage ?

Les technologies PCR de génotypage utilisent toujours différents marqueurs tels que les SNPs pour obtenir en laboratoire des résultats fiables. Des logiciels informatiques de prédiction du génotype ont également fait leur apparition pour optimiser les coûts et le gain de temps. Ils offrent un outil de comparaison des résultats obtenus expérimentalement et in silico.

SpolPred est un logiciel de prédiction rapide et précis des spoligotypes de *M. tuberculosis* à partir de séquences génomiques courtes appelées reads [23]. Cet outil développé par Coll F. et al. fonctionne efficacement avec des reads provenant de plateformes telles que Illumina GAII ou HiSeq. SpolPred utilise des fichiers de séquences de reads simples ou par paires au format FASTQ, afin de produire une

23. **Reads** : mélange de courtes séquences oligonocléotidiques de 20 à 200 bp générées par des séquenceurs

prédiction de spoligotype au format octal, qui est ensuite comparée au spoligotype correspondant dans la base SITVIT.

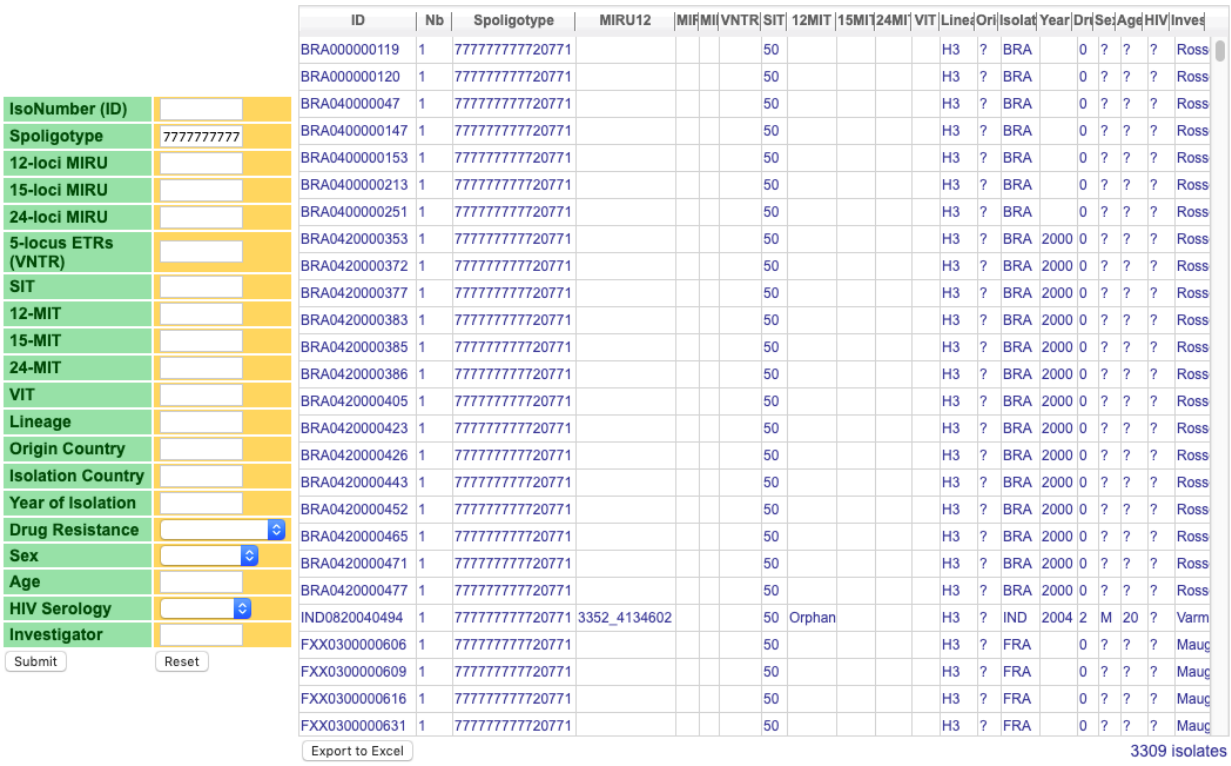


Figure 8 – Exemple de recherche effectuée sur SITVIT2 à partir du spoligotype 777777777720771 dont le résultat est exploitable au format Excel

Dans leur étude [2], Coll F. et al. montrent en 2012 l'utilité de SpolPred en comparant les spoligotypes obtenus par le logiciel avec les résultats de laboratoire. Ils dévoilent ainsi les limites de la méthode expérimentale qui a répertorié cinq faux spoligotypes, alors que SpolPred a su éviter ces erreurs de classification du génotype. Par ailleurs, il apparaît que SpolPred offre plus de rapidité avec des résultats pratiquement identiques à ceux obtenus avec la méthode bio-informatique par assemblage. Cette dernière, développée en 2008 à l'aide du logiciel Velvet, consiste à fusionner des fragments d'ADN issus d'une plus longue séquence afin d'en reconstruire la séquence originale.

Toutefois, d'après l'étude de Xia et al.[9], la précision de SpolPred est fortement réduite lorsque les reads n'ont pas une taille uniforme, comme par exemple lorsqu'ils proviennent de séquençages Ion Torrent ou de la plateforme de diagnostique clinique Illumina MiSeq. Ainsi, lorsque les reads ne sont pas uniformes, la précision des résultats dépend fortement de leurs tailles et donc du choix initial fait par l'opérateur. Par ailleurs, SpolPred demande à l'utilisateur de spécifier la direction de lecture des reads, et le logiciel n'utilise donc qu'une partie des informations fournies par les reads.

Une problématique de SpolPred en 2020 est que le logiciel n'est plus disponible au téléchargement en ligne. En effet, une visite sur le site officiel `http://www.pathogenseq.org/spolpred`, fourni comme référence dans le document [2] de Coll F. et al., montre que le nom du domaine est à vendre. Preston M., qui a fait partie de l'équipe de recherche de Coll F. pour le développement de SpolPred, a bien créé un site proposant le téléchargement du logiciel `https://www.mybiosoftware.com/spolpred-predict-the-spoligotype-from-raw-sequence-reads.html`, mais le lien est inactif en janvier 2020.

Une alternative à SpolPred est SpoTyping présenté en 2016 dans l'article [9] de Xia et al. comme

étant 20 à 40 fois plus rapide que SpolPred pour prédire avec précision des spoligotypes de *M. tuberculosis* à partir de reads de taille uniforme ou variable. Par ailleurs, SpoTyping lit chaque read dans les deux directions en exploitant complètement les informations fournies. SpoTyping réduit la durée des recherches en intégrant l'algorithme BLAST [24] dans ses calculs. Il compare les isolats testés avec ceux ayant le même spoligotype dans la base de données mondiale SITVIT, qui regroupe les données épidémiologiques [25] associées à des isolats de même spoligotype.

L'intérêt d'un outil tel que SpolPred ou Spotyping est qu'il est capable de combiner le spoligotypage avec d'autres méthodes telles que MIRU (unités répétitives entrecoupées de mycobactéries) et VNTR nombre variable de répétitions d'ADN en tandem) en utilisant la base de données SITVIT.

SpoTyping utilise des fichiers de séquences de reads simples ou par paires au format FASTQ et des fichiers de séquences complètes de génomes ou de contigs [26] assemblés au format FASTA. Les séquences de reads sont regroupées en une unique séquence continue au format FASTA pour être ensuite soumise à l'algorithme BLAST qui détecte les régions similaires. Finalement la base de données SITVIT permet d'identifier les isolats ayant le même spoligotype. SpoTyping est limité à une lecture de 250 Mbp au sein des séquences de reads testées, lors de l'utilisation du swift mode qui accélère le temps de traitement.

SpoTyping propose un rapport statistique permettant de résumer le rapprochement avec les spoligotypes trouvés dans la base de données SITVIT, ainsi qu'une estimation du nombre de correspondances positives pour chaque espaceur.

D'après le repository https://github.com/xiaeryu/SpoTyping-v2.0/blob/master/SpoTyping-v2.0-commandLine/SpoTyping-README.pdf, les spécifications techniques de SpoTyping sont les suivantes :
- SpoTyping peut s'exécuter sur les principaux systèmes d'exploitation, contrairement à SpolPred qui utilise exclusivement Linux. Il se présente à la fois sous forme de script et sous forme d'application avec une interface graphique.
- SpoTyping est un logiciel open-source qui peut se télécharger gratuitement à l'adresse https://github.com/xiaeryu/SpoTyping-v2.0. SpoTyping nécessite l'utilisation de Python2.7 et BLAST.
- il est recommandé d'utiliser le swift mode paramétré par défaut si le débit de séquençage [27] est inférieur à 135 Mbp. Pour les débits de séquençage inférieurs à 135 Mbp ou supérieurs à 1260 Mbp, les seuils doivent être réglés entre 0.018 et 0.1486 fois la profondeur de lecture estimée pour les hits sans erreur, et entre 0.018 et 0.1488 fois la profondeur de lecture estimée pour les hits tolérant une erreur. Notons que la profondeur de lecture est définie par le débit de séquençage divisé par 4 500 0000 qui correspond à l'estimation de la longueur d'un génome de *M. tuberculosis*.

Le fichier obtenu propose une prédiction de spoligotype au format de code binaire et octal. Le fichier log obtenu contient le nombre de correspondances positives des résultats de BLAST pour chaque séquence d'espaceurs. Le fichier xls Excel obtenu fournit le résultat de la recherche de spoligotype auprès de la base de données SITVIT2 [28].

24. **BLAST Basic Local Alignment Search Tool** : logiciel basé sur l'algorithme du même nom qui détecte des régions similaires entre plusieurs séquences biologiques. Le programme compare les séquences de nucléotides aux séquences contenues dans la base de données BLAST pour fournir des résultats statistiquement significatifs.
25. **Epidémiologie** : discipline scientifique qui étudie les problèmes de santé dans les populations humaines, leur fréquence, leur géographie ainsi que les facteurs influents.
26. **Contig** : séquence génomique continue et ordonnée, générée par l'assemblage des clones d'une bibliothèque génomique qui se chevauchent.
27. **Séquençage du génome** : consiste, par des méthodes chimiques ou de biologie moléculaire, à déterminer l'ordre des nucléotides de l'ADN.
28. **Base de données SITVIT2** : mise à jour de la base de données SITVIT, consultable en ligne http://www.pasteur-guadeloupe.fr:8081/SITVIT2/index.jsp

Figure 9 – Exemple de fonctionnement de SpoTyping, d'après *SpoTyping : fast and accurate in silico Mycobacterium spoligotyping from sequence reads*

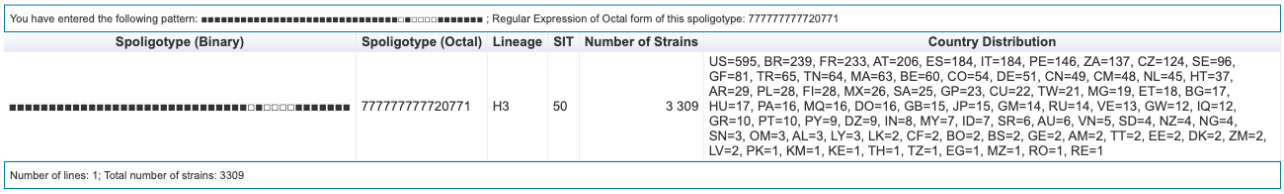
L'étude de Iwai H. et al.[11] envisage une autre possibilité de travail et montre l'intérêt d'une analyse de *M. tuberculosis* à l'aide de serveurs, appelée CASTB, et notamment le spoligotypage. Le Webserver fournit une vue complète des données, mais les performances de chaque outil utilisé ne sont pas décrites dans l'article. Il est probable que le spoligotypage prenne plus de temps en passant par un serveur suite au problème de disponibilité des données et aux lenteurs de téléchargement de ces données. Il semblerait que SpoTyping, de par sa configuration locale, puisse fournir un résultat en une minute.

4.3. Comparaison de spoligotypes

Une fois les spoligotypes de différentes lignées obtenus, il est nécessaire de les comparer pour chercher à faire ressortir les points communs ou certains traits pouvant être liés à une mutation particulière. Il existe à l'heure actuelle un premier outil en ligne de comparaison du nom de SpolSimilaritySearch, accessible à l'adresse `http://www.pasteur-guadeloupe.fr:8081/SpolSimilaritySearch/index.jsp`, et présenté par Couvin D. et al.[19]. SpolSimilaritySearch incorpore un algorithme de recherche de similitudes entre spoligotypes dans la base de données SITVIT2. Cet outil permet d'analyser les modèles de propagation et d'évolution de *M. tuberculosis* en comparant des modèles de spoligotypes similaires, de distinguer les modèles répandus, confinés ou spécifiques, d'identifier les modèles ayant de grands blocs supprimés ou encore de fournir les modèles de distribution par pays pour chaque spoligotype interrogé.

Par exemple, si on sélectionne le spoligotype 777777777720771 appartenant à la lignée H3, et qu'on interroge la base SpolSimilaritySearch, on obtient les rapprochements suivants :

Cet outil pourrait donc s'avérer utile pour commencer à chercher des liens entre les sept lignées de *M. tuberculosis* et les spoligotypes de différentes souches. Un tableau comparatif de différents



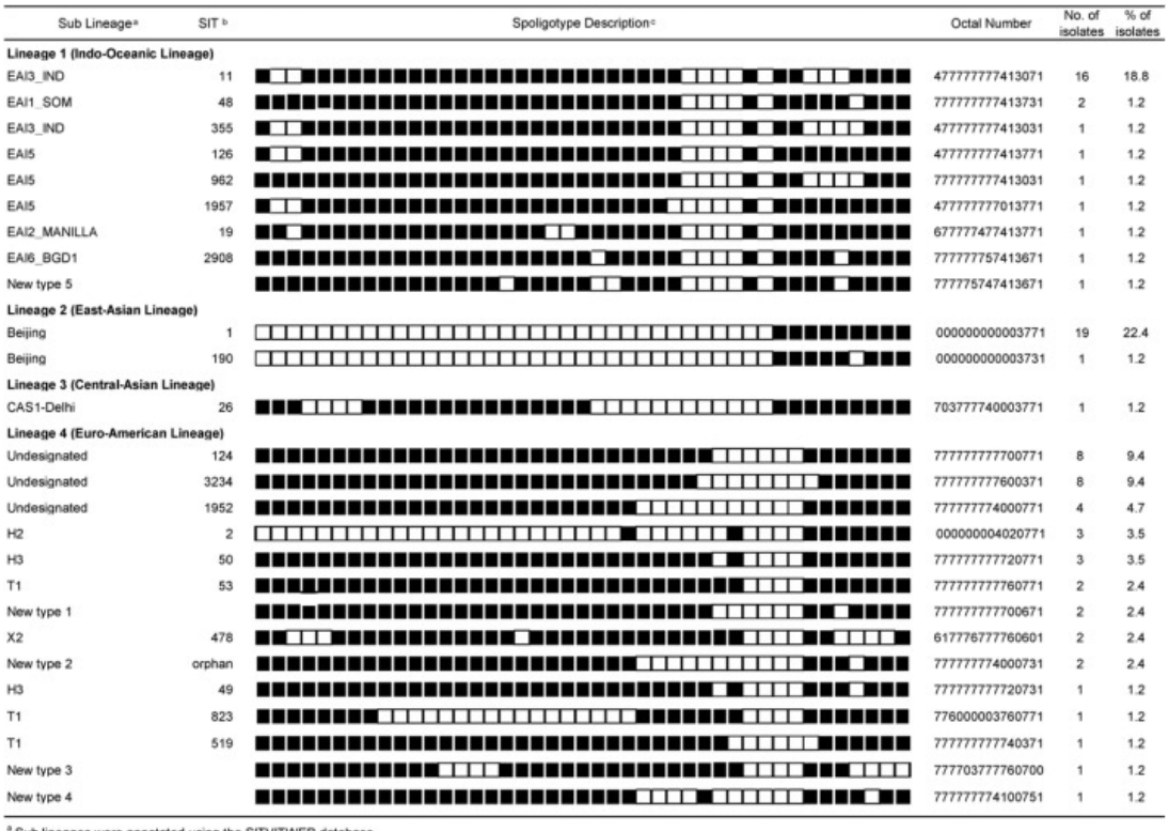
You have entered the following pattern: ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■□■□□□□■■■■■■■ ; Regular Expression of Octal form of this spoligotype: 777777777720771

Spoligotype (Binary)	Spoligotype (Octal)	Lineage	SIT	Number of Strains	Country Distribution
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■□■□□□□■■■■■■■	777777777720771	H3	50	3 309	US=595, BR=239, FR=233, AT=206, ES=184, IT=184, PE=146, ZA=137, CZ=124, SE=96, GF=81, TR=65, TN=64, MA=63, BE=60, CO=54, DE=51, CN=49, CM=48, NL=45, HT=37, AR=29, PL=28, FI=28, MX=26, SA=25, GP=23, CU=22, TW=21, MG=19, ET=18, BG=17, HU=17, PA=16, MQ=16, DO=16, GB=15, JP=15, GM=14, RU=14, VE=13, GW=12, IQ=12, GR=10, PT=10, PY=9, DZ=9, IN=8, MY=7, ID=7, SR=6, AU=6, VN=5, SD=4, NZ=4, NG=4, SN=3, OM=3, AL=3, LY=3, LK=2, CF=2, BO=2, BS=2, GE=2, AM=2, TT=2, EE=2, DK=2, ZM=2, LV=2, PK=1, KM=1, KE=1, TH=1, TZ=1, EG=1, MZ=1, RO=1, RE=1

Number of lines: 1; Total number of strains: 3309

Figure 10 – Recherche effectuée sur le site de SpolSimilaritySearch

spoligotypes en fonction des lignées de *M. tuberculosis* pourrait ressembler à celui de la figure 11.



Sub Lineage[a]	SIT [b]	Spoligotype Description[c]	Octal Number	No. of isolates	% of isolates
Lineage 1 (Indo-Oceanic Lineage)					
EAI3_IND	11		477777777413071	16	18.8
EAI1_SOM	48		777777777413731	2	1.2
EAI3_IND	355		477777777413031	1	1.2
EAI5	126		477777777413771	1	1.2
EAI5	962		777777777413031	1	1.2
EAI5	1957		477777777013771	1	1.2
EAI2_MANILLA	19		677777477413771	1	1.2
EAI6_BGD1	2908		777777757413671	1	1.2
New type 5			777775747413671	1	1.2
Lineage 2 (East-Asian Lineage)					
Beijing	1		000000000003771	19	22.4
Beijing	190		000000000003731	1	1.2
Lineage 3 (Central-Asian Lineage)					
CAS1-Delhi	26		703777740003771	1	1.2
Lineage 4 (Euro-American Lineage)					
Undesignated	124		777777777700771	8	9.4
Undesignated	3234		777777777600371	8	9.4
Undesignated	1952		777777774000771	4	4.7
H2	2		000000004020771	3	3.5
H3	50		777777777720771	3	3.5
T1	53		777777777760771	2	2.4
New type 1			777777777700671	2	2.4
X2	478		617776777760601	2	2.4
New type 2	orphan		777777774000731	2	2.4
H3	49		777777777720731	1	1.2
T1	823		776000003760771	1	1.2
T1	519		777777777740371	1	1.2
New type 3			777703777760700	1	1.2
New type 4			777777774100751	1	1.2

Figure 11 – Exemple comparatif de 26 spoligotypes correspondant à 4 lignées différentes de *M. tuberculosis*, d'après *Insight into genetic diversity of Mycobacterium tuberculosis in Kandy, Sri Lanka reveals predominance of the Euro-American lineage*

3

La notion de package

1. Vocabulaire

Définissons tout d'abord le vocabulaire que nous allons utiliser dans ce rapport de stage :

- module : un fichier contenant du code Python,
- package : un répertoire contenant des modules Python,
- distribution : une archive de modules ou packages (au format tar, whl, ...)

2. Composition d'un package standard

Un package standard comporte obligatoirement un fichier *__init__.py* qui va définir la version du projet et le nom du module de lancement du programme (souvent *__main__.py*). Il est toutefois acceptable et parfois même recommandé de conserver un fichier *__init__.py* vide. Par ailleurs, chaque sous dossier vide ou contenant des modules Python devra également contenir un fichier *__init__.py*

Les fichiers *setup.py*, *requirements.txt*, *LICENSE*, *README.md* et *MANIFEST.in* sont également nécessaires.

- *setup.py* est le script de construction et configuration destiné au *setuptools*. Il définit notamment le nom et la version du package, ainsi que les fichiers qu'il contient. Il sert également d'interface en ligne de commande relative aux différentes fonctionnalités du package. *setup.cfg* est un fichier d'initialisation qui contient les options par défaut des commandes du *setup.py*.
- *requirements.txt* permet l'installation des dépendances à l'aide d'un unique fichier contenant un module à installer par ligne. Il nécessite l'instruction

  ```
  pip install -r requirements.txt
  ```

  pour commencer ces installations.
- *LICENSE* définit les termes légaux de la distribution. De nombreux pays n'autorisent pas l'utilisation ou la distribution d'un package qui ne dispose pas de licence.
- *README.md* décrit l'objectif du package, son installation, la nature de ses dépendances et les principales fonctionnalités.

- *MANIFEST.in* permet d'inclure dans le package certains fichiers qui ne sont pas automatiquement intégrés.

Il est important de regrouper l'ensemble des modules nécessaires au fonctionnement du package dans un répertoire portant le même nom que ce package, accompagné d'un fichier *__init__.py*, sinon l'archive et la distribution du package se construiront à vide. L'installation à partir de PyPI fonctionnera mais l'exécution du package restera inactive.

Avec la PEP-518 [1], le PyPA [2] a proposé un nouveau standard au format *pyproject.toml*, qui remplace les fichiers *setup.py, requirements.txt, setup.cfg*, et *MANIFEST.in*. C'est ce nouveau standard qui est utilisé lors de la création d'un package avec Poetry.

Les paquets construits pour des systèmes Unix (Linux et MacOS) nécessitent l'incorporation de fichiers build.sh et meta.yaml. Les paquets construits pour les systèmes Windows nécessitent l'incorporation des fichiers bld.bat et meta.yaml. A VERIFIER DANS LE CAS DE POETRY

3. Quelle licence choisir ?

Trois licences retiennent notre attention. En voici les principales caractéristiques :

- la licence MIT, courte et permissive, préserve exclusivement le copyright et les avis de licence. Toute modification ultérieure peut être distribuée suivant une licence différente et notamment utilisée à des fins personnelles ou commerciales, sans obligation de publication des codes source,
- la licence Apache (2.0) est également permissive et sensiblement similaire dans ses conditions à la licence MIT. Toutefois, elle requiert de préciser les modifications effectuées lors de nouvelles distributions,
- la licence GNU (GPL v3.0) préserve également le copyright et les avis de licence. Elle peut être utilisée à des fins personnelles et commerciales. Elle impose en outre, en cas de modification, la publication complète des codes et l'utilisation de la licence GNU pour les nouvelles distributions.

Dans le cadre de ce projet, aucune spécification restrictive n'étant requise, nous avons choisi la licence MIT qui est simple et peu contraignante.

1. **PEP-518 :** specifying minimum build system requirements for Python projects
2. **PyPA :** Python Packaging Authority

4

Le choix des outils

Note préalable : toutes les explications données dans ce chapitre concernent les plateformes Linux et sont transposables en une moindre mesure sur MacOS.

1. L'environnement de développement

Avant de créer le package, il faut commencer par construire localement le programme. Pour ce faire, nous définissons un environnement de développement, qui nous servira également à effectuer des tests.

L'outil de création d'environnement virtuel *venv* nous permet de créer un environnement de développement afin de tester les codes du projet. *venv* commence par constituer un dossier contenant tous les exécutables nécessaires à l'utilisation des modules d'un projet Python.

Dans le cadre de notre étude, nous créons un répertoire **biologie**, nous nous plaçons dedans puis nous créons, à l'aide de *venv*, un environnement de développement **env_bio**.

```
$ cd biologie
$ python3 -m venv env_bio
```

Dans cet environnement vient d'être créé le répertoire **bin** comprenant notamment le fichier *activate*, qui permet d'activer les fonctionnalités de notre environnement. Cette activation ne peut se faire qu'à partir du répertoire parent **biologie**. Il faut ensuite se placer à l'intérieur de **env_bio** pour définir les modules utiles à l'environnement.

```
$ source env_bio/bin/activate
(env_bio)$ cd env_bio
```

La plupart des systèmes d'exploitation incorporent Python2.7 par défaut. Il convient donc de définir une version 3 de Python dans cet environnement de développement, et plus précisément une version 3.6 ou supérieure. Pour cela, le module *pyenv* nous permet de définir une version de Python comme version locale de travail dans l'environnement.

On peut ainsi choisir parmi les versions disponibles, grâce à l'instruction

```
$ pyenv install --list
```

La liste étant particulièrement longue, on peut éventuellement être plus spécifique en demandant les versions allant de 3.6 à 3.8 avec l'instruction

```
$ pyenv install --list | grep " 3[678]"
```

L'installation de la version 3.6.2 par exemple se fait grâce à l'instruction

```
$ pyenv install -v 3.6.2
```

La version 3.6.2 de Python se trouve maintenant installée dans **~/.pyenv/versions/**

Pour connaître toutes les versions de Python installées par *pyenv*, il suffit d'exécuter l'instruction suivante qui fournit une liste de résultats

```
$ pyenv versions
* system
2.7.10
3.6.2
3.8.0
```

où le caractère "*" indique la version active dans le répertoire courant.

On peut alors choisir la version 3.6.2 pour notre environnement de travail. Il suffit de se placer dans cet environnement et d'exécuter l'instruction

```
$ pyenv local 3.6.2
```

En fait, *pyenv* s'insère dans la variable d'environnement PATH et devient l'exécutable appelé par le système d'exploitation. On peut voir que l'interpréteur Python utilise bien cet environnement *pyenv* pour appeler Python grâce à l'instruction

```
$ which python
```

qui renvoie **~/.pyenv/shims/python** au lieu de **/usr/bin/python**.

A noter que dans un cadre plus général, l'instruction

```
$ python --version
```

permet de connaître quelle version de Python est utilisée dans le répertoire courant, qu'on se trouve dans un environnement de développement ou non.

Finalement, nous installons dans notre environnement de développement les dépendances nécessaires à l'aide de l'outil *pip* :

- ```
  $ pip install xmltodict
  ```

  le module *xmltodict* permet de lire du code XML comme si il s'agissait de code JSON. Il permet donc une lecture plus rapide des fichiers.

- ```
  $ pip install openpyxl
  ```

le package *openpyxl* permet de lire et d'écrire dans des fichiers Excel au format *xlsx, xlsm, xltx* ou *xltm*. Il comprend les modules *et-xmlfile* et *jdcal*.

- ```
  $ pip install xlrd
  ```

  le module *xlrd* extrait des données d'un tableur Excel à partir de la version 2.0 et avant de les formatter.

- ```
  $ pip install biopython
  ```

  le package *biopython* regroupe un ensemble d'outils Python pour le traitement informatique de la biologie moléculaire et comprend le module numpy.

Notons que certains modules utilisés dans ce package sont nativement présents dans la librairie standard Python3. C'est le cas par exemple :

- du module *os* qui fournit une manière portable d'utiliser les fonctionnalités dépendantes du système d'exploitation,
- du module *pickle* qui permet de sérialiser et désérialiser une structure d'objet Python. Il remplace le module primitif *marshal*,
- du module *csv* qui implémente des classes pour lire et écrire des données liées à des feuilles de calcul ou des bases de données au format *csv*,
- du module *shutil* qui propose des opérations sur les fichiers et collections de fichiers, notamment la copie et suppression de fichiers,
- du module *subprocess.run* qui permet de gérer de nouveaux processus, de se connecter à leurs flux d'input/output/erreurs. Il remplace plusieurs modules dépréciés : *os.system, os.spawn*, os.popen*, popen2.*, commands.** ,
- du module *pathlib* qui
- du module *argparse* qui

Le package, une fois installé par l'utilisateur, devra fournir un environnement de travail similaire à celui que nous venons de construire, c'est à dire qu'il devra installer automatiquement les dépendances nécessaires à son bon fonctionnement. Ce processus devra rester transparent pour l'utilisateur.

On pourra retrouver les modules installés via *pip* dans un fichier *requirements.txt* ou similaire qu'on pourra fabriquer en gelant l'état de l'environnement à un moment précis grâce à une instruction du type

```
$ pip freeze > requirements.txt
```

Notons que l'outil Poetry quant à lui, regroupe ces modules dans un fichier appelé *pyproject.toml*.

2. L'environnement de test

3. L'outil d'empaquetage

Le PyPA recommande l'utilisation de :

- *setuptools* pour définir des projets et créer des sources de distribution,
- *pipenv* pour la gestion des dépendances de packages lors du développement d'applications,
- *venv* pour isoler les dépendances particulières d'une application et créer un environnement de travail,
- *conda* permettant de fournir un environnement de travail favorable aux projets scientifiques, avec notamment tous les modules essentiels pré-installés,
- *buildout* pour les projets de développement Web,
- *poetry* pour un besoin particulier non couvert par *pipenv*,
- *pip* pour l'installation de librairies à partir de PyPI *Python Package Index*.

Au regard de ces recommandations, nous avons testé les outils suivants dans le but de définir un environnement de développement et de construire un package incorporant les dépendances requises.

pipenv est un gestionnaire de haut niveau pour les environnements, les dépendances et les packages Python. Contrairement à *virtualenv*, *pipenv* distingue les dépendances du projet et les dépendances des dépendances du projet. Par ailleurs, *pipenv* différencie le mode développement du mode production. Il offre l'avantage de bien fonctionner sur Windows. Toutefois, la communauté Python l'a peu mis à jour depuis 2018.

Anaconda est une distribution de logiciels multiplateformes (Windows, Linux, MacOS) qui facilite l'installation des librairies scientifiques *Numpy* et *Scipy*, ce qui est particulièrement intéressant dans le cas des plateformes Windows où ce processus est plus complexe. Elle incorpore une librairie open-source appelée *conda* permettant la gestion des dépendances, de l'environnement de travail ainsi que la création de packages. *Anaconda* semble être approprié au projet, mais c'est une distribution trop lourde pour être intégrée à notre package et *Miniconda*, qui ne comporte que Python, *conda* et *pip*, ne répond pas aux besoins du projet.

Nous avons tout d'abord cherché à construire le package manuellement, à partir de *pipenv* associé à *pip*, puis de *conda* qui dispose d'une riche bibliothèque. Cet effort s'est avéré laborieux et a révélé des incompatibilités lors du *build* qui n'ont pas permis de valider les exigences de la plateforme *testPyPI*.

Nous avons donc décidé de construire notre package en utilisant *Poetry*, qui est un outil complet multiplateformes autour duquel la communauté Python reste très active. Il propose à la fois la gestion des dépendances, l'empaquetage (création d'une structure pour un projet et la génération de fichiers de configuration et de manifestes) et la publication. *Poetry* automatise ces différents procédés et facilite grandement le travail. Nous étudions en détail l'utilisation de cet outil dans la section suivante.

5

Construction du package avec Poetry

1. Création du package et gestion des dépendances

Pour créer notre package, il ne faut pas commencer par définir un dossier du nom du package, car cela sera automatiquement fait par Poetry. En revanche, il faut s'assurer que Python utilise bien la version 3 pour tout le travail de construction, ce qui se traduit par exemple par une variable d'environnement du type **~/.poetry/_vendor/py3.6** en réponse à l'instruction

```
poetry build
```

Pour parvenir à un tel résultat, et comme expliqué précédemment, on commence par choisir une version appropriée de Python dans le répertoire parent dans lequel on souhaite créer notre package (ici la version 3.6.2)

```
$ cd biologie
$ pyenv versions
* system
2.7.10
3.6.2
3.8.0
$ pyenv local 3.6.2
$ python --version
```

Puis la création du projet se fait à l'aide de la commande

```
poetry new crisprbuilder_tb
```

qui permet de générer le squelette de l'application contenant les éléments suivants

```
├── crisprbuilder_tb
│   └── __init__.py
├── pyproject.toml
├── README.rst
└── tests
    ├── __init__.py
    └── test_crisprbuilder_tb.py
```

- le répertoire **crisprbuilder_tb** contient initialement le fichier *__init__.py* et c'est dans ce répertoire que nous allons rajouter tous les composants principaux du code *__main__.py, fonctions.py, bdd.py* et les répertoires **data**, **doc**, **REP** et **tmp** avec leurs contenus. En effet, la création de l'archive va compresser ce répertoire **crisprbuilder_tb** afin de le publier. Il devra contenir tout ce qui est nécessaire au fonctionnement de l'application. Lors de sa création, le fichier *__init__.py* contenait uniquement la version du package. Pour éviter tout conflit de version avec le fichier *pyproject.toml*, nous vidons le fichier *__init__.py* du répertoire *crisprbuilder_tb*
- le répertoire **tests** contient les tests unitaires relatifs au bon fonctionnement du package
- le fichier *README.rst* fournit une brève description du package, et nous le modifions au format markdown *README.md* à l'usage plus répandu
- le fichier *pyproject.toml* remplace les anciens standards de définition de packages : *setup.exe* et *requirements.txt*. Il précise notamment le nom du package, sa version, sa description, l'emplacement de son dépôt (par exemple sur GitHub), l'adresse email de l'auteur du package, et la version des dépendances
- nous rajoutons à la racine du package un fichier *LICENSE*, un fichier *.gitignore* pour la gestion des versions.

Notons qu'il est préférable de choisir un nom de package en minuscules pour éviter toute confusion car, bien que les majuscules soient conservées sur PyPI, l'appel du package en ligne de commande devra s'effectuer en minuscules. En outre, par convention il faut éviter le symbole '-' qui est automatiquement remplacé par '_' lors de la création du package.

En ce qui concerne la gestion des versions de dépendances, il faut tout d'abord se placer dans le répertoire **crisprbuilder_tb** puis utiliser l'instruction

```
$ poetry add nom_dependance
```

qui assure la compatibilité de la dépendance *nom_dependance* avec le package *crisprbuilder_tb*. Il est également possible d'imposer certaines contraintes sur les versions des dépendances ou encore de rentrer manuellement les dépendances dans le fichier *pyproject.toml*, mais l'instruction

```
$ poetry add nom_dependance
```

offre l'avantage de chercher automatiquement une version compatible de la dépendance, puis de l'inscrire dans *pyproject.toml*.

Il faut donc ajouter les dépendances suivantes au projet :

```
$ cd crisprbuilder_tb
$ poetry add python3
$ poetry add xmltodict
$ poetry add openpyxl
$ poetry add xlrd
$ poetry add biopython
```

Le fichier *pyproject.toml* comporte maintenant les versions suivantes

```
[tool.poetry.dependencies]
xlrd = "1.2.0"
xmltodict = "0.12.0"
biopython = "1.76"
```

MANQUE OPENPYXL ET PYTHON

On peut ensuite compléter manuellement ce fichier pour fournir des informations d'ordre général du type

```
[tool.poetry]
name = "crisprbuilder_tb"
version = "0.1.2"
description = "Collect and annotate Mycobacterium tuberculosis WGS data for
CRISPR investigations."
authors = ["stephane-robin <robin.stephane@outlook.com>"]
license = "MIT"
readme = "README.md"
homepage = "https://github.com/stephane-robin/crisprbuilder_tb.git"
repository = "https://github.com/stephane-robin/crisprbuilder_tb.git"
keywords = ["tuberculosis", "CRISPR"]
```

La rubrique [tool.poetry.dev-dependencies] définit le module qui procèdera aux tests unitaires concernant le fonctionnement du package.

La rubrique [build-system] définit les versions de *Poetry* et *poetry.masonry.api* nécessaires à la construction du package.

Pour installer ensuite les dépendances du projet au sein du package, il est nécessaire d'utiliser l'instruction

```
$ poetry install
```

qui crée les fichiers *poetry.lock* et *setup.py*, ainsi que le répertoire **crisprbuilder_tb.egg-info** comprenant le code, les ressources et les métadonnées du projet. Ce format facilite la désinstallation et la mise à niveau du code, car le projet est essentiellement autonome dans un unique répertoire. Il autorise ainsi l'installation de plusieurs versions d'un même projet simultanément. Il s'agit d'un format adapté à la distribution de plugins pour des applications extensibles et des frameworks, similaire au format *.jar* en Java. Il permet également de distribuer les gros packages en entités séparées. De son côté, *poetry.lock* empêche les dépendances de télécharger la dernière version au moment de l'installation, en fixant la version utilisable par le package. Finalement, *setup.py* n'est qu'un fichier temporaire servant à la construction de **crisprbuilder_tb.egg-info**.

2. Construction du package et publication

Pour empaqueter le projet, il faut utiliser l'instruction

```
$ poetry build
```

qui va permettre de créer un répertoire source **dist** contenant une archive *crisprbuilder_tb-0.1.2.tar.gz* et une distribution compilée *crisprbuilder_tb-0.1.2-py3-none-any.whl*. Attention, si la distribution comporte la mention "py2" au lieu de "py3", cela signifie que la construction sous Poetry a été réalisée à partir de Python2. Il faut donc bien s'assurer de choisir une version de Python3 avant la création du package.

On peut vérifier la conformité du package avec l'instruction

```
$ poetry check
```

qui renvoie

```
All set !
```

si le package ne comporte aucune discordance et peut être publié.

Avant d'être publié, la structure du package est alors la suivante

```
.
├── crisprbuilder_tb
├── crisprbuilder_tb.egg-info
├── dist
├── poetry.lock
├── pyproject.toml
├── README.md
└── tests
```

Le dossier **crisprbuilder_tb** contient les éléments suivants

```
.
├── bdd.py
├── data
│   ├── 1_3882_SORTED.xls
│   ├── Brynildsrud_Dataset_S1.xls
│   ├── lineage.csv
│   ├── NC_000962.3.txt
│   ├── spoligo_new.fasta
│   ├── spoligo_old.fasta
│   ├── spoligo_vitro.fasta
│   └── spoligo_vitro_new.fasta
├── doc
│   ├── crisprbuilder_tb.ipynb
│   ├── crisprbuilder_tb.md
│   ├── femto.png
│   ├── LICENSE
│   ├── selection.png
│   └── tree.png
├── fonctions.py
├── __init__.py
├── __main__.py
├── __pycache__
│   ├── bdd.cpython-36.pyc
│   ├── fonctions.cpython-36.pyc
│   ├── __init__.cpython-36.pyc
│   └── __main__.cpython-36.pyc
├── REP
│   └── sequences
│       └── __init__.py
└── tmp
    └── __init__.py
```

Le dossier **crisprbuilder_tb.egg-info** contient les éléments suivants

```
.
├── dependency_links.txt
├── PKG-INFO
├── requires.txt
├── SOURCES.txt
└── top_level.txt
```

Le dossier **dist** contient les éléments suivants

```
.
├── crisprbuilder_tb-0.1.24-py2.py3-none-any.whl
└── crisprbuilder_tb-0.1.24.tar.gz
```

C'est le fichier *README.md* qui est utilisé pour la description du projet par PyPI sur le site ou encore testPyPI sur le site test.pypi.org/project/crisprbuilder_tb

Avant de publier le package, il faut tester son comportement à l'installation en dehors de l'environnement de développement. Pour cela, il est possible de le publier officieusement sur la plateforme de test *testPyPI*. Il faut alors se placer à la racine du package et lancer en ligne de commande l'instruction

```
$ twine upload --repository testpypi dist/*
```

Il est donc possible de télécharger un package provenant d'index différents de PyPI. Par exemple, l'installation de *crisprbuilder_tb* à partir de *testPyPI* se fait à l'aide de l'instruction

```
$ pip3 install -i https://test.pypi.org/simple/ crisprbuilder_tb
```

ou si on précise la version du package

```
$ pip3 install -i https://test.pypi.org/simple/ crisprbuilder_tb==0.1.2
```

A noter qu'il ne faut utiliser de guillemets autour de la version du package.

Lorsque le package est finalement prêt pour être publié sur PyPI, on utilise l'instruction

```
$ poetry publish
```

Pour cela, l'auteur du package doit être enregistré sur PyPI avec un identifiant et un mot de passe. A partir de ce moment-là, le package est rendu disponible publiquement.

L'installation du package par un utilisateur quelconque se fait maintenant grâce à l'instruction

```
$ python3 -m pip install nom_package
```

Les explications relatives à l'utilisation pratique de *Poetry* sont reprises dans le tutoriel suivant que nous venons de publier sur YouTube : adresse YuouTube

3. Installation du package

Au moment de l'installation, que ce soit à partir de *PyPI* ou d'un autre index tel que *testPyPI*, l'utilisateur ne choisit pas l'emplacement d'installation. Celui-ci est définit automatiquement par le sys-

tème en fonction des variables d'environnement. Par exemple, si on utilise une version particulière de Python avec *pyenv*, alors le package pourra se trouver dans **~/.pyenv/versions/3.6.5/lib/python3.6/**. Il est également fréquent de le retrouver dans **~/.local/lib/python3.6/site-packages/** avec la plupart des différentes packages. Comme on ne connaît pas avec certitude l'emplacement dans lequel le package va s'installer, il convient de définir une variable d'environnement qui pointe vers l'emplacement d'installation.

Lors de l'exécution du package, il est nécessaire de l'appeler par son nom en utilisant l'indice "-m" signifiant "module". Il n'est pas utile de préciser l'utilisation de Python3 si le répertoire courant travaille déjà avec cette version. En revanche, les chemins relatifs dans les modules du package ne devront pas utiliser le nom crisprbuilder_tb car celui-ci ne sera pas reconnu, une fois à la racine du package. EXPLICATIONS A CHANGER

Par défaut, Python cherche ses modules et packages dans la variable d'environnement $PYTHONPATH. On peut alors consulter les différents chemins connus par cette variable d'environnement en exécutant l'instruction

```
$ python
⋙ import sys
⋙ print(sys.path)
```

afin d'obtenir une réponse sous forme de liste du type

```
['',
'~/.pyenv/versions/3.6.5/lib/python36.zip',
'~/.pyenv/versions/3.6.5/lib/python3.6',
'~/.pyenv/versions/3.6.5/lib/python3.6/lib-dynload',
'~/.local/lib/python3.6/site-packages',
'~/.pyenv/versions/3.6.5/lib/python3.6/site-packages']
```

4. La structure du package construit par Poetry

Le répertoire **crisprbuilder_tb** contient obligatoirement à la racine les fichiers *__init__.py* et *__main__.py*

Par ailleurs, tous les répertoires comprenant des modules Python doivent contenir un fichier *__init__.py*.

Lors de l'exécution de l'instruction

```
$ poetry new crisprbuilder_tb
```

s'est créé le package *crisprbuilder_tb* disposant de la structure suivante :

- un répertoire **doc** comprenant la documentation nécessaire pour comprendre et utiliser convenablement le package,
- un répertoire **__pycache__** comprenant des fichiers compilés et un fichier *__init__.py*,
- un répertoire **data** comprenant les bases de données utiles au programme, sans aucun module et donc aucun fichier *__init__.py*,
- un répertoire **REP** comprenant un répertoire séquences qui contient les résultats de recherche au sujet de différents SRA,

- un répertoire **tmp** comprenant un fichier nb.txt qui change pour chaque SRA à la racine du package, les fichiers *__init__.py*, *__main__.py*, *fonctions.py* et *bdd.py*.

6

Le package crisprbuilder_tb

1. Fonctionnement du package crisprbuilder_tb

La compréhension des chemins d'accès est au coeur du fonctionnement d'un package

Le package *crisprbuilder_tb* ne fonctionne pas comme un module qu'on importerait en début de code Python afin d'en utiliser les différentes méthodes.

Par ailleurs, il n'est pas prévu que l'utilisateur se place dans le package pour exécuter directement le module *__main__.py* en écrivant en ligne de commande

```
$ python crisprbuilder_tb --collect ERR2704808
```

sans le "-m", car dans ce cas l'exécution entraînerait la levée d'une exception due aux différents chemins d'accès qui seraient erronés. (explications des chemins d'accès ci-dessous)

L'exécution du package est réalisée en ligne de commande, et une fois que l'utilisateur a écrit

```
$ python -m crisprbuilder_tb --collect ERR2704808
```

l'interpréteur va chercher le package à son emplacement de stockage XXX et l'exécute comme si ce dernier se trouvait dans le répertoire courant. Pourtant, le package ne se trouve pas dans le répertoire courant et tous les chemins d'accès aux bases de données et modules du package doivent tenir compte de la variable d'environnement placée au début de chaque chemin d'accès.

2. Description détaillée du package

Le package est décrit en détail dans le document *crisprbuilder_tb.md* présent dans le répertoire **doc** du package. Nous reprenons ci-dessous les principaux éléments aprés les avoir traduits en français.

DOC

7

Tester le package

Les tests font partie du processus de développement et sont effectués dans chacune des phases de création du code afin de pouvoir continuer à l'étape suivante. Ainsi, les tests unitaires durant la création de chaque fonction, les tests d'intégration au moment

Plusieurs niveaux de tests doivent être effectués pour valider le package avant publication. Ils nécessitent des environnement de travail différents :

- l'environnement de développement permet de procéder aux tests unitaires effectués durant la création de chaque fonction, aux tests d'intégration effectués au moment où l'on regroupe tout le code, aux tests de performance et aux tests de non-régression effectués à chaque étape lors d'une modification,

- l'environnement de validation permet de procéder aux tests fonctionnels, aux tests CLI, aux tests de configuration et aux tests d'installation effectués lors en dernier.

Il faut donc comprendre le fonctionnement, puis tester chaque portion de code et chaque fonction individuellement avec des paramètres choisis avant de l'incorporer dans le module *__main__.py*. Il est donc nécessaire d'adapter au reste du code les fonctions déjà écrites, les documenter en anglais en vue d'une utilisation ultérieure, les rendre performantes en utilisant des compréhensions de listes, éviter les erreurs potentielles en rajoutant des blocs **try ... except** et des méthodes **.setdefault** et **.get**. Il faut également créer de nouvelles fonctions pour éviter la répétition de code.

1. Les tests unitaires

Les tests unitaires consistent à évaluer individuellement les composants de l'application (parties de code autonomes, fonctions, ...), en terme de qualité et validation des résultats attendus.

Les tests unitaires sont d'abord effectués manuellement fonction par fonction en évaluant chaque résultat obtenu.

Puis nous utilisons le framework *unittest* intégré dans la distribution standard de Python3.

> EXPLICATIONS unittest

2. LES TESTS D'INTÉGRATION

Les tests d'intégration permettent de valider l'inter-utilisabilité des différents modules entre eux dans leur environnement d'exploitation définitif. Nous réalisons également les tests d'intégration dans l'environnement de développement que nous avons construit.

Pour cela, nous avons créé le package, nous nous sommes placés à la racine de l'application et avons exécuté le code du module *__main__.py* directement.

Dans le module *__main__.py*, il faut d'abord importer le package *crisprbuilder_tb*, puis importer les modules *crisprbuilder_tb.fonctions.py* et *crisprbuilder_tb.bdd.py* sans oublier le nom du package sans quoi l'interpréteur ne parviendra pas à trouver ces modules même s'ils se situent à la racine du package.

2.1. Amélioration du code avec pylint

pylint fournit une note entre 0 et 10 qui reflète le respect des règles du PEP-8 [1] et du PEP-257 [2], notamment en ce qui concerne la lisibilité du code en terme d'espaces entre les signes, de longueur maximale chaque ligne (80 caractères) et d'indentation pour assurer la compréhension lors d'un retour à la ligne. *pylint* reprend également les erreurs de structure de code, les manques de définition de variables ou les ambiguïtés dans l'utilisation de variables locales portant le même nom. Toutefois, il ne comprend pas bien l'utilisation des compréhensions de listes et des méthodes telles que *.setdefault* pour récupérer la valeur d'une variable dans un dictionnaire. Par ailleurs, *pylint* considère que les modules ne devraient pas excéder 1000 lignes. De ce fait, nous avons développé notre module initial *__main__.py* en trois modules *__main__.py, fonctions.py* et *bdd.py* et nous nous sommes astreints aux règles du PEP-8 pour obtenir une note de 9.72 / 10 pour le module *__main__.py* avec les commentaires suivants :

1. **PEP-8 :** style guide for Python code
2. **PEP-257 :** docstring conventions

```
$ pylint __main__.py
************* Module __main__
__main__.py:36:0: R0914: Too many local variables (57/15) (too-many-locals)
__main__.py:152:23: W0123: Use of eval (eval-used)
__main__.py:160:23: W0123: Use of eval (eval-used)
__main__.py:606:0: R1721: Unnecessary use of a comprehension
(unnecessary-comprehension)
__main__.py:643:0: R1721: Unnecessary use of a comprehension
(unnecessary-comprehension)
__main__.py:679:0: R1721: Unnecessary use of a comprehension
(unnecessary-comprehension)
__main__.py:585:16: W0612: Unused variable 'item2' (unused-variable)
__main__.py:36:0: R0912: Too many branches (88/12) (too-many-branches)
__main__.py:36:0: R0915: Too many statements (317/50) (too-many-statements)
__main__.py:734:0: R0914: Too many local variables (17/15) (too-many-locals)
__main__.py:734:0: R0912: Too many branches (19/12) (too-many-branches)
__main__.py:734:0: R0915: Too many statements (101/50) (too-many-statements)
---------------------------------

Your code has been rated at 9.72/10 (previous run: 9.70/10, +0.02)
```

et une note de 9, / 10 pour le module *fonctions.py* avec les commentaires suivants :

XXX

Le module *bdd.py* ne contenant que le dictionnaire *Origines* n'a pas été évalué par *pylint*.

L'exécution de *pylint* se fait directement en ligne de commande en se plaçant dans le dossier parent du fichier à tester (par exemple ici *__main__.py*) avec l'instruction

```
$ pylint __main__.py
```

S'agissant d'un test statique, le code n'est pas exécuté et la performance du code n'est pas prise en compte lors de l'évaluation faite par *pylint*. Ceci explique que *pylint* peut reprocher l'utilisation injustifiée de listes compréhensives, sans tenir compte de l'efficacité d'un tel traitement.

2.2. Les problèmes liés aux chemins d'accès

L'incompatibilité des chemins d'accès entre les systèmes Posix et Windows a nécessité l'utilisation du module PurePath permettant de construire des chemins sans recourir aux caractères "/" ou "\".

3. Les tests fonctionnels

Les tests fonctionnels permettent de vérifier la conformité du package développé par rapport à l'objectif et aux fonctionnalités initiales. A partir des tests fonctionnels, nous n'utilisons plus l'environnement de développement, mais un environnement de test qui peut être soit une machine virtuelle, soit un environnement neutre dénué de toute spécification, mais capable d'installer le package.

L'objectif initial était de fournir un package qui fournit des informations au *SRA* dont la référence serait saisie en ligne de commande par un utilisateur. Ce dernier pourrait également fournir une liste

de *SRA* dans un fichier texte avec un *SRA* par ligne. Il pourrait également consulter et modifier la base de données *lineage.csv*.

Nous fournissons au package un fichier au format *.txt* comprenant plus qu'un *SRA* par ligne. RESULTATS

4. Les tests CLI

L'objectif des tests CLI est de vérifier que les lignes de commandes prévues pour l'exécution du code fournissent bien le résultat escompté. Pour cela, nous testons les commandes

```
$ python -m crisprbuilder_tb --help
```

fournit bien les informations relatives aux options du package, à savoir A COMPLETER ? ? ?

```
$ python -m crisprbuilder_tb --change 0
```

```
$ python -m crisprbuilder_tb --remove 0
```

```
$ python -m crisprbuilder_tb --add 0
```

```
$ python -m crisprbuilder_tb --print 0
```

La fonctionnalité ——*collect* a déjà été testée précédemment.

5. Les tests de configuration

Les tests de configuration cherchent à évaluer l'impact des différents systèmes d'exploitations et environnements matériels sur le fonctionnement d'une application.

Nous cherchons donc à définir la configuration matériele minimale afin que le package fonctionne correctement, ainsi que l'utilisabilité des différents systèmes d'exploitation.

CONFIG MINI

6. Les tests de performance

L'objectif des tests de performance serait de valider la capacité des serveurs et réseaux pour supporter les charges d'accès importantes. Ceci ne s'applique pas vraiment à notre package, qui n'utilise, lors de ses requêtes fastq et blast, que les serveurs de NCBI.

Ne s'agissant pas d'une application Web, l'utilisation simultanée de plusieurs ordinateurs exécutant le package n'a pas d'incidence sur celui-ci. Si les serveurs de NCBI sont occupés, les réponses obtenues n'en seront que différées.

En revanche, augmentation des performances

La taille importante des fichiers sur lesquels nous travaillons génère des délais importants de traitement. Au début de notre travail sur le code, nous avons cherché à transformer les instructions Linux

appelées par les méthodes Python **subprocess.run** et **os.system** en fonctions Python pour améliorer la portabilité de l'application en évitant de se limiter aux systèmes Linux. Lors des premiers tests d'intégration, le temps de réponse du programme était tellement élevé que nous avons Nous avons constaté lors des premiers tests d'intégration un fort ralentissement du programme lorsque ces modifications étaient effectuées sur l'ensemble du code. fortement le code lors des premiers tests de fonctionnement global de l'application, au vu de la taille importante des fichiers à lire par l'interpréteur. Nous avons donc décidé de revenir à l'utilisation de **subprocess.run** et **os.system**. Si cette démarche n'a aucune incidence pour la plateforme MacOS, elle imposerait en revanche un changement de code pour la plateforme Windows. Or, en fonction des versions du systéme Windows, l'utilisateur dispose d'une invite de commandes ou d'un PowerShell dont les instructions diffèrent. Nous avons donc choisi de conserver les appels aux instructions en lignes de commande pour les système Posix et d'utiliser des fonctions Python pour les systèmes Windows. Cela entraîne une relative lenteur de traitement pour ces derniers.

Pour mieux comprendre ce ralentissement, prenons un exemple sur un traitement simple consistant à concaténer deux fichiers .fasta. L'instruction suivante

```
os.system("cat SRR8368696_1.fasta SRR8368696_2.fasta >
SRR8368696_shuffled_system.fasta")
```

est bien plus rapide qu'une fonction Python que nous pourrions définir pour concaténer et qui permettrait d'améliorer la portabilité du programme. En effet, nous constatons que le code ci-dessous exécuté 10 fois sur un système Ubuntu 18.04 LTS avec un processus Intel Pentium de 1,6 GHz 64 bits 4 coeurs

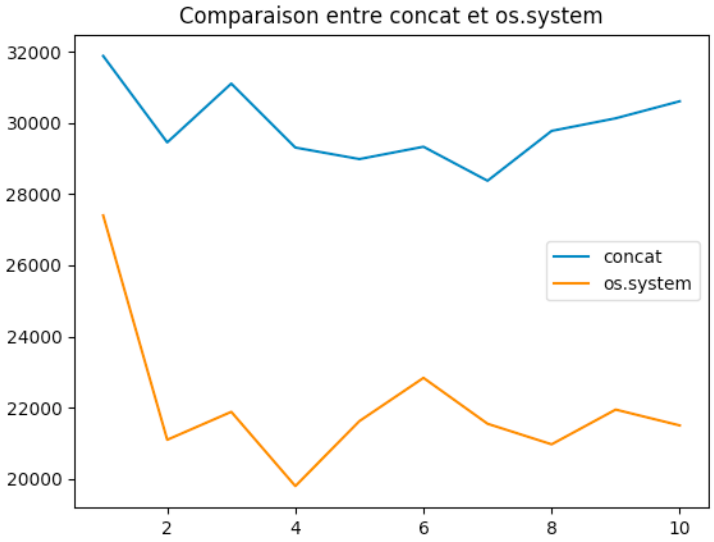
```
import time
import os

def concat(p_f1, p_f2, p_shuffled):
    with open(p_f1, 'r') as file_1, open(p_f2, 'r') as file_2, \
            open(p_shuffled, 'w') as f_shuffled:
        lignes_1 = file_1.readlines()
        for elt in lignes_1:
            f_shuffled.write(elt)
        lignes_2 = file_2.readlines()
        for elt in lignes_2:
            f_shuffled.write(elt)

debut_concat = int(round(time.time() * 1000))
concat('SRR8368696_1.fasta', 'SRR8368696_2.fasta', 'SRR8368696_shuffled_concat.fasta')
fin_concat = int(round(time.time() * 1000))
temps_concat = fin_concat - debut_concat
print(temps_concat)

debut_system = int(round(time.time() * 1000))
os.system("cat SRR8368696_1.fasta SRR8368696_2.fasta > SRR8368696_shuffled_system.fasta")
fin_system = int(round(time.time() * 1000))
temps_system = fin_system - debut_system
print(temps_system)
```

affiche un temps moyen d'exécution de l'instruction **os.system** de 19,3 secondes, alors qu'il est de 26,7 secondes lorsqu'on exécute la fonction *concat*.



concat (ms)	31881	29452	31104	29307	28984	29330	28374	29775	30129	30608
os.system (ms)	27403	21099	21884	19798	21624	22838	21549	20971	21947	21502

Cette différence de quelques secondes pour une seule fonction entraîne par conséquence de nombreuses minutes de retard sur l'ensemble du code lors de son exécution.

6.1. Amélioration des performances

7. Les tests d'installation

Une fois que l'application a été validée et publiée sur testPyPI, nous avons contrôlé la documentation au sein des modules et des fonctions, dans le fichier *README.md* et surtout dans le fichier *crisprbuilder_tb.md*.

Nous avons également vérifié le bon fonctionnement de l'installation sur un système Ubuntu 18.04 LTS, un MacOS Sierra et un Windows Vista. Les ordinateurs utilisés ne disposaient pas de l'environnement de travail nécessaire à l'application avant l'installation de celle-ci. Les installations ont été testées de bout en bout avec les commandes

COMMANDES D'INSTALLATION

Le même procédé a finalement été appliqué au package publié sur PyPI.

8. Les tests de non-régression

Ce type de tests permet de vérifier que les modifications apportées au programme durant l'ensemble de la phase de test n'ont pas altérées le fonctionnement du package.

Pour cela, lorsqu'un changement est effectué, les résultats obtenus doivent être comparés aux résultats pécédemment obtenus. Nous nous sommes alors exclusivement basé sur les commandes

```
$ python -m crisprbuilder_tb --collect ERR2704808
```

et

```
$ python -m crisprbuilder_tb --print 0
```

pour procéder aux vérifications aprés modification de code.

L'utilisation de l'outil de versioning *Git* a constitué une aide précieuse au cours de chaque modification significative de code. Ce stage a été l'occasion de manipuler quotidiennement les principales commandes *Git*. L'utilisation conjointe de *Poetry* et de *Git* qui construisent tous les deux un dossier **crisprbuilder_tb** au moment de la création d'un package par

```
$ poetry new crisprbuilder_tb
```

et au moment de la création d'un repository par

```
$ git clone https://github.com/stephane-robin/crisprbuilder_tb.git
```

Conclusion

Ce stage m'a ouvert l'esprit vers le domaine de la bioinformatique et m'a permis de réfléchir à

Notons qu'à terme, l'objectif du package sera de proposer une reconstitution du *CRISPR*.

L'élaboration de ce package m'a permis de développer les compétences concrètes suivantes :

- création d'un environnement de développement,
- travail sur le code, concentration sur l'augmentation des performances de traitement,
- documentation du package PEP-257 docstring conventions
- création d'une Command Line Interface,
- création et publication d'un package
- travail sur la portabilité d'une application
- création d'un environnement de test,
- tests du package (unitaires, ... test pylint, module unittest,
- utilisation de l'outil de versionnement *Git*
- utilisation d'un environnement de développement intégré

PROBLEME

mv /home/stephane/.local/lib/python3.6/site-packages/crisprtest /home/stephane/Téléchargements

répétition de pip install

9. Les principaux éléments du code

dico est composé de la façon suivante :

Un fichier pkl tel que dico_africanum.pkl est créé par pickle et contient un flux d'octets représentant les objets à sérialiser.

pickle permet aux objets d'être sérialisés en fichiers sur disque et désérialisés dans le programme au moment de l'exécution.

Références

[1] Comas I. et al. *Out-of-Africa migration and Neolithic coexpansion of Mycobacterium tuberculosis with modern humans*, Nat Genet. 45(10) : 1176–1182. doi :10.1038/ng.2744

[2] Coll F. et al., *SpolPred : rapid and accurate prediction of Mycobacterium tuberculosis spoligotypes from short genomic sequences*, Bioinformatics. 28(22) :2991–3

[3] Brynildsrud O.B. et al., *Global expansion of Mycobacterium tuberculosis lineage 4 shaped by colonial migration and local adaptation*, 4(10) : eaat5869. doi : 10.1126/sciadv.aat5869

[4] Driscoll J. R., *Spoligotyping for molecular epidemiology of the Mycobacterium tuberculosis complex*, 551 :117-28. doi : 10.1007/978-1-60327-999-4 10

[5] Jinek M. et al, *A programmable dual-RNA-guided DNA endonuclease in adaptive bacterial immunity*, 337(6096) :816-21. doi : 10.1126/science.1225829

[6] Gori A. et al, *Spoligotyping and Mycobacterium tuberculosis*, 11(8) : 1242–1248. doi : 10.3201/1108.040982

[7] Perry S. et al., *Infection with Helicobacter pylori is associated with protection against tuberculosis*, 5(1) :e8804. doi : 10.1371/journal.pone.0008804

[8] Perry S. et al, *The immune response to tuberculosis infection in the setting of Helicobacter pylori and helminth infections*, 141(6) : 1232–1243. doi : 10.1017/S0950268812001823

[9] Xia E. et al., *SpoTyping : fast and accurate in silico Mycobacterium spoligotyping from sequence reads*, 8 :19. doi 10.1186/s13073-016-0270-7

[10] Dale JW. et al., *Spacer oligonucleotide typing of bacteria of the Mycobacterium tuberculosis complex : recommendations for standardised nomenclature.*, 5(3) :216–9

[11] Iwai H et al., *CASTB (the comprehensive analysis server for the Mycobacterium tuberculosis complex) : A publicly accessible web server for epidemiological analyses, drug-resistance prediction and phylogenetic comparison of clinical isolates. Tuberculosis.*, 95(6) :843–4

[12] Demay C. et al., *SITVITWEB - A publicly available international multimarker database for studying Mycobacterium tuberculosis genetic diversity and molecular epidemiology.*, Infect Genet Evol. 12 :755–66

[13] McGovern Institute Channel, *Genome Editing with CRISPR-Cas9*, https://www.youtube.com/watch?v=2pp17E4E-O8

[14] O'Neil M.B. et al., *Lineage specific histories of Mycobacterium tuberculosis dispersal in Africa and Eurasia*, bioRxiv. 10.1101/210161

[15] Ratovonirina N. H., *Etudes descriptive, épidémiologique, moléculaire et spatiale des souches Mycobacterium tuberculosis circulant à Antananarivo, Madagascar*, Thèse de Doctorat de l'Université Paris-Saclay

[16] Kamerbeek et al., *Simultaneous detection and strain differentiation of Mycobacterium tuberculosis for diagnosis and epidemiology*, 35(4) : 907–914

[17] Mendis C. et al., *Insight into genetic diversity of Mycobacterium tuberculosis in Kandy, Sri Lanka reveals predominance of the Euro-American lineage*, International Journal of Infectious Diseases 87 84-91

[18] ,Gomgnimbou M. K. et al., *Tuberculosis-Spoligo-Rifampin-Isoniazid Typing : an All-in-One Assay Technique for Surveillance and Control of Multidrug-Resistant Tuberculosis on Luminex Devices*, 51(11) :3527-34. doi : 10.1128/JCM.01523-13

[19] Couvin D. et al., *SpolSimilaritySearch - A web tool to compare and search similarities between spoligotypes of Mycobacterium tuberculosis complex*, 105 :49-52. doi : 10.1016/j.tube.2017.04.007

https://packaging.python.org/guides/

https://realpython.com/pypi-publish-python-package/

Index