# Rapport de stage
# Publication du package crisprbuilder_tb

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

6 août 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 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.

Je tiens également à remercier Jean Marc Gervais qui a eu la gentillesse de bien vouloir relire le présent rapport de stage et me proposer des commentaires pertinents à son sujet.

## 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] et exécutable. Il doit donc être rendu fonctionnel et plus efficace si possible. Son exécution doit se réaliser en utilisant une interface en ligne de commande. L'application doit finalement être empaquetée pour qu'un utilisateur puisse l'exploiter quelque soit sa plateforme (Linux, macOS ou Windows), une fois installée. 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.

Ainsi, l'élaboration de ce package a nécessité de passer par différentes étapes permettant de développer les compétences suivantes :

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

- amélioration des connaissances Python car il a fallu reconstruire le programme à partir d'éléments de code et chercher à optimiser les performances de traitement,
- création d'un notebook de documentation en anglais et d'une docstring pour le package dans le respect des conventions du PEP-257,
- travail de portabilité du code en le modifiant en fonction des différents systèmes d'exploitation Posix ou Windows,
- création d'un environnement de développement car il a fallu tester le code durant son élaboration,
- utilisation d'un environnement de développement intégré tel que *PyCharm* pour faciliter l'écriture du code, le debuggage et travailler dans des conditions réelles de développement. Cela implique également la gestion des modules installés pour les rendre compatibles avec les interpréteurs Python sous *PyCharm*,
- utilisation de l'outil de versionnement Git, nécessaire pour sauvegarder le projet et pour revenir sur une version fonctionnelle après une modification erronée,
- création d'une interface en ligne de commande, telle que souhaitée par le projet,
- choix de l'outil empaquetage et de gestion des dépendances. Création et publication d'un package sur PyPI, impliquant la bonne compréhension du fonctionnement des packages ainsi que des chemins d'accès aux différents fichiers,
- création d'une campagne de tests du package (tests unitaires, d'intégration, de non-régression, de validation, de performance, de configuration),
- création de machines virtuelles pour effectuer des tests sur différents systèmes d'exploitation, et création d'un environnement de test sur ces machines virtuelles
- travail de compréhension des mots-clés en bio-informatique afin de pouvoir exercer 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.

Il faut se confronter au moins une fois aux compétences pratiques énumérées ci-dessus afin d'en découvrir les dysfonctionnements et d'en appréhender les difficultés. Nous allons voir en détail dans ce rapport de stage les différentes étapes suivies, et nous allons expliquer en détail la construction et le fonctionnement d'un package en prenant bien évidemment l'exemple de CRISPRbuilder_TB.

Le package CRISPRbuilder_TB peut s'installer à partir de PyPI où il est répertorié sous le nom de crisprbuilder_tb. Il est présent sous *GitHub* à l'adresse suivante
https://github.com/stephane-robin/crisprbuilder_tb.

## 3. Remarque préalable

Toutes les explications données dans ce rapport de stage ne concernent que les plateformes Linux. Celles-ci doivent être configurées pour exécuter par défaut une version récente de Python 3 (supérieure ou égale à 3.6) et une version de pip compatible avec Python 3.

Définissons par ailleurs 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, des modules C, des bases de données et des métadonnées,
- archive : représentation abstraite d'un ensemble de fichiers (au format *.tar.gz*, *.zip*, ...)
- distribution : une version particulière de modules ou packages (au format *.whl*, *.exe*, ...)

Le code de ce projet a été organisé sous forme de modules, tel que préconisé par le PEP.

# 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.[9] 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.

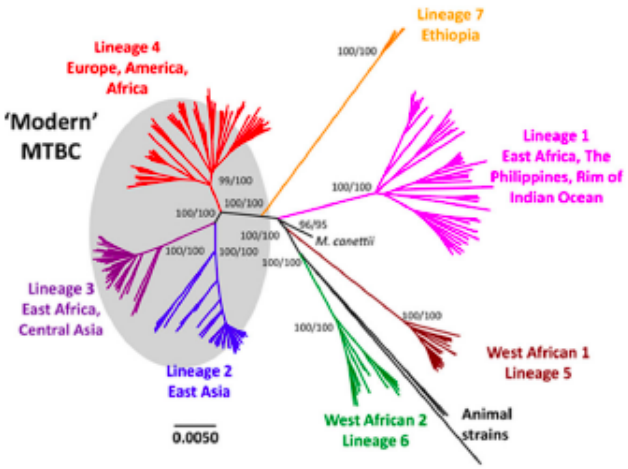



**Figure 1** – Phylogénie du génome complet de MTBC, d'après *Out-of-Africa migration and Neolithic coexpansion of Mycobacterium tuberculosis with modern humans*

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.[9] 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.[15, 16] 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

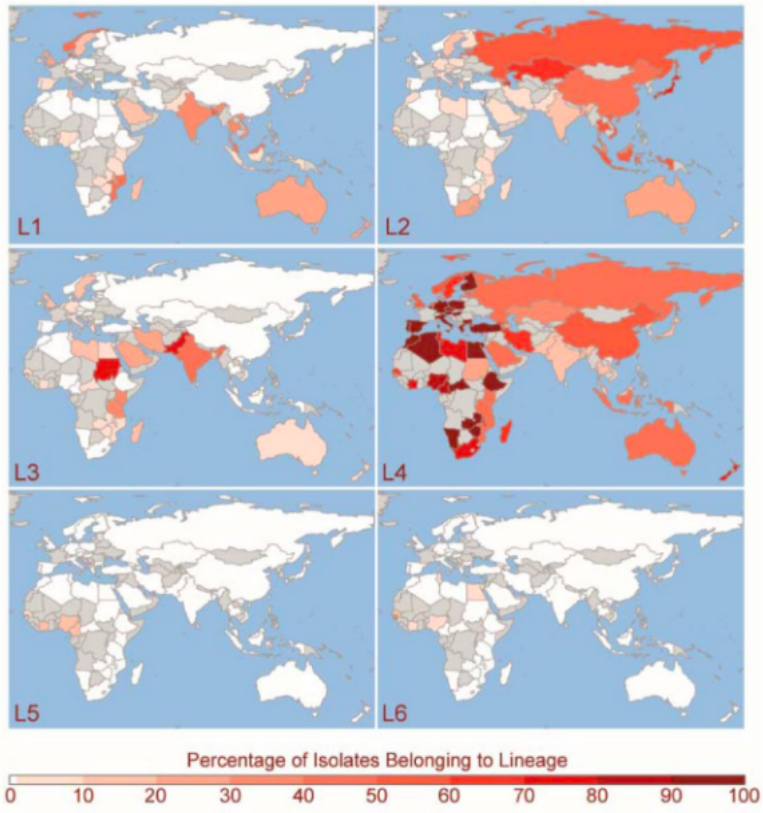



**Figure 2** – Distribution géographique des lignées 1 à 6, d'après *Lineage specific histories of Mycobacterium tuberculosis dispersal in Africa and Eurasia*

contre les 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.[11] 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.[22] 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.[11] 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 $\beta$[11] de l'ARN polymérase[12] de la bactérie. Dans la tuberculose non résistante, la rifampicine se lie à cette sous-unité $\beta$ 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 $\beta$. Dans ce cas, la rifampicine ne peut plus se lier à la sous-unité $\beta$ 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** $\beta$ : 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.

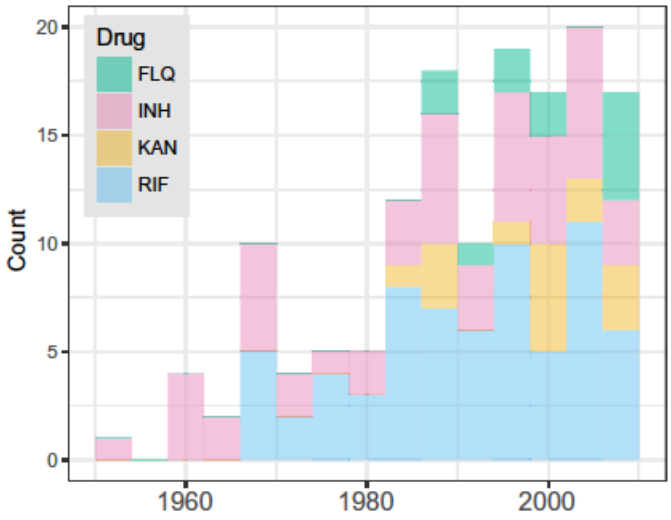



**Figure 3** – Transmission de la résistance ces dernières années à l'échelle mondiale, d'après *Global expansion of Mycobacterium tuberculosis lineage 4 shaped by colonial migration and local adaptation*. FLQ=fluoroquinolone, INH=isoniazide, KAN=aminoglycoside, RIF=rifampicine

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.[20] contient les génotypes de *M. tuberculosis* obtenus à partir des marqueurs moléculaires MIRU et VNTR.

### 3.2. Description du locus CRISPR-Cas

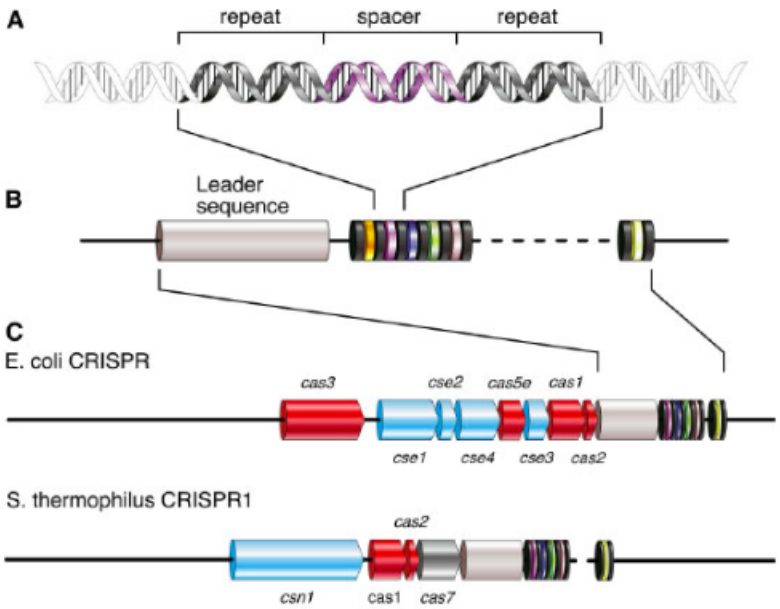



**Figure 4** – Structure du locus CRISPR-Cas, d'après https://www.sinobiological.com/crispr-locus.html

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 [24] 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.

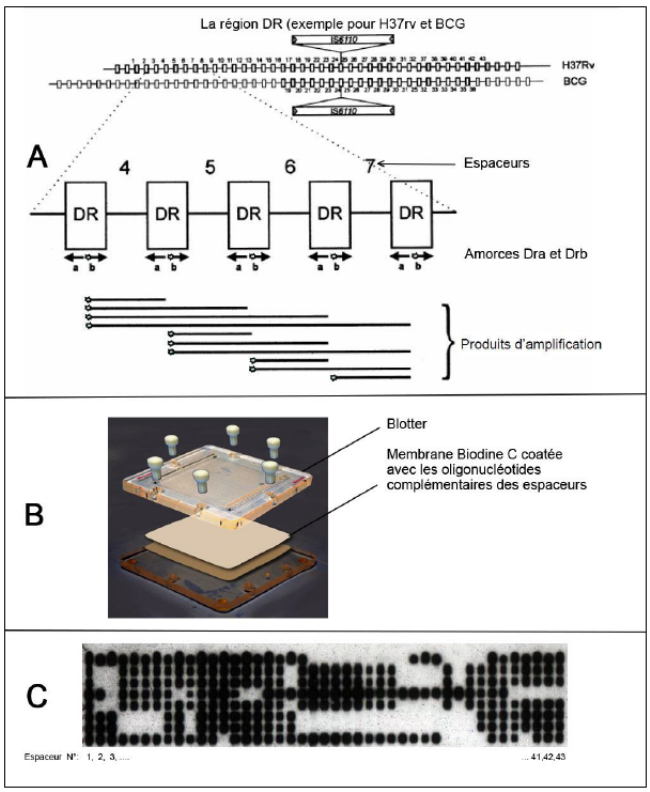



**Figure 5** – Les différentes étapes du spoligotypage d'après *Etudes descriptive, épidémiologique, moléculaire et spatiale des souches Mycobacterium tuberculosis circulant à Antananarivo, Madagascar*

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 [26]. 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 [18].

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.[18] 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.

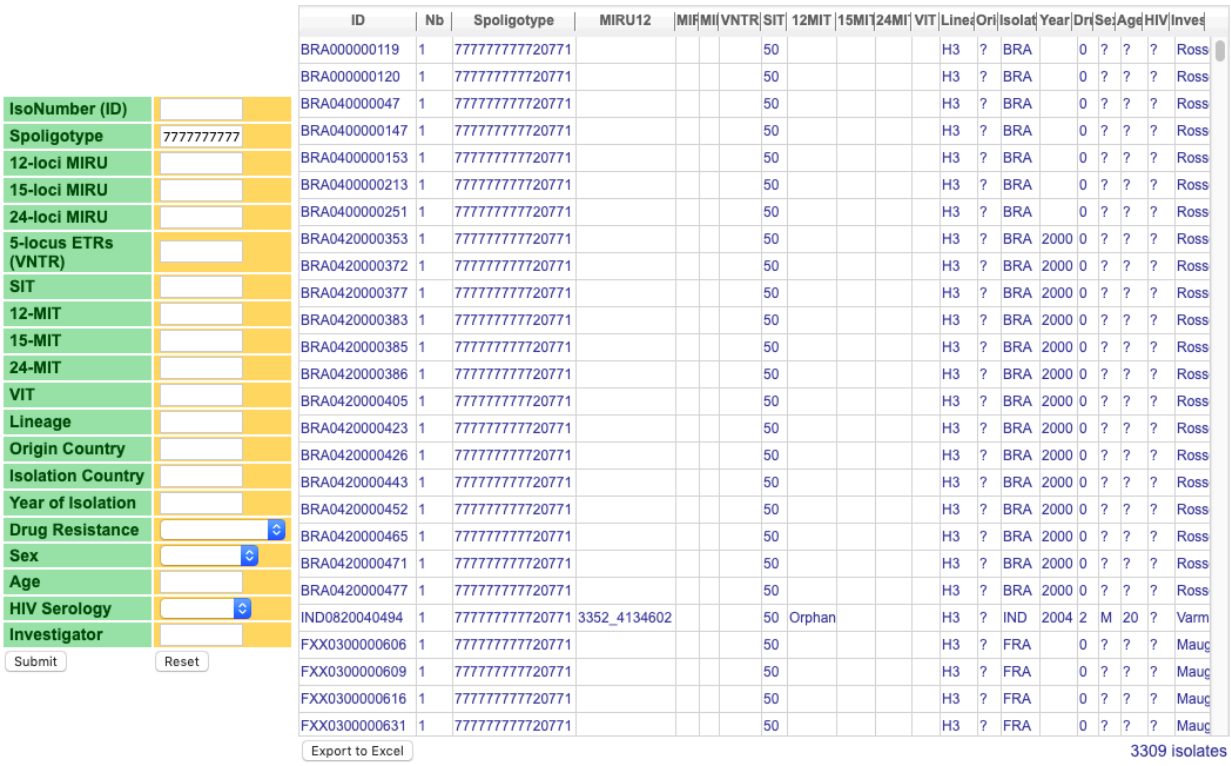



**Figure 8** – Exemple de recherche effectuée sur SITVIT2 à partir du spoligotype 777777777720771 dont le résultat est exploitable au format Excel

Dans leur étude [10], 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.[17], 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 [10] 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 [17] 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*

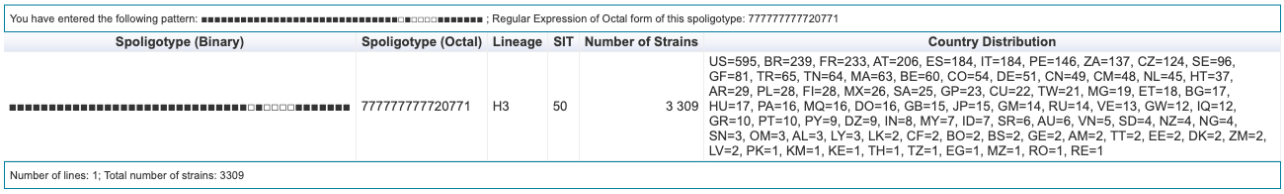
L'étude de Iwai H. et al.[19] envisage une autre possibilité de travail et montre l'intérêt d'une analyse de *M. tuberculosis* à l'aide de serveurs, appelée CASTB, et notamment le spoligotypage. Le Webserver fournit une vue complète des données, mais les performances de chaque outil utilisé ne sont pas 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.[27]. 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



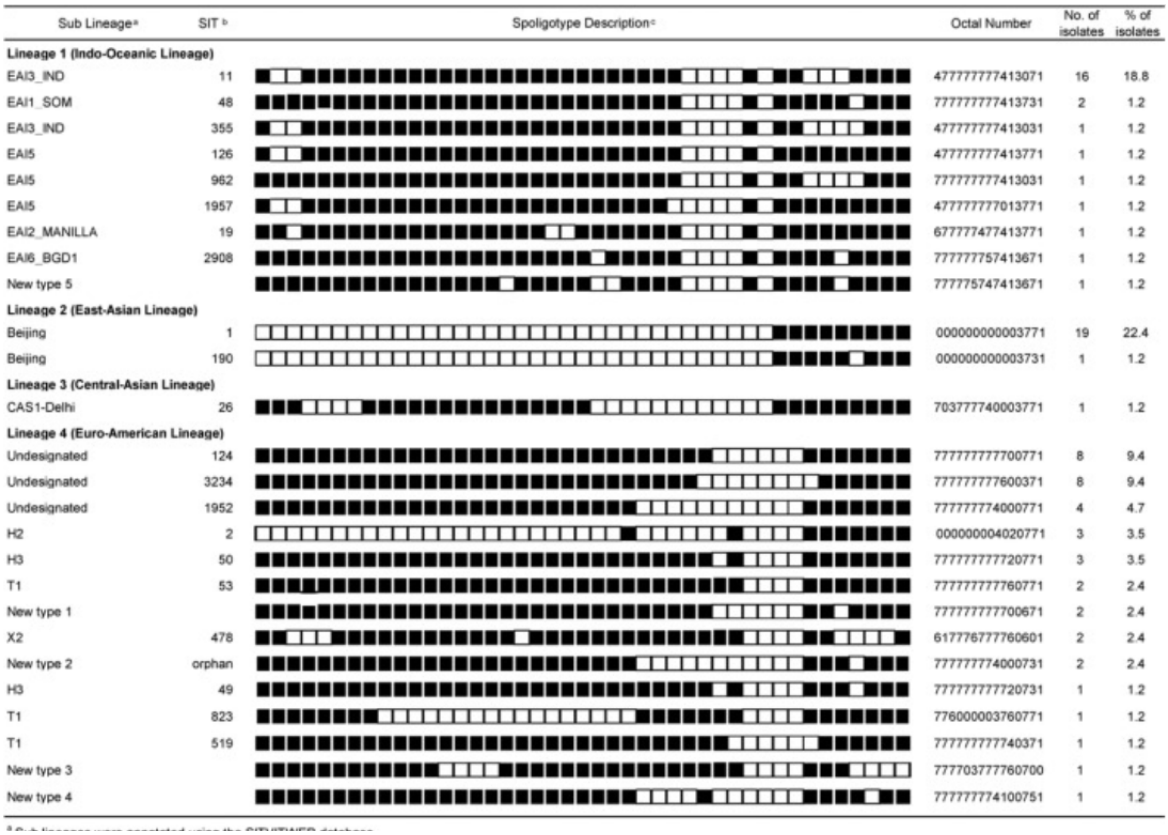
You have entered the following pattern: ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■□■□□□□■■■■■■■ ; Regular Expression of Octal form of this spoligotype: 777777777720771

| 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

Le packaging ou distribution de programmes pose traditionnellement plus de problèmes en Python que dans d'autres langages. En effet, les standards ont évolué de façon concurrentielle plutôt que collaborative. Par ailleurs, la documentation sur Internet est parfois obsolète et entraîne de nombreuses incompréhensions. Nous verrons que l'arrivée de l'outil *Poetry* a grandement facilité la création et la gestion des packages. Des sites tels que `https://realpython.com/python-modules-packages/`, `https://realpython.com/pypi-publish-python-package/`, `https://python-poetry.org/` ou `https://packaging.python.org/overview/` ont également contribué à rendre accessible le packaging au grand public.

## 1. Composition d'un package standard

Un permet de fournir une structure hiérarchisée du programme, pour en faciliter la compréhension et l'utilisation. Il s'agit en fait d'un ensemble de modules Python, de modules écrits en C, de métadonnées et de bases de données.

Un package standard comporte traditionnellement un fichier **__init__.py** dans chaque répertoire, même si ce répertoire est vide. En particulier, le répertoire racine du package contient un fichier **__init__.py** qui peut définir la version du projet et le nom des modules à importer. Il est toutefois acceptable et parfois même recommandé de conserver ce fichier **__init__.py** vide.

Par ailleurs, dans le cas d'un package exécutable, il est essentiel que celui-ci contienne un module de lancement du programme appelé **__main__.py**.

Du point de vue du développeur, il est nécessaire de rajouter des fichiers permettant la construction du package et comportant les métadonnées du package. Ainsi, les fichiers **setup.py**, **requirements.txt**, **LICENSE**, **README.md** et **MANIFEST.in** sont 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.

L'ensemble des modules nécessaires au fonctionnement du package sont regroupés dans un répertoire portant le même nom que celui du package. C'est ce répertoire qui sera archivé et distribué en mettant à jour le numéro de version dans les fichiers **__init__.py** et **setup.py**. Si au lieu de stocker les modules dans ce répertoire, on les place à la racine du package, alors la distribution du package se construit à vide, l'installation à partir de PyPI fonctionne quand même, mais l'exécution du package ne produit rien.

Par ailleurs, les packages construits pour des systèmes Linux et macOS nécessitent l'incorporation de fichiers **build.sh** et **meta.yaml** alors que les packages construits pour les systèmes Windows nécessitent l'incorporation des fichiers **bld.bat** et **meta.yaml**.

Il faut également rajouter l'archive et la distribution compilée en vue de publier le package.

Ainsi, lorsqu'un utilisateur télécharge l'archive d'un package à partir de PyPI, celui-ci présente la structure suivante

```
.
├── MANIFEST.in
├── mon_package
│   ├── __init__.py
│   ├── LICENSE
│   ├── __main__.py
│   └── requirements.txt
├── README.md
└── setup.py
```

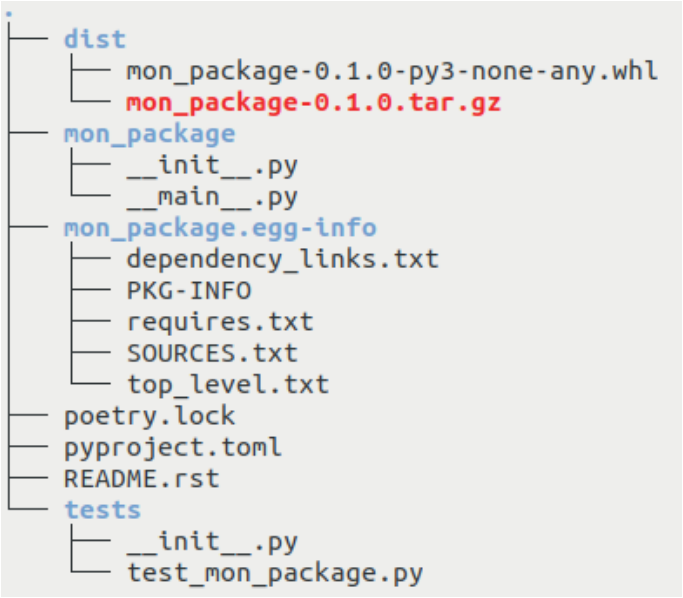
Nous ne rentrerons pas plus dans le détail de la création d'un package standard, car nous avons choisi de construire CRISPRbuilder_TB avec l'outil *Poetry*. En effet, nous avons commencé par construire manuellement notre package en incorporant les fichiers ci-dessus un par un, jusqu'à ce que les discordances rencontrées entre les contenus de ces fichiers orientent nos recherches vers un outil permettant l'automatisation du processus et la gestion des incompatibilités.

## 2. Composition d'un package sous poetry

Avec la PEP-518 [1], le PyPA [2] a proposé un nouveau standard au format *.toml* pour regrouper les métadonnées d'un package et le construire à l'aide de *Poetry*.

Du point de vue du développeur, un package, appelé *mon_package*, présente la structure suivante lorsqu'il est construit avec *Poetry* :

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




- le répertoire **dist** contient l'archive des sources du package **mon_package-0.1.0.tar.gz** (ici la version 0.1.0), construite en compressant le répertoire **mon_package**. Il contient également la distribution compilée **mon_package-0.1.0-py3-none-any.whl** permettant d'installer l'archive sur le système de l'utilisateur et de rendre le package exécutable.

- la création du répertoire **mon_package**, portant le même nom que celui du package, est automatique avec *Poetry*, de même que celle des fichiers **__init__.py**. Le répertoire **mon_package** contient tous les modules et bases de données nécessaires à l'exécution du programme ainsi que tous les éléments qui se retrouveront compressés dans l'archive **mon_package-0.1.0.tar.gz**.

- le répertoire **mon_package.egg-info** contient les ressources et les métadonnées du projet nécessaires à la création de la distribution compilée **mon_package-0.1.0-py3-none-any.whl**. Le format *.egg* facilite la désinstallation et la mise à niveau du code, car le projet est essentiellement autonome dans un unique répertoire. Il autorise aussi 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. Il est fréquent, en revanche, que *pip* ne parvienne pas à installer les fichiers au format *.egg* qui ne sont pas standardisés et ne tiennent pas compte des différents interpréteurs CPython, jython, pypy, ...

- le fichier **poetry.lock** empêche les dépendances de télécharger la dernière version au moment de leur installation, en fixant la version utilisable par le package.

- le fichier **pyproject.toml** précise les métadonnées du package telles que son nom, 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. Il sert à la construction du package et crée automatiquement le fichier **setup.py** qui, dans le cas de *Poetry*, sert à la création de **mon_package.egg-info**. Le fichier **pyproject.toml** remplace donc les fichiers **setup.py**, **setup.cfg** et**MANIFEST.in**. Il incorpore également les différents formats **build.sh**, **meta.yaml**, **bld.bat**. Il gère finalement l'installation des dépendances et remplace donc le fichier **requirements.txt**.

- le fichier **README.rst** reste au niveau de la racine du package afin de compléter la rubrique "Project description" de PyPI au moment de la publication du package. **README.rst** sera

par ailleurs automatiquement incorporé à l'archive au moment de sa création.

- le répertoire **tests** permet de valider la conformité de la distribution dont le numéro de version apparaît dans le fichier **__init__.py** qu'il contient.

Une fois que le package est publié dans l'index PyPI, sa distribution est en fait une copie du dossier **dist**, qu'on peut installer ou télécharger. Dans le premier cas, le package s'installe à l'emplacement prévu pour les packages sur le système de l'utilisateur grâce à l'outil *pip*, et il est exécutable directement à partir du répertoire courant. Dans le second cas, les deux fichiers **mon_package-0.1.0.tar.gz** et **mon_package-0.1.0-py3-none-any.whl** s'enregistrent par défaut dans le dossier **Téléchargements**.

- Lorsque l'utilisateur installe le package avec l'instruction `pip install mon_package` et qu'il se déplace dans le dossier du package à l'emplacement prévu pour les packages, il constate la structure suivante

```
.
├── __init__.py
├── LICENSE
└── __main__.py
```

  qui correspond en fait au dossier **mon_package** dans l'arborescence du package.

- Lorsque l'utilisateur télécharge l'archive du package à partir de PyPI, sa structure est la suivante

```
.
├── __init__.py
├── mon_package
│   ├── __init__.py
│   ├── LICENSE
│   └── __main__.py
├── PKG-INFO
├── pyproject.toml
├── README.rst
└── setup.py
```

  On remarque que le dossier **dist** relatif à la création de la distribution a bien évidemment disparu, ainsi que le dossier **tests** et le fichier **poetry.lock** qui a servi à fixer la version des dépendances. A partir du répertoire d'installation **mon_package.egg-info**, seul le fichier **PKG-INFO** a été conservé.

  L'archive du package ne sera, dans ce cas, pas installée. Elle sera, en revanche, présente par défaut dans le dossier **Téléchargements** de l'ordinateur. Cela signifie que l'exécution du package par l'instruction standard

```
python -m  mon_package
```

  ne fonctionne pas car le système n'est pas en mesure de trouver le package à l'emplacement prévu des packages.

  A partir du dossier **Téléchargements**, l'utilisation de l'instruction

```
python mon_package
```

  sans le "-m" ne fonctionne pas non plus, car l'interpréteur qui ne considère pas `mon_package` comme un package et ne cherche pas le fichier **__main__.py** pour l'exécuter.

  En revanche, si on se place à l'intérieur de l'archive, l'utilisation de l'instruction

```
python __main__.py
```

exécute le package en tant que programme. Cela n'a que peu d'intérêt vu qu'on attend d'un package qu'il s'exécute simplement sans qu'on ait à chercher préalablement son emplacement.

En conséquence, il est nécessaire pour l'utilisateur d'installer le package sur son ordinateur à l'emplacement réservé pour les packages afin de pouvoir l'exécuter. C'est le rôle de la distribution compilée **mon_package-0.1.0-py3-none-any.whl** qui va s'exécuter avec l'instruction

```
pip install mon_package
```

On peut dès lors entrevoir une confusion possible pour définir la racine du package. Pour l'utilisateur, la racine pourrait se trouver au niveau du module **__main__.py** si il installe le package, ou au niveau du fichier **pyproject.toml** si il télécharge le package. Dans le reste de ce document, nous considérons que la racine du package se trouve au niveau du fichier **pyproject.toml**, et que tous les chemins d'accès ont pour référence le module d'exécution primaire **__main__.py**.

Au moment de la construction du package

- le fichier **pyproject.toml** doit obligatoirement se trouver à la racine du package pour que les métadonnées du package puissent être interprétées.
- le module **__main__.py** doit obligatoirement se trouver à l'intérieur du répertoire **mon_package** pour qu'il soit effectivement installé et puisse être exécuté prioritairement.

## 3. Fonctionnement d'un package

Il est possible d'utiliser les packages de différentes manières.

- certains packages peuvent *s'importer* (partiellement ou en totalité) en début de code afin d'en appeler les méthodes dans un module.
- d'autres packages sont *exécutables* suivant une interface graphique ou suivant une interface en lignes de commandes. C'est le cas de CRISPRbuilder_TB qui s'installe et s'exécute en lignes de commandes à partir du répertoire courant. Ainsi, c'est à partir du répertoire courant qu'on appelle les modules et fichiers se trouvant à l'intérieur du package, lui-même installé à l'emplacement prévu pour les packages. Ce processus est transparent pour l'utilisateur qui n'a nul besoin de connaître l'emplacement de téléchargement de ses packages. Il est toutefois essentiel pour le développeur de connaître les chemins d'accès vers les modules d'un package, qui sont différentes des chemins d'accès vers les modules d'un programme local. On peut commencer par développer et tester le package en utilisant des chemins locaux, mais dès que celui-ci est prêt à être installé, il faut modifier ces chemins pour que n'importe quel ordinateur puisse exécuter le package convenablement.

Dans un programme local, le code s'exécute dans un répertoire courant et accède aux différents modules et fichiers support en utilisant un chemin relatif à ce répertoire courant ou encore en utilisant un chemin absolu. Les différentes dépendances sont importées en début de code, sans référence à un chemin d'accès car elles ont préalablement été installées dans l'environnement de développement de l'interpréteur Python. Dans le cas où on utilise un environnement de développement intégré tel que PyCharm, Spyder, Pydev, Wing, ..., il faut en plus les installer dans cet environnement.

Dans un package en revanche, c'est le module **__main__.py** qui est cherché par l'interpréteur et exécuté prioritairement. Prenons l'exemple concret du package CRISPRbuilder_TB. Le module

**__main__.py** se trouve dans le dossier **crisprbuilder_tb** à l'emplacement prévu pour les pakages, indépendamment du répertoire courant. Il s'agit la plupart du temps de ∼/**.local/lib/python3.8/site-packages** ou encore ∼/**.local/lib/python3.8/dist-packages**. Lorsque *Anaconda* est installé et que son environnement est activé par défaut au démarrage de la machine, cet emplacement devient ∼/**anaconda3/bin**. L'utilisateur ne trouvera aucun dossier du nom de **crisprbuilder_tb** dans le répertoire courant après installation du package, bien qu'il l'ait installé à partir du répertoire courant. On peut chercher l'emplacement exact du package et le déplacer ensuite, mais ceci n'a aucun intérêt vu que le système mémorise l'emplacement de ses packages.

Afin d'expliquer le fonctionnement interne d'un package, nous conservons l'exemple de CRISPRbuilder_TB. L'utilisateur se place dans son répertoire courant, exécute

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

qui va chercher le package CRISPRbuilder_TB à son emplacement de stockage ∼/**anaconda3/bin** et l'exécute comme si celui-ci se trouvait dans le répertoire courant. Ainsi le module **__main__.py** s'exécute et appelle les modules **fonctions.py**, **bdd.py** et les fichiers du répertoire **data**. On pourrait définir des chemins d'accès absolus vers ces modules et fichiers, mais ces chemins seraient différents pour chaque système d'exploitation. On définit plutôt des chemins relatifs au module **__main__.py**, qui sert de référence pour les chemins d'accès. Ainsi, tous les fichiers et modules sont atteignables grâce à une adresse relative commençant par une variable d'environnement qui ne contient pas le nom du package `crisprbuilder_tb`, car l'exécution du code se trouve déjà dans le package lors de la lecture du fichier **__main__.py**. Par exemple le chemin

```
P_REP = path.join(path.dirname(__file__), 'REP')
```

utilise la variable d'environnement __file__ qui définit un chemin relatif au fichier courant (le fichier **__main__.py**) suivi du dossier **REP**.

En revanche, l'importation des modules et fichiers du package en début de code doit tenir compte du nom du package. On trouve ainsi dans les premières lignes du module **__main__.py**

```
import crisprbuilder_tb
import crisprbuilder_tb.fonctions as fonctions
import crisprbuilder_tb.bdd as bdd
```

Bien évidemment, l'importation des dépendances en début de code reste indépendante du package CRISPRbuilder_TB qui ne sera donc pas cité :

```
import subprocess
from argparse import ArgumentParser
from csv import reader, writer, QUOTE_MINIMAL
from os import remove, listdir, rename, system, name, path
from pathlib import Path
from pathlib import PurePath
from shutil import rmtree, move
```

Em conclusion, il est important de comprendre que l'exécution du package :

- **ne se fait pas** depuis l'intérieur du package avec l'instruction `python __main__.py`, qui ne permet pas d'utiliser les options du package.
- **ne se fait pas** depuis le répertoire parent du package avec l'instruction `python crisprbuilder_tb --collect ERR2704808` sans le "-m". Dans ce cas, l'interpréteur

ne va pas considérer `crisprbuilder_tb` comme un package et ne va pas chercher le fichier **__main__.py** pour l'exécuter.

- **se fait** à partir du répertoire courant, en utilisant l'instruction `python -m crisprbuilder_tb --collect ERR2704808`. C'est à l'ordinateur d'aller chercher le package à son emplacement de stockage. Tous les chemins décrits dans le code sont des chemins relatifs au module **__main__.py**.

## 4. 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. Nous l'incorporons dans le fichier **LICENSE**.

# 4

# Le choix des outils

## 1. L'outil d'empaquetage

Le PyPA recommande l'utilisation de :

- *setuptools* pour définir des projets et créer des sources de distribution. Il gère les formats *.egg* et les dépendances.
- *pip* pour l'installation de distributions à partir de PyPI, en tenant compte des dépendances. Certains packages sont référencés sur l'index sans que les distributions ne soient disponibles sur PyPI, et il est également possible de télécharger les distributions à partir d'URL définies dans un fichier **setup.py**.
- *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 développement.
- *pyenv* pour la gestion des versions de Python.
- *conda* permettant de fournir un environnement de développement 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*. *Poetry* constitue en fait le nouveau standard de création de packages.

Au regard de ces recommandations, nous avons testé les outils préceédents 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 comprend une version de Python et facilite l'installation des librairies scientifiques *Numpy, Pandas, Matplotlib* et *Scipy*, ce qui est particulièrement intéressant dans le cas des plateformes Windows où ce processus est plus complexe. *Anaconda* incorpore une librairie open-source appelée *conda* permettant la

gestion des dépendances, de l'environnement de travail ainsi que la création de packages. *Anaconda* évite les problèmes d'installation de packages et de dépendances qui sont gérées de façon plus individualisée avec *pip*. *Anaconda* semble donc être appropriée au projet, mais c'est une distribution trop lourde pour être intégrée à notre package. *Miniconda* est une distribution plus légère, qui comporte essentiellement Python et *conda*, mais ne permet pas toujours d'installer toutes les dépendances d'un package comme nous le verrons ultérieurement. L'utilité d*Anaconda* est avérée dans le cadre de la création de l'environnement de développement et durant les tests d'installation, comme nous le verrons ultérieurement. Toutefois, en raison de son poids important, nous ne pouvons pas l'intégrer au package.

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*, comme cela est maintenant recommandé sur Internet. *Poetry* est un outil complet multiplateformes autour duquel la communauté Python reste très active. Il propose à la fois la gestion des dépendances, le packaging [1] et la publication. *Poetry* automatise ces différents processus et facilite grandement le travail. Nous étudions en détail l'utilisation de cet outil dans le chapitre suivant.

## 2. L'environnement de développement

Les packages s'installent par défaut à l'adresse ∼/**.local/lib/python3.8/site-packages**, et différentes versions d'un même package peuvent donc se retrouver dans le même dossier. Ce qui peut conduire à utiliser la mauvaise version d'une dépendance et entraîner le dysfonctionnement d'un projet. Pour résoudre ce problème, les environnements de développement permettent d'installer les dépendances de projets différents à des adresses différentes tenant compte de chaque environnement. Ainsi plus aucune confusion n'est possible concernant la version d'une dépendance.

Avant de créer le package CRISPSbuilder_TB, il faut commencer par construire un programme qui fonctionne localement sur une machine correctement paramétrée. Pour cela, nous définissons un environnement de développement, qui nous sert à exécuter le code et à le tester au fur et à mesure de son élaboration. Cet environnement contient tous les modules et packages nécessaires à son bon fonctionnement, c'est à dire les dépendances du projet.

L'outil de création d'environnement virtuel *venv* nous permet de créer cet 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 à l'intérieur puis nous créons, à l'aide de *venv*, un environnement de développement **env_bio** avec l'instruction suivante

```
cd biologie
python -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. Ceci se traduit par l'instruction

1. **packaging :** il s'agit de la création d'une structure pour un projet, de la génération de fichiers de configuration et de manifestes

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

De nombreux systèmes d'exploitation incorporent Python 2.7 par défaut. Il convient donc de définir une version 3 de Python dans cet environnement de développement, et dans le cadre de notre projet une version 3.7 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 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 3.8.0 par exemple se fait grâce à l'instruction

```
pyenv install -v 3.8.0
```

La version 3.8.0 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 3.8.0 pour notre environnement de travail. Il suffit de se placer dans cet environnement et d'exécuter l'instruction

```
cd env_bio
pyenv local 3.8.0
```

On peut vérifier que l'environnement a bien configuré la version 3.8.0 en exécutant de nouveau

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

ou encore avec l'instruction `python --version` qui permet de connaître quelle version de Python est utilisée dans le répertoire courant, qu'on se trouve dans un environnement de développement ou non.

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 si on utilise l'instruction

```
which python
```

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

On peut également choisir de mettre à jour la version de Python pour toute la machine, et de définir Python 3 comme étant la version par défaut. On utilise dans ce cas les instructions

```
sudo apt-get install python 3.8.0
alternatives --list | grep -i python
alternatives --install /usr/bin/python python /usr/bin/python3.8 1
alternatives --install /usr/bin/python python /usr/bin/python2.7 2
python --version
python 3.8.0
```

Cette méthode est certes plus facile, mais peut engendrer des problèmes de fonctionnement du système pour certaines applications qui utilisent la version 2 de Python ou encore une version 3 spécifique.

Finalement, nous installons dans notre environnement de développement les dépendances nécessaires grâce à l'outil *pip* qui les recherche directement dans PyPI :

- `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*. Il sera utile pour lire et transformer des fichier Excel mais ne sera pas nécessaire pour le fonctionnement du package.

- `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.

- `pip install parallel-fastq-dump`

  le package *parallel-fastq-dump* permet de téléchargement de fichiers fasta en utilisant plusieurs threads simultanément. Il procure donc un gain de temps à l'application.

- `pip install blastn+`

  le package *blast* permet de trouver des régions similaires entre plusieurs séquences biologiques. Le programme compare en fait des séquences de protéines ou de nucléotides à une base de données du NCBI afin d'en calculer la signification statistique.

Pour connaître l'emplacement où ces packages se sont installés, il est possible d'utiliser le script Python en lignes de commande

```
python
import site
site.getsitepackages()
```

qui peut par exemple renvoyer comme réponse

- **~/.local/lib/python3.8/site-packages**

- ∼/**.local/lib/python3.8/dist-packages**
- ∼/**.pyenv/versions/3.8.0/lib/python3.8/site-packages**
- ∼/**venv/lib/python3.8/site-packages**
- ∼/**anaconda3/bin**

On remarque ainsi qu'en fonction du type d'environnement de développement utilisé, les packages ne s'installent plus à l'adresse par défaut ∼/**.local/lib/python3.8/site-packages**.

On peut obtenir une information plus détaillée au sujet d'un package (par exemple *biopython*) à l'aide de l'instruction

```
pip show biopython
```

qui renvoie comme réponse

```
Name: biopython
Version: 1.77
Summary: Freely available tools for computational molecular biology.
Home-page: https://biopython.org/
Author: The Biopython Contributors
Author-email: biopython@biopython.org
License: UNKNOWN
Location: /home/stephane/.pyenv/versions/3.8.0/lib/python3.8/site-packages
Requires: numpy
Required-by: crisprbuilder-tb
```

On remarque, en plus de l'emplacement, la dépendance *numpy* ainsi que le package *crisprbuilder_ tb* dont *biopython* constitue une dépendance.

Notons que certains modules utilisés dans CRISPRbuilder_TB sont nativement présents dans la librairie standard Python 3, et il n'est donc pas nécessaire de les installer séparément. Ils sont installés en tant que *system packages* et non pas *site packages*. C'est le cas par exemple de

- 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 fournit des classes de chemins purs indépendants du système d'exploitation, ce qui permet au code de manipuler des chemins sans accéder au système d'exploitation,
- du module *argparse* qui gère la création et l'utilisation d'interfaces en ligne de commande.

Le package CRISPRbuilder_TB, une fois installé par l'utilisateur, devra fournir un environnement de travail similaire à celui que nous venons de construire, c'est à dire que les dépendances nécessaires à son bon fonctionnement devront également être installées.

On pourrait par exemple retrouver les modules installés via *pip* dans un fichier **requirements.txt** construit en gelant l'état de l'environnement à un moment précis grâce à une instruction du type

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

L'utilisateur pourrait alors installer ces modules à l'identique en utilisant l'instruction

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

Cette méthode fonctionne bien pour construire un environnement de développement à partir d'un répertoire courant où se trouve le fichier **requirements.txt**. C'est d'ailleurs la technique que nous avons utilisée pour construire notre environnement. Toutefois, elle n'est pas adaptée à l'installation des dépendances de CRISPRbuilder_TB par un utilisateur, car celui-ci devrait trouver l'emplacement du package, s'y rendre pour ensuite exécuter `pip install -r requirements.txt`.

Ce processus d'installation devrait rester transparent pour l'utilisateur. C'est pourquoi, il devrait être pris en charge par un gestionnaire de package tel que *Poetry* regroupant les versions des dépendances dans un fichier **pyproject.toml** qui remplace **requirements.txt**. Nous verrons au cours des tests de fonctionnement que les dépendances ne sont pas toujours reconnues à partir de *Poetry* et qu'il est quand même nécessaire pour l'utilisateur d'avoir préalablement installé et paramétré sur sa machine les outils du SRA Toolkit *parallel-fastq-dump* et *blast*. CRISPRbuilder_TB étant destiné à un public de bio-informaticiens, ce prérequis devrait déjà être installé sur l'ordinateur des utilisateurs du package. Il convient toutefois de le préciser dans la documentation d'installation **crisprbuilder_tb.ipynb**.

## 3. L'environnement de développement avec Anaconda

Nous venons de voir qu'il est possible de créer un environnement de développement adapté à notre projet, sans modifier le reste de la structure de notre machine. C'est d'ailleurs cette méthode que nous avons initialement utilisée. Toutefois, cet environnement nécessite un paramétrage de la variable d'environnement $PATH afin que les modules recherchés puissent être trouvés par le système au moment de l'exécution du package. Afin que ce paramétrage reste permanent, il faut modifier cette variable d'environnement à l'intérieur même du ∼/**.bashrc** en utilisant l'instruction

```
echo $PATH
export PATH=$PATH:~/.local/bin/python3.8/site-packages
```

ou selon le chemin d'accès choisi

```
echo $PATH
export PATH=$PATH:~/venv/lib/python3.8/site-packages
```

Il existe une alternative plus simple qui a retenu notre attention seulement à la fin du projet. Cela consiste à installer le package *Anaconda*, qui comprend notamment l'outil `conda` et le module *bioconda* dont nous aurons besoin. Il faut ensuite installer toutes les dépendances du projet à l'aide de `conda`. Ces dépendances s'installent systématiquement dans ∼/**anaconda3**/**bin** et la variable d'environnement $PATH se met automatiquement à jour. Le package CRISPRbuilder_TB étant destiné à un public de bio-informaticiens, il est fort probable que ces derniers aient déjà installé *Anaconda* sur leur machine, car il s'agit d'un outil classique de travail dans un environnement scientifique.

Attention toutefois de ne pas installer conjointement *Anaconda* avec *Miniconda*. En effet, dans ce cas certaines dépendances s'installent dans ∼/**anaconda3**/**bin** et d'autres dans ∼/**miniconda**/**bin**. C'est tentant au début, car *Miniconda* est une distribution légère qui donne accès à l'outil `conda` et au module *bioconda*. Toutefois, ce dernier utilise le Cloud d'*Anaconda* pour installer certaines dépendances de notre projet, ce qui rend l'installation d'*Anaconda* nécessaire. Pour éviter tout problème de chemin d'accès, il faut alors désinstaller *Miniconda* avant d'installer *Anaconda*.

L'installation de *Anaconda* est décrite sur le site `https://docs.anaconda.com/anaconda/install/linux/`. Il faut penser à ordonner les priorités des différents *channels*, dont notamment *bioconda* en ajoutant dans l'ordre

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

Il convient ensuite d'installer toutes les dépendances du projet

```
conda install -c conda-forge xmltodict
conda install -c anaconda openpyxl
conda install -c anaconda xlrd
conda install -c anaconda biopython
conda install -c bioconda parallel-fastq-dump
conda install -c bioconda blast
conda install -c kantorlab blastn
```

Notons que, contrairement au choix de créer un environnement local de développement, *Anaconda* propose un environnement de travail scientifique pour l'ensemble de la machine. Par ailleurs, il s'installe avec sa propre version de Python, qui est nécessairement récente. Lorsqu'il est paramétré pour être exécuté par défaut, les packages se placent dans son environnement à l'adresse ∼/**anaconda3**/**bin**.

# 5

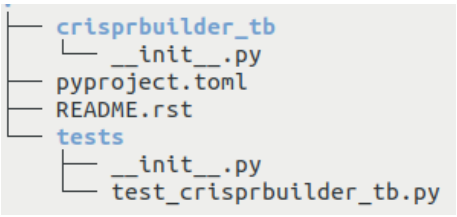
# Construction du package avec Poetry

## 1. Création du package

Pour créer notre projet, il ne faut pas commencer par définir un dossier du nom du package, car cela sera automatiquement fait par *Poetry*. Il suffit d'exécuter l'instruction

```
poetry new crisprbuilder_tb
```

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

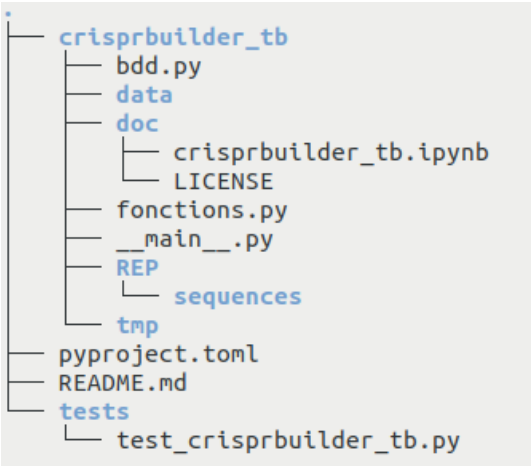



- le répertoire **crisprbuilder_tb** contient initialement le fichier **__init__.py** et c'est dans ce répertoire que nous allons rajouter tous les composants principaux du code, à savoir **__main__.py**, **fonctions.py**, **bdd.py** et les répertoires **data**, **doc**, **REP**, **sequences** et **tmp** avec leurs contenus. En effet, la création de l'archive va compresser ce répertoire **crisprbuilder_tb** afin de le publier. Nous rajoutons également dans le répertoire **doc** une **LICENSE** qui pourra être consultée par l'utilisateur. Par ailleurs, la version 3.3 de Python a introduit la notion de *Implicit Namespace Packages*, qui permet la création de packages sans la présence de fichiers **__init__.py**. Nous supprimons donc le fichier **__init__.py** du répertoire **crisprbuilder_tb**.
- le fichier **pyproject.toml** se trouve à la racine du package.
- le fichier **README.rst** fournit une brève description du package, et nous le modifions au format markdown **README.md** pour des raisons pratiques.
- nous rajoutons à la racine du package un fichier **.gitignore** pour la gestion des versions avec l'outil *Git*.
- nous supprimons le fichier **__init__.py** du répertoire **tests**.

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.

Notre projet ressemble maintenant à ceci




## 2. Gestion des dépendances

Pour chaque dépendance, *Poetry* va choisir la version la plus récente compatible avec les autres dépendances du package. Ce processus est automatisé en utilisant l'instruction

```
poetry add nom_dependance
```

après s'être placé dans le répertoire **crisprbuilder_tb**. La dépendance `nom_dependance` avec sa version est alors incorporée dans le fichier **pyproject.toml**, le fichier **poetry.lock** est créé, afin de fixer la version de cette dépendance. Il est également possible d'imposer certaines contraintes sur les versions des dépendances en les incorporant manuellement dans le fichier **pyproject.toml**.

Il faut maintenant ajouter les dépendances suivantes au projet :

```
cd crisprbuilder_tb
poetry add xlrd
poetry add xmltodict
poetry add biopython
poetry add parallel-fastq-dump
```

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

```
[tool.poetry.dependencies]
python = "^3.6"
xlrd = "^1.2.0"
xmltodict = "^0.12.0"
biopython = "^1.77"
parallel-fastq-dump = "^0.6.5"
blast = "^2.10.1"
blastn = "^2.7.1"
```

Notons que la version de Python se complète automatiquement d'après celle utilisée par défaut lors de l'appel au module *Poetry*. On peut toutefois la changer manuellement dans le fichier **pyproject.toml**, ce que nous faisons en imposant a minima la version 3.6, récente mais encore assez ré-

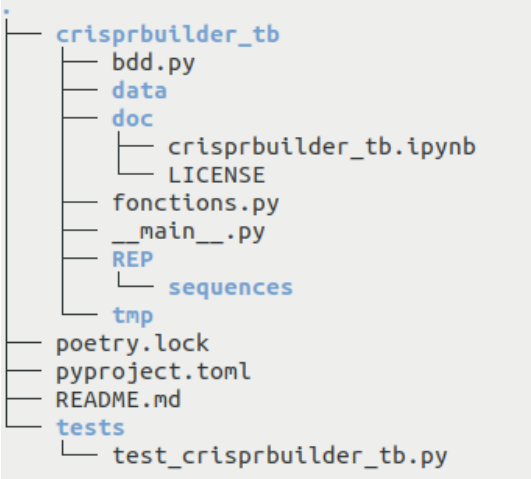
pandue. On peut également compléter manuellement ce fichier pour fournir des informations d'ordre général du type

```
[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 concernant le fonctionnement du package.

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

Notre projet dispose maintenant de la structure suivante




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

```
poetry install
```

qui crée le fichier **setup.py** à partir de **pyproject.toml**. Ce fichier **setup.py** va servir à créer automatiquement le répertoire **crisprbuilder_tb.egg-info**, comprenant le code, les ressources et les métadonnées du projet.

Lors de l'exécution du code `poetry install`, on constate également que *Poetry* crée en fait un environnement virtuel dans lequel il va installer les dépendances et le package

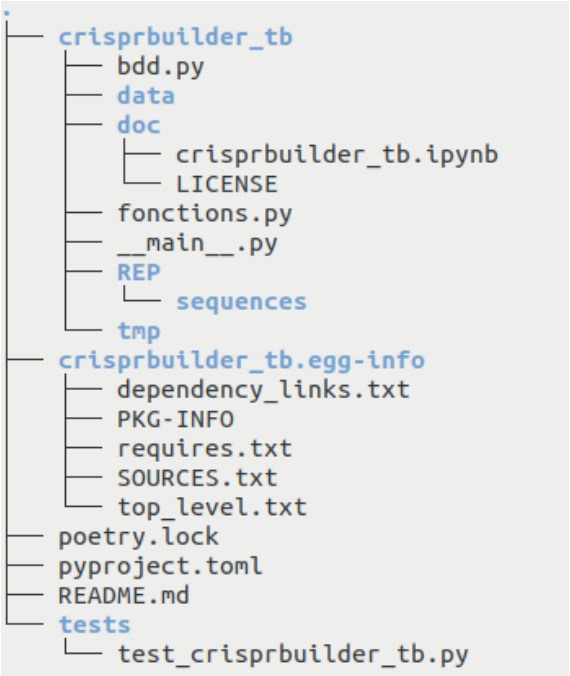
```
poetry install
Creating virtualenv crisprbuilder-tb-oJzk4q3N-py3.8 in ~/.cache/pypoetry/virtualenvs
Installing dependencies from lock file

Package operations: 14 installs, 0 updates, 0 removals
```

```
- Installing pyparsing (2.4.7)
- Installing six (1.14.0)
- Installing atomicwrites (1.4.0)
- Installing attrs (19.3.0)
- Installing more-itertools (5.0.0)
- Installing numpy (1.18.4)
- Installing packaging (20.3)
- Installing pluggy (0.13.1)
- Installing py (1.8.1)
- Installing wcwidth (0.1.9)
- Installing biopython (1.76)
- Installing pytest (4.6.9)
- Installing xlrd (1.2.0)
- Installing xmltodict (0.12.0)
- Installing parallel-fastq-dump (0.6.5)
- Installing crisprbuilder_tb (0.1.26)
- Installing blast (2.10.1)
- Installing blastn (2.7.1)
```

Notons qu'on peut s'assurer à ce moment-là de la version de Python utilisée pour tout le travail de construction, à savoir dans cet exemple Python 3.8 qui apparaît à la fin de `crisprbuilder-tb-oJzk4q3N-py3.8`.

Notre projet dispose maintenant de la structure suivante




## 3. Construction du package

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**, à partir du répertoire **crisprbuilder_tb** et du fichier d'installation **crisprbuilder_tb.egg-info**. Lors de l'exécution du code `poetry build`, on constate bien la constitution des fichiers mentionnés

```
poetry build
Building crisprbuilder_tb (0.1.2)
  - Building sdist
  - Built crisprbuilder_tb-0.1.2.tar.gz

  - Building wheel
  - Built crisprbuilder_tb-0.1.2-py3-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 Python 2. Il faut donc bien s'assurer, avant la création du package, de choisir une version de Python 3 dans le fichier **pyproject.toml**.

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.

Notre package présente maintenant la structure suivante avant d'être publié

```
.
├── crisprbuilder_tb
│   ├── bdd.py
│   ├── data
│   ├── doc
│   │   ├── crisprbuilder_tb.ipynb
│   │   └── LICENSE
│   ├── fonctions.py
│   ├── __main__.py
│   ├── REP
│   │   └── sequences
│   └── tmp
├── crisprbuilder_tb.egg-info
│   ├── dependency_links.txt
│   ├── PKG-INFO
│   ├── requires.txt
│   ├── SOURCES.txt
│   └── top_level.txt
├── dist
│   ├── crisprbuilder_tb-0.1.2-py3-none-any.whl
│   └── crisprbuilder_tb-0.1.2.tar.gz
├── poetry.lock
├── pyproject.toml
├── README.md
└── tests
    └── test_crisprbuilder_tb.py
```

## 4. Publication 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. On peut l'installer avec l'outil *pip* ou en télécharger l'archive. Le répertoire **dist** est enregistré sur PyPI à la rubrique "Download

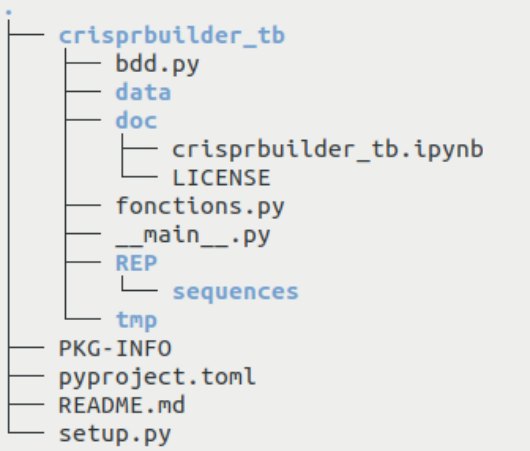
files" qui contient les fichiers **crisprbuilder_tb-0.1.2.tar.gz** et **crisprbuilder_tb-0.1.2-py3-none-any.whl**.

On peut bien évidemment publier une nouvelle version du package sur PyPI ou encore le supprimer. En revanche, par mesure de sécurité, il n'est pas possible de modifier une version du package ou de re-créer un package du même nom que celui déjà supprimé.

On pourra retrouver un cas concret simple de publication de package avec *Poetry* dans le tutoriel que nous avons publié à l'adresse suivante : http://www.youtube.com

## 5. La structure de l'archive du package

Le fichier **crisprbuilder_tb-0.1.2.tar.gz** est une archive qui, une fois décompressée contient les éléments suivants :

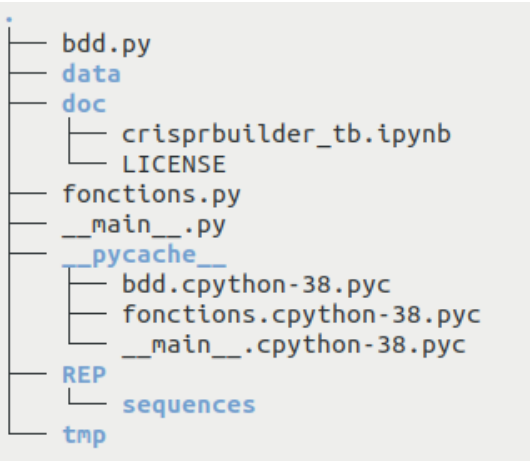



On constate une arborescence assez semblable à celle obtenue lors de la création du package. Toutefois, les dossiers **crisprbuilder_tb.egg-info**, **dist**, **tests** et le fichier **poetry.lock**, nécessaires à la construction et l'installation du package, ont bien évidemment disparu. Du répertoire **crisprbuilder_tb.egg-info** a été uniquement conservé le fichier **PKG.INFO**. Le fichier **setup.py** est également archivé.

- un répertoire **doc** comprenant la documentation nécessaire pour comprendre et utiliser convenablement le package,
- un répertoire **__pycache__** comprenant les fichiers compilés correspondant aux fichiers **__main__.py**, **fonctions.py** et **bdd.py**,
- un répertoire **data** comprenant les bases de données utiles au programme,
- un répertoire **REP** comprenant un répertoire **sequences** qui contiendra les résultats de recherche concernant les différents SRA,
- un répertoire **tmp** qui contiendra, entre autres, un fichier **nb.txt**, différent pour chaque SRA
- les fichiers **__main__.py**, **fonctions.py** et **bdd.py**, immédiatement accessibles.

## 6. La structure du package installé

Il n'est pas nécessaire pour l'utilisateur de se rendre à l'emplacement prévu pour les packages. Toutefois, afin de connaître la structure du package une fois installé, nous nous rendons à l'adresse ~/**anaconda3/bin/cripsrbuilder_tb** où la structure suivante apparaît




Nous remarquons, par rapport à l'archive **crisprbuilder_tb-0.1.2.tar.gz**, que seul le contenu du répertoire **crisprbuilder_tb** a été conservé. Ainsi, les fichiers de construction **PKG-INFO**, **pyproject.toml** et **setup.py** ont naturellement disparu. Le fichier **README.md** a également disparu. Il a servi uniquement à compléter la section *Project description* sur PyPI. On pourrait l'inclure dans le répertoire **crisprbuilder_tb** afin que l'utilisateur puisse le consulter. Toutefois, cela ne semble pas utile, vu qu'il est déjà présent en ligne sur PyPI et qu'une documentation plus exhaustive existe déjà dans l'archive du package, à savoir **crisprbuilder_tb.ipynb**.

## 7. Résumé des instructions Poetry

| **Instructions en lignes de commandes** | **Effet obtenu** |
|---|---|
| `poetry new crisprbuilder_tb` | création du package, des dossiers **crisprbuilder_tb** et **tests**, des fichiers **pyproject.toml** et **README.rst** |
| `poetry add xlrd` | rajout de la dépendance *xlrd* dans **pyproject.toml** et création du fichier **poetry.lock** |
| `poetry install` | création du fichier **setup.py** à partir de **pyproject.toml** et création du répertoire **crisprbuilder_tb.egg-info** et de ses méta-données à partir du fichier **setup.py** |
| `poetry build` | création du répertoire **dist**, des fichiers **crisprbuilder_tb-0.1.2.tar.gz** et **crisprbuilder_tb-0.1.2-py3-none-any.whl** à partir des méta-données du répertoire **crisprbuilder_tb.egg-info** |
| `poetry publish` | enregistrement en ligne des fichiers **crisprbuilder_tb-0.1.2.tar.gz** et **crisprbuilder_tb-0.1.2-py3-none-any.whl** |
| `pip install crisprbuilder_tb` | installation du package à partir du fichier **crisprbuilder_tb-0.1.2-py3-none-any.whl** |

# 6

# La documentation du package (version originale)

La création d'un package nécessite l'élaboration d'une documentation détaillée en anglais permettant de bien comprendre son fonctionnement. Une documentation trop succincte ou peu détaillée aurait pour conséquence le manque d'intérêt pour le package ou une incompréhension légitime qui démotiverait ses utilisateurs.

Cette documentation doit comporter trois volets :

- une *docstring* inclue dans le code au début de chaque fonction et de chaque module,
- un fichier **README.md** servant de description succincte du package pour le site de PyPI,
- une documentation détaillée **crisprbuilder_tb.ipynb** qu'on peut retrouver sur `https://github.com/stephane-robin/crisprbuilder_tb/blob/master/crisprbuilder_tb/doc/crisprbuilder_tb.ipynb`, et qui est également inclue dans le package sous le répertoire **doc**.

Nous présentons ci-dessous cette documentation détaillée dans sa version originale en anglais.


The CRISPRbuilder_TB package is an open source software made available under generous terms. Please see the LICENSE file for further details.

If you use CRISPRbuilder_TB in work contributing to a scientific publication, we ask that you cite the following application note : Femto-ST Institute, UMR 6174 CNRS, University of Bourgogne Franche-Comté, France.

This package originated in the work of Section [1] and Section [2].

## 1. Purpose of this package

CRISPRbuilder_TB will help you collect and annotate Mycobacterium tuberculosis whole genome sequencing data for CRISPR investigations. Given a Sequence Read Archive reference, the package will provide the genome information dictionary comprising the following elements :

- the reads - number of reads for the study, length of those reads, coverage of the study,
- the study - source of publication for the SRA reference, authors who discovered it, location of discovery, date of discovery, center that discovered the SRA, reference of the study, study accession number,
- identity - name of the SRA reference, strain for the SRA, taxid, bioproject number,
- spoligotypes - description, numbers, new version numbers, vitro description, vitro numbers, vitro new version description, vitro new version numbers, Spoligotype International Type, Spoligotype International Type silico,
- lineages - Coll et al. 2014 (for all lineages) Section [3], Stucki et al. 2016 (for Lineage 4) Section [4], Palittapongarnpim et al. 2018 (for Lineage 1) Section [5], Shitikov et al. 2017 (for Lineage 2) Section [6].

## 2. How to install the package ?

Make sure your system is set to **Python 3** in your current directory and *pip* is set to work with Python 3, before using the `pip` instruction and executing the package. The default version of Python on your computer might be different. You can also check that *pip* is linked to Python3 by writing `pip --version`, which will display a line similar to

```
pip 20.1.1 from ~/pyenv/versions/3.8.0/lib/python3.8/site-packages/pip (python 3.8)
```

When installing CRISPRbuilder_TB, the following dependencies will be automatically installed, and you will have no further action to take :

- xmltodict
- xlrd
- biopython
- numpy

Please note that CRISPRbuilder_TB requires a recent version of Python (3.6 or higher), and the installation process will seek such a version.

Prior to the installation, it is important that your system is correctly set for using NCBI tools *parallel-fastq-dump* and *blast*. The installation process will require these dependencies, but will not set the environment variable $PATH to find them on your system. You will need to do it manually by updating the *∼/.bashrc* like it is explained in https://opensource.com/article/17/6/set-path-linux. We recommand that you use *Anaconda*, which provides a simple way to update the $PATH and download these packages with the command lines

```
conda install -c bioconda parallel-fastq-dump
conda install -c bioconda blast
conda install -c kantorlab blastn
```

This package comes with a Command Line Interface, so it should be installed and executed using your command prompt. For Linux, macOS or Windows platforms, the installation requires the same instruction. From your current directory, execute the following line :

```
pip install crisprbuilder_tb
```

Please note that `cripsrbuilder_tb` is written without capital letters in the command prompt.

First the program will download the last version of CRISPRbuilder_TB, and the dependencies *xmltodict, parallel-fastq-dump, xlrd, biopython, numpy*. Then it will install the dependencies and the package.

```
Collecting crisprbuilder_tb
Collecting xmltodict<0.13.0,>=0.12.0 (from crisprbuilder_tb)
Collecting xlrd<2.0.0,>=1.2.0 (from crisprbuilder_tb)
Collecting parallel-fastq-dump<0.7.0,>=0.6.5 (from crisprbuilder_tb)
Collecting biopython <2.0,>=1.77 (from crisprbuilder_tb)
Collecting numpy (from biopython<2.0,>=1.77->crisprbuilder_tb)
Installing collected packages: xmltodict, xlrd, parallel-fastq-dump, numpy, biopython, crispr
Successfully installed biopython-1.77 crisprbuilder_tb-1.2.0 numpy-1.19.0 parallel-fastq-dump
```

It is not necessary to first install the dependencies, since CRISPRbuilder_TB will deal with them on its own.

If a previous version of CRISPRbuilder_TB already exists on your computer, you can upgrade it using

```
pip install --upgrade crisprbuilder_tb
```

If this instruction doesn't work, you can enforce the upgrade by first removing the package before installing it again

```
pip uninstall crisprbuilder_tb
pip install crisprbuilder_tb
```

You can also choose a specific version of the package on PyPI, like for example

```
pip install crisprbuilder_tb==1.2.0
```

CRISPRbuilder_TB is not referrenced in the Anaconda Cloud. So it is not possible to install the package with the instruction

```
conda install crisprbuilder_tb
```

## 3. How to use the Command Line Interface ?

After installing the package, you can find help with the command prompt, listing the different available options, such as `--collect`, `--list`, `--add`, `--print`, `--remove`, `--change`, and their expected syntax. Stay in the directory containing the installed package and write in the command prompt :

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

The help menu will display the following information :

```
usage: crisprbuilder_tb [-h] [--collect] [--list] [--add] [--remove] [--change]
[--print] sra
```

```
Collects and annotates Mycobacterium tuberculosis whole genome sequencing data
for CRISPR investigation.

positional arguments:
  sra         requires the reference of a SRA, the path to a file of SRA references
              or 0. See doc.

optional arguments:
  -h, --help  show this help message and exit
  --collect   collects the reference of a SRA to get information about this SRA.
              See doc.
  --list      collects the path to a file of SRA references to get information
              about. See doc.
  --add       collects data to add to the file data/lineage.csv. Requires 0 as
              argument. See doc.
  --remove    removes data from the file data/lineage.csv. Requires 0 as
              argument. See doc.
  --change    collects data to update the file data/lineage.csv. Requires 0 as
              argument. See doc.
  --print     prints the file data/lineage.csv. Requires 0 as argument. See doc.
```

### 3.1. Executing CRISPRbuilder_TB with a SRA reference

You can run the package if you wish to find information regarding a specific SRA. From your current directory, write in the command prompt :

```
python -m crisprbuilder_tb --collect {SRA_reference}
```

Please note the importance of "-m" to run crisprbuilder_tb as a package, otherwise the interpreter will look at crisprbuilder_tb as a folder inside the current folder and not a package. CRISPRbuilder_TB uses version 3 of Python, so make sure your system is set to Python 3 in your current directory before using the `python`instruction. The default version on your computer might be different.

**Caution :** it is not recommanded to find the location of the package and go to the directory containing CRISPRbuilder_TB in order to execute it. If you are doing so, the instruction above won't work.

**Exemple :** to find information about SRR8368696, write :

```
python -m crisprbuilder_tb --collect SRR8368696
```

### 3.2. Executing CRISPRbuilder_TB with a list of SRA references

To run the package with a list of SRA references, this list must be composed of one SRA reference per line in a txt format file. Then, if you wish to find information regarding a list of specific SRAs, stay in the directory containing the package and write in the command prompt :

```
python -m crisprbuilder_tb --list {path_to_the_file}
```

**Exemple :** to find information about the different SRAs included in the file **my_file.txt** from the **Documents** directory, write :

```
python -m crisprbuilder_tb --list /Documents/my_file.txt
```

### 3.3. Printing the database lineage.csv

If you want to consult the **lineage.csv** database, stay in the directory containing the package and write in the command prompt :

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

**Caution :** it is necessary to add the 0 after `--print`.

The program will then display the content of the database **lineage.csv** whose extract can be read below :

```
lineage, Position, Gene coord., Allele change, Codon number, Codon change, Amino acid change, Locus Id, Gen
e name, Gene type, Type of mutation, 5' gene, 3' gene, Strand, Sublineage surname, Essential, Origin,
4.1.1, 3798451, , C/G, , GGG/GGC, G/G, Rv3383c, idsB, non essential, syn, 3797437, 398489, -, X, , Stucki e
t al.,
4.1.2, 3013784, , C/G, , GTC/CTC, V/L, Rv2697c, dut, ess, non-syn, 3013683, 3014147, -, Haarlem, , Stucki e
t al.,
4.1.3, 4409231, , T/G, , GAG/GCG, E/A, Rv3921c, -, ess, 4280033, 4408969, 4410069, -, Ghana, , Stucki et a
l.,
4.2, 2181026, , G/C, , CCC/CCG, P/P, Rv1928c, , non essential, syn, 2180450, 2181217, -, -, , Stucki et a
l.,
4.3, 1480024, , G/T, , TTC/TTA, F/L, Rv1318c, , non essential, non-syn, 1479199, 1480824, -, LAM, , Stucki
et al.,
4.4, 3966059, , G/C, , ACG/AGG, T/R, Rv3529c, -, non essential, non-syn, 3965884, 3967038, -, -, , Stucki e
t al.,
```

### 3.4. Adding a record to lineage.csv

If you want to add a record to the **lineage.csv** database, stay in the directory containing the package and write in the command line :

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

**Caution :** it is necessary to add the 0 after `--add`.

You will then be asked the values of the different fields (lineage, position, gene coordinates, allege change, codon number, codon change, amino acide change, locus ID, gene name, gene type, type of mutation, 5' gene, 3' gene, strand, sublineage surname, essential, origin of the study). In case you don't know the value of a particular field, just press enter.

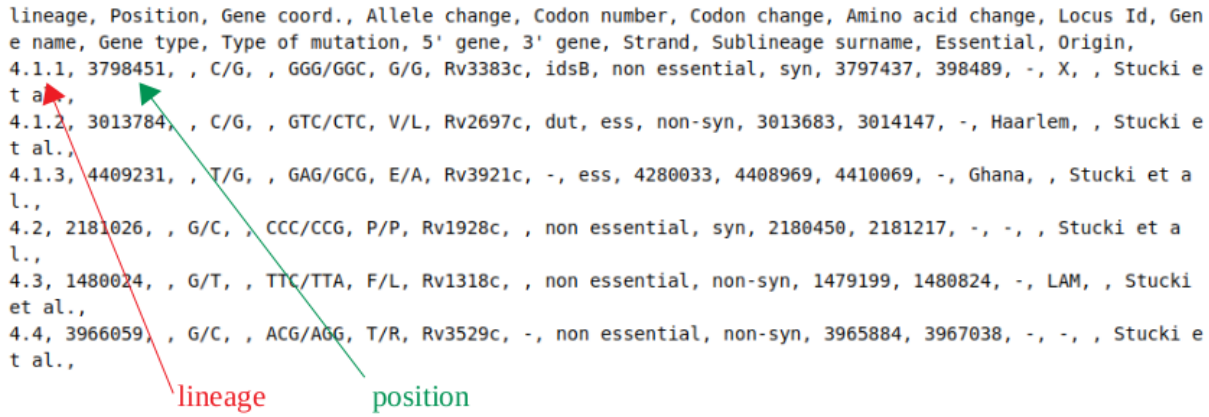
### 3.5. Removing a record from lineage.csv

If you want to remove a record from the **lineage.csv** database, stay in the directory containing the package and write in the command line :

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

**Caution :** it is necessary to add the 0 after `--remove`.

After your confirmation for removal, you will be asked for the reference of the line you wish to delete, which consists of the lineage and the position (see below). If you don't know this reference, you can always print the **lineage.csv** database beforehand to find it.




The execution of the code will end with :

```
The line has been removed.
```

or

```
Your request was cancelled.
```

in case the SRA reference is not correct.

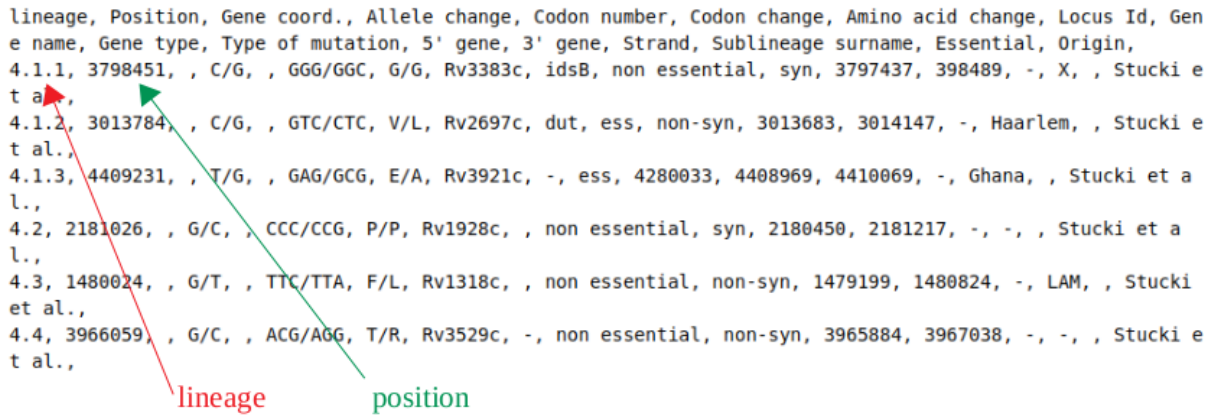
### 3.6. Changing a record from lineage.csv

If you want to change a record from the **lineage.csv** database, stay in the directory containing the package and write in the command line :

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

**Caution :** it is necessary to add the 0 after `--change`.

After confirmation for change, you will be asked for the reference of the line you wish to delete, which consists of the lineage and the position (see below). If you don't know the reference, you can always print the **lineage.csv** database beforehand to find it.




You will then be asked the values of the different fields : lineage, position, gene coordinates, allege change, codon number, codon change, amino acide change, locus ID, gene name, gene type, type of mutation, 5' gene, 3' gene, strand, sublineage surname, essential, origin of the study. In case you don't know the value of a particular field, just press enter.

The execution of the code will end with :

```
The line has been changed.
```

or

```
Your request was cancelled.
```

## 4. Composition of the package and dependencies

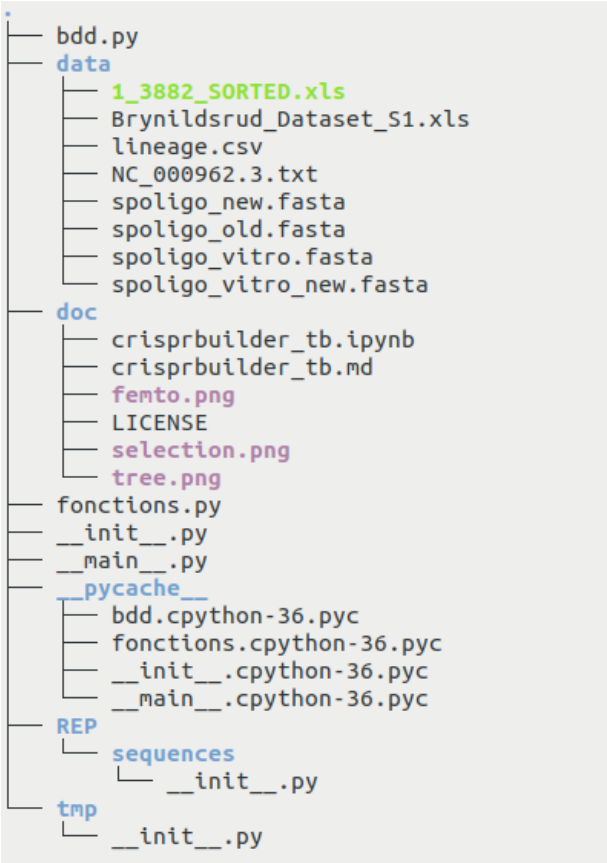
CRISPRbuilder_TB needs the following dependencies to work :

- python = "^3.6.4"
- xlrd = "^1.2.0"
- openpyxl = "^3.0.3"
- xmltodict = "^0.12.0"
- biopython = "^1.76"
- datetime = "^4.3"
- parallel-fastq-dump
- balstn+

These different versions are automatically downloaded when installing CRISPRbuilder_TB. Please note that this package doesn't support Python 2.

The package structure is the following :

- The **__main__.py** and **fonctions.py** files contain the actual code for the package,
- the **bdd.py** file contains the Origines dictionary,
- The **data** directory contains the necessary database to compare with the SRA reference when the code is executed,
- The **REP/sequences** directory contains the different result database for each execution of the code. For example, you can find a directory ERR2704808 containing the fasta files ERR2704808_1.fasta, ERR2704808_2.fasta and ERR2704808_shuffled.fasta along with the files ERR2704808.00.nhr, ERR2704808.00.nin, ERR2704808.00.nsq, ERR2704808.nal and the directory ERR2704808 containing the blast database. That's typically where you can find the files ERR2704808_old.blast and ERR2704808_new.blast.
- The **doc** directory contains the necessary documentation to explain how this package works. It includes a **README.md** file explaining how to install the package for those who just need the essential, the copyright conditions for this package with the **LICENSE** file and the present documentation.




## 5. Retrieving the genome information dictionary

Let's assume you're looking for information regarding ERR2704808. After writing `python3 -m crisprbuilder_tb --collect ERR2704808` in the command prompt, a directory called **ERR2704808** will be created in **REP/sequences** and the item ERR2704808 will be added to the database. The following message will be displayed :

```
We're creating a directory ERR2704808.
We're adding ERR2704808 to the database.
```

Then the files in fasta format will be downloaded to **REP/sequences/ERR2704808** as in the following code :

```
if len([u for u in listdir(rep) if 'fasta' in u]) == 0:
    print("We're downloading the files in fasta format")

    try:
        completed = subprocess.run(['parallel-fastq-dump', '-t', '8',
                                    '--split-files', '--fasta', '-O', P_REP,
                                    '-s', item], check=True)
        completed.check_returncode()
        # if the download worked
        print("fasta files successfully downloaded.")
        for k in listdir(P_REP):
            if k.endswith('.fasta'):
```

```
                p_item_k = str(PurePath(crisprbuilder_tb.__path__[0], 'REP',
                                        'sequences', item, k))
                p_k = str(PurePath(crisprbuilder_tb.__path__[0], 'REP', k))
                try:
                    move(p_k, p_item_k)
                except FileNotFoundError:
                    print("We can't transfer the fasta files in the proper "
                          "repository.")
    except subprocess.CalledProcessError:
        # if the download didn't work, we delete the SRA from dico_afr
        del dico_afr[item]
        print("Failed to download fasta files.")
```

If ERR2704808_shuffled.fasta is not in the **ERR2704808** directory, then the ERR2704808_1.fasta and ERR2704808_2.fasta files will be mixed in a new ERR2704808_shuffled.fasta file in the **ERR2704808** directory, as in the following code :

```
p_shuffled = str(PurePath(crisprbuilder_tb.__path__[0], 'REP', 'sequences',
                          item, item + '_shuffled.fasta'))
    if item + '_shuffled.fasta' not in listdir(rep):

        print("We're mixing both fasta files, which correspond to the two "
              "splits ends.")

        p_fasta_1 = str(PurePath(crisprbuilder_tb.__path__[0], 'REP',
                                 'sequences', item, item + '_1.fasta'))
        p_fasta_2 = str(PurePath(crisprbuilder_tb.__path__[0], 'REP',
                                 'sequences', item, item + '_2.fasta'))

        if name == 'posix':
            system("sed -i 's/" + item + './' + item + "_1./g' " + p_fasta_1)
            system("sed -i 's/" + item + './' + item + "_2./g' " + p_fasta_2)
            system("cat " + p_fasta_1 + " " + p_fasta_2 + " > " + p_shuffled)
        else:
            fonctions.change_elt_file(p_fasta_1, '_1', item)
            fonctions.change_elt_file(p_fasta_2, '_2', item)
            fonctions.concat(p_fasta_1, p_fasta_2, p_shuffled)
```

The number of reads represented by '>' in ERR2704808_shuffled.fasta will be counted and assigned to nb_reads, as in the following code :

```
if 'nb_reads' not in dico_afr[item] or dico_afr[item]['nb_reads'] == '':
    if name == 'posix':
        system("cat " + p_shuffled + " | grep '>' | wc -l > " + P_TXT_POSIX)
        nb_reads = eval(open(P_TXT_POSIX).read().split('\n')[0])
    else:
        with open(p_shuffled, 'r') as f_in, open(P_TXT_WIN, 'w') as f_out:
            lignes = f_in.readlines()
            cpt = 0
            for elt in lignes:
                cpt += elt.count('>')
                f_out.write(str(cpt))
```

```
        nb_reads = eval(open(P_TXT_WIN).read().split('\n')[0])

    dico_afr[item]['nb_reads'] = nb_reads
```

The length of the reads will be evaluated from ERR2704808_shuffled.fasta as in the following code :

```
if 'len_reads' not in dico_afr[item]:
    nb_len = len(''.join(open(p_shuffled).read(10000).split('>')[1].split(
                '\n')[1:]))
    dico_afr[item]['len_reads'] = nb_len
```

which eventually will be displayed by :

```
nb_reads: 14304698
len_reads: 108
```

When starting the download, the following information will be displayed :

```
SRR ids: ['ERR2704808']
extra args: ['--split-files', '--fasta']
tempdir: /tmp/pfd_1a9adanv
ERR2704808 spots: 7152349
blocks: [[1, 894043], [894044, 1788086], [1788087, 2682129], [2682130, 3576172], [3576173, 44
```

If the NCBI server is busy or your connection unstable, the following message could show up :

```
2020-05-09T09:25:58 fastq-dump.2.8.2 sys: libs/kns/unix/syssock.c:606:KSocketTimedRead: timed
```

Ignore the message and let the program run. It might take a while, but you'll eventually find that the download is processing correctly with the following message :

```
Read 894043 spots for ERR2704808
Written 894043 spots for ERR2704808
fasta files successfully downloaded.
We're mixing both fasta files, which correspond to the two splits ends.
```

The coverage will be calculated and, if it is too, low the procedure will be cancelled :

```
if 'couverture' not in dico_afr[item] or \
        dico_afr[item].get('couverture') == '':
    dico_afr[item]['couverture'] = round(dico_afr[item].get('nb_reads') *
                                         dico_afr[item].get('len_reads') /
                                         TAILLE_GEN, 2)


if dico_afr[item].get('couverture') < 50:
    del dico_afr[item]
    print(f"The coverage is too low. {item} is being removed from the "
          "database")
```

which will be displayed by :

```
couverture: 350.2
```

Then, a database for blast will be created in **REP/sequences/ERR2704808/ERR2704808**, as in the code :

```
if item+'.nal' not in listdir(rep) and item+'.nin' not in listdir(rep):
    print("We're creating a database for Blast")
```

```
    try:
        completed = subprocess.run(['makeblastdb', '-in', p_shuffled,
                                    '-dbtype', 'nucl', '-title', item,
                                    '-out', repitem], check=True)
        completed.check_returncode()
    except subprocess.CalledProcessError:
        print("We can't proceed blasting file.")
```

You will then read the following message :

```
We're creating a database for Blast
Building a new DB, current time: 05/09/2020 12:18:30
New DB name:   /home/stephane/Biologie/env_bio/CRISPRbuilder-TB/REP/sequences/  ERR2704808/ER
New DB title:  ERR2704808
Sequence type: Nucleotide
Keep MBits: T
Maximum file size: 1000000000B
Adding sequences from FASTA; added 14304698 sequences in 1213.96 seconds.
```

Some research will eventually be made in the dataset Brynildsrud, according to the code :

```
brynildsrud = fonctions.to_brynildsrud()
if item in brynildsrud:
    for elt in brynildsrud[item]:
        dico_afr[item][elt] = brynildsrud[item][elt]
        print(f"{item} is in the database Brynildsrud")
else:
    print(f"{item} is not in the database Brynildsrud")
```

If ERR2704808 doesn't belong to the dataset Brynildsrud, the following message will appear :

```
ERR2704808 is not in the database Brynildsrud
```

Sequences from the fasta files will be added and the spoligotypes will be blasted, as in the code :

```
if 'spoligo' not in dico_afr[item] or dico_afr[item]['spoligo'] == '':
    print("The spoligotypes are being blasted")
    dico_afr[item]['spoligo'] = ''
    dico_afr[item]['spoligo_new'] = ''

    p_spoligo_old = str(PurePath(crisprbuilder_tb.__path__[0], 'data',
                                 'spoligo_old.fasta'))
    p_spoligo_new = str(PurePath(crisprbuilder_tb.__path__[0], 'data',
                                 'spoligo_new.fasta'))
    p_old_blast = str(PurePath(crisprbuilder_tb.__path__[0], 'tmp', item
                               + "_old.blast"))
    p_new_blast = str(PurePath(crisprbuilder_tb.__path__[0], 'tmp', item
                               + "_new.blast"))

    try:
        completed = subprocess.run("blastn -num_threads 12 -query " +
                                   p_spoligo_old + " -evalue 1e-6 -task "
                                   "blastn -db " + repitem + " -outfmt "
                                   "'10 qseqid sseqid sstart send qlen "
```

```
                                "length score evalue' -out " +
                                p_old_blast, shell=True, check=True)
        completed.check_returncode()
    except subprocess.CalledProcessError:
        print("We can't proceed blasting file.")

    try:
        completed = subprocess.run("blastn -num_threads 12 -query " +
                                p_spoligo_new + " -evalue 1e-6 -task "
                                "blastn -db " + repitem + " -outfmt "
                                "'10 qseqid sseqid sstart send qlen "
                                "length score evalue' -out " +
                                p_new_blast, shell=True, check=True)
        completed.check_returncode()
    except subprocess.CalledProcessError:
        print("We can't proceed blasting file.")

    for pos, spol in enumerate(['old', 'new']):
        p_blast = str(PurePath(crisprbuilder_tb.__path__[0], 'tmp', item
                               + '_' + spol + '.blast'))
        p_fasta = str(PurePath(crisprbuilder_tb.__path__[0], 'data',
                               'spoligo_' + spol + '.fasta'))

        with open(p_blast) as file:
            matches = file.read()
            nb_max = open(p_fasta).read().count('>')
            for k in range(1, nb_max + 1):
                if matches.count('espaceur' + spol.capitalize() + str(k)
                                 + ',') >= 5:
                    dico_afr[item]['spoligo' + ['', '_new'][pos]] \
                                 += '\u25A0'
                else:
                    dico_afr[item]['spoligo' + ['', '_new'][pos]] \
                                 += '\u25A1'

        dico_afr[item]['spoligo' + ['', '_new'][pos] + '_nb'] = [
            matches.count('espaceur' + spol.capitalize() + str(k) + ',')
            for k in range(1, nb_max + 1)]
        try:
            move(p_blast, rep)
        except FileNotFoundError:
            print(p_blast, " is already in the SRA directory.")
```

You will read the message :

```
The spoligotypes are being blasted
The spoligo-vitro are being blasted
```

The different files ERR2704808_*.blast will then be moved to **REP/sequences/ERR2704808**

As a result, the programm will display the following information :

```
spoligo: ■□■■■■□□□□■■■■■■■■□□□■■■□□■■■■■■■□□■■■■

spoligo_nb: [294, 0, 0, 283, 316, 284, 298, 0, 0, 0, 0, 0, 290, 331, 304, 324, 279, 259, 284, 357, 0, 0, 0,
0, 500, 296, 315, 0, 0, 351, 273, 312, 341, 299, 309, 339, 0, 0, 0, 230, 254, 277, 294]

spoligo_new: ■■□□■■■■■■■■■■■□□□□□■■■■■■■□□□■■■□□■■■■■■■■■■□□□□□□□□□■■■■■■■□□□□□□□□□□□□□□□□□□□□□□□□
□□□□

spoligo_new_nb: [295, 294, 0, 0, 258, 309, 359, 321, 310, 258, 300, 283, 316, 284, 298, 310, 0, 0, 0, 0, 0,
0, 290, 331, 304, 324, 279, 259, 284, 357, 0, 0, 0, 0, 500, 296, 315, 0, 0, 351, 273, 312, 341, 299, 293, 3
09, 339, 309, 297, 306, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 299, 230, 254, 277, 294, 275, 307, 323, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The same process is applied to the spoligotypes_vitro.

```
spoligo_vitro: ■□□■■■■□□□□■■■■■■■■□□□■■■□□■■■■■■■□□■■■■

spoligo_vitro_nb: [(327, 314), (0, 0), (0, 0), (303, 302), (360, 353), (332, 310), (342, 334), (0, 0), (0,
0), (0, 0), (0, 0), (0, 0), (346, 334), (357, 359), (345, 336), (345, 354), (310, 301), (291, 287), (297, 3
27), (391, 382), (0, 0), (0, 0), (0, 0), (0, 0), (500, 500), (332, 321), (346, 362), (0, 0), (0, 0), (377,
371), (312, 297), (338, 356), (401, 364), (343, 341), (329, 339), (368, 363), (0, 0), (0, 0), (0, 0), (269,
270), (289, 270), (331, 305), (333, 327)]

spoligo_vitro_new: ■■□□■■■■■■■■■■■□□□□□■■□■■■■■□□□■■■□□■■■■■■■■■■□□□□□□□□□■■■■■■■□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□

spoligo_vitro_new_nb: [(347, 325), (327, 314), (0, 0), (0, 0), (298, 280), (338, 353), (396, 396), (375, 35
5), (351, 335), (288, 285), (329, 329), (303, 302), (360, 353), (332, 310), (342, 334), (344, 329), (0, 0),
(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (346, 334), (357, 359), (0, 301), (345, 354), (310, 301), (291, 28
7), (297, 327), (391, 382), (0, 0), (0, 0), (0, 0), (0, 0), (500, 500), (332, 321), (346, 362), (0, 0), (0,
0), (377, 371), (312, 297), (338, 356), (401, 364), (343, 341), (341, 326), (329, 339), (368, 363), (341, 3
37), (338, 337), (362, 338), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0,
0), (336, 338), (269, 270), (289, 270), (331, 305), (332, 327), (313, 293), (322, 335), (347, 371), (0, 0),
(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0,
0), (0, 0), (300, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0,
0), (0, 0), (0, 0)]
```

The Spoligotype International Type and Spoligotype International Type silico will be retrieved according to the following code :

```
if 'SIT' not in dico_afr[item] or dico_afr[item]['SIT'] == '':
    fonctions.add_spoligo_dico('SIT', dico_afr, item, spol_sit)

if 'SIT_silico' not in dico_afr[item]:
    fonctions.add_spoligo_dico('SIT_silico', dico_afr, item, spol_sit)
```

which will display the following message :

```
We're adding the SIT: X to the database
We're adding the SIT_silico: X to the database
```

Finally, the lineage will be defined by comparing the blast files with the different lineages L6+animal, Principal Genotyping Group, Principal Genotyping Group-cp, Coll, Pali, Shitikov and Stucki, as for example in the code :

```
if 'lineage_Pali' not in dico_afr[item]:
    lignee = []
    lignee_snp = fonctions.to_reads('Pali')
    print("We're adding the lineage according to the SNPs Pali")

    for item2, pos0 in enumerate(lignee_snp):
```

```
            seq1, seq2 = lignee_snp[pos0][:2]
            p_blast = str(PurePath(crisprbuilder_tb.__path__[0], 'tmp',
                                   'snp_Pali.blast'))
            with open(fonctions.P_FASTA, 'w') as f_fasta:
                f_fasta.write('>\n' + seq2)
            cmd = "blastn -query " + fonctions.P_FASTA + " -num_threads 12" \
                  " -evalue 1e-5 -task blastn -db " + repitem + \
                  " -outfmt '10 sseq' -out " + p_blast
            system(cmd)
            with open(p_blast) as f_blast:
                formatted_results = f_blast.read().splitlines()

            nb_seq1 = fonctions.to_nb_seq(seq1, formatted_results, 16, 20,
                                          21, 25)
            nb_seq2 = fonctions.to_nb_seq(seq2, formatted_results, 16, 20,
                                          21, 25)

            if nb_seq2 > nb_seq1:
                lignee.append(lignee_snp[pos0][2])

        lignee = [u for u in sorted(set(lignee))]

        dico_afr[item]['lineage_Pali'] = lignee
```

During its work the program will display the following message :

```
We're adding the lineage according to the SNPs L6+animal
We're adding the lineage according to the SNPs PGG
The lineage is being updated.
We're adding the lineage according to the SNPs Coll
We have selected specific reads to compare with different lineages
We're adding the lineage according to the SNPs Pali
We have selected specific reads to compare with different lineages
We're adding the lineage according to the SNPs Shitikov
We have selected specific reads to compare with different lineages
We're adding the lineage according to the SNPs Stucki
```

which will produce the following result :

```
lineage_L6+animal: 1
lineage_PGG_cp: ['1', 'X']
lineage_PGG: X
lineage_Coll: ['4', '4.9', '5']
lineage_Pali: ['1']
lineage_Shitikov: []
Lignee_Stucki: ['4.10']
```

Some general information will be retrieved from the dataset Origines or directly from NCBI, according to the code :

```
if 'Source' not in dico_afr[item]:
    for ref in bdd.Origines:
        if item in ref['run accessions']:
```

```
            for elt in ['Source', 'Author', 'study accession number',
                        'location']:
                dico_afr[item][elt] = ref.get(elt)

if 'taxid' not in dico_afr[item]:
    dicobis = fonctions.get_info(item)
    for elt in dicobis:
        dico_afr[item][elt] = dicobis[elt]
```

If the SRA belongs to Origines, the following message will be displayed :

```
ERR2704808 is in the database Origines
```

which will produce the following result :

```
Source: Unexpected Genomic and Phenotypic Diversity of Mycobacterium africanum   Lineage 5 Af
Author: Ates et al. 2018
study accession number: PRJEB25506
location: France
date: 2007
SRA: ERS2280688
center: DST/NRF Centre of Excellence for Biomedical TB research, SAMRC Centre for TB   Resear
strain:
taxid: 33894
name: Mycobacterium tuberculosis variant africanum
study: ena-STUDY-DST/NRF Centre of Excellence for Biomedical TB research, SAMRC Centre for TB
bioproject: PRJEB25506
```

# 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.

Plusieurs niveaux de tests doivent être effectués pour valider le package avant publication. Ils nécessitent un environnement de développement dont nous avons déjà parlé et un environnement de test. Celui-ci est re-créé pour chaque système d'exploitation faisant partie de la campagne de tests. Il doit simuler le cas le plus défavorable, c'est à dire qu'il doit être représentatif d'un système d'exploitation "vierge" ne disposant pas des dépendances nécessaires à l'exécution du package. Il doit toutefois fonctionner avec Python 3 par défaut ou appeler le package à l'aide de Python 3.

Notre campagne de tests comporte les démarches suivantes :

- durant la création de chaque fonction, les tests unitaires sont effectués à l'aide de l'environnement de développement,
- au moment du regroupement du code et des différents modules, les tests d'intégration utilisent l'environnement de développement
- lors d'une modification de code, les tests de non-régression utilisent l'environnement de développement,
- lorsque le package est construit, les tests de performance et les tests fonctionnels s'appuient également sur l'environnement de développement
- en dernier lieu, les tests d'installation en fonction de différents systèmes d'exploitation nécessitent un environnement de test,
- finalement, les tests de configuration pour déterminer les systèmes les mieux adaptés utilisent également un environnement de test.

## 1. Les tests unitaires, le module unittest

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.

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.

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

TODO - EXPLICATIONS unittest

## 2. LES TESTS D'INTÉGRATION

Les tests d'intégration permettent de valider l'inter-utilisabilité des différentes unités de programmes et des différents modules entre eux. Nous réalisons les tests d'intégration directement dans l'environnement de développement que nous avons construit.

Pour cela, nous nous plaçons à la racine du package et nous exécutons directement le code du module **__main__.py** par la commande

```
cd crisprbuilder_tb
python __main__.py
```

### 2.1. Les problèmes rencontrés

Une fois que le code a été testé individuellement avec comme valeur pour le SRA ERR2704808, qu'il a été rendu compatible pour chaque fonction, chaque variable et chaque paramètre, qu'il a passé avec succès l'analyse de pylint et les tests statiques de l'EPI PyCharm, nous avons rencontré les problèmes suivants :

- subprocess.run : au moment de l'exécution du code, une erreur de type `XX` apparaît. Nous nous rendons compte que l'installation manuelle de *subprocess.run*, loin de résoudre le problème, empêche la bonne exécution du code. Il est donc recommandé de ne pas l'installer et de se contenter de la version native présente dans Python.
- parallel-fastq-dump :
- la gestion des exceptions : la gestion des exceptions avec des blocs **try ... except FileNotFoundError** ne permet pas au programme de déplacer un fichier vers un répertoire contenant déjà ce fichier. Ce problème est plus sensible au moment des tests car nous utilisons à plusieurs reprises le même SRA. Toutefois, il pourrait également advenir à un utilisateur qui aurait inscrit plusieurs fois sur une liste le même SRA. Il convient donc de tester la présence d'un fichier dans un répertoire avant de le déplacer.

La première erreur qui apparaît est l'impossibilité pour l'interpréteur Python de trouver les modules **fonctions.py** et **bdd.py**. Dans le module **__main__.py**, il faut donc importer le package *crisprbuilder_tb*, puis importer les modules **crisprbuilder_tb.fonctions.py** et **crisprbuilder_tb.bdd.py** sans oublier à chaque fois le nom du package, pour que l'interpréteur parvienne à trouver ces modules. Ceci est un peu surprenant car ces modules se situent également à la racine du package, et devraient donc être reconnus automatiquement.

## 2.2. Le respect des règles du PEP 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 pour 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 comprend mal l'utilisation des compréhensions de listes et des méthodes telles que .setdefault pour récupérer la valeur d'une variable dans un dictionnaire.

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

```
cd crisprbuilder_tb
pylint __main__.py
```

Comme *pylint* considère que les modules ne devraient pas excéder 1000 lignes, nous avons développé notre module initial __main__.py en trois modules __main__.py, fonctions.py et bdd.py. Nous nous sommes par ailleurs astreints aux règles du PEP pour obtenir une note de 9.72 / 10 avec les commentaires suivants concernant le module __main__.py.

```
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
```

De la même façon, le module fonctions.py a obtenu la note de 9.80 / 10 avec les commentaires suivants

```
pylint fonctions.py
************* Module fonctions
fonctions.py:372:13: W1510: Using subprocess.run without explicitly set ‘check‘ is not recomm
fonctions.py:380:0: R0913: Too many arguments (6/5) (too-many-arguments)
fonctions.py:400:0: R0913: Too many arguments (7/5) (too-many-arguments)

-----------------------------------
Your code has been rated at 9.80/10
```

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

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

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.

TODO PROBLEME PASSAGE PACKAGE CHEMINS D'ACCES

## 3. Les tests de non-régression

Les tests de non-régression permettent 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 --print 0
```

et

```
python -m crisprbuilder_tb --collect ERR2704812
```

pour procéder aux vérifications aprés modification de code.

### 3.1. Le versioning avec Git

L'utilisation de l'outil de versioning *Git* a constitué une aide précieuse au cours de chaque modification significative du code, permettant de revenir en arrière en cas de besoin et de ne pas perdre le travail effectué. Ce stage a été l'occasion de manipuler régulièrement les principales commandes *Git*.

Notons à ce propos que *Poetry* construit un dossier **crisprbuilder_tb** au moment de la création du package avec l'instruction

```
poetry new crisprbuilder_tb
```

et que *Git* construit également un autre dossier **crisprbuilder_tb** au moment de la création d'un répertoire *Git* avec l'instruction

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

Il est donc nécessaire de créer d'abord le package avec *Poetry* puis d'initaliser ce répertoire en liaison avec un dépôt *Git*, en se plaçant à la racine de ce répertoire eten utilisant l'instruction

```
git init
git remote https://github.com/stephane-robin/crisprbuilder_tb.git
git add .
git commit -m "first commit"
git push origin master
```

## 4. Les tests d'installation

Les tests d'installation permettent de s'assurer que l'utilisateur sera en mesure d'installer le package en dehors de son environnement de développement, qu'il dispose d'un système d'exploitation Linux,

macOS ou Windows. Pour simuler le fonctionnement de ces systèmes, nous choisissons de travailler sur les quatre machines virtuelles suivantes créées par *VirtualBox*, toutes fonctionnant en 64 bits :

- Ubuntu 18.04 LTS,
- Ubuntu 20.04 LTS,
- macOS 10.12 Sierra,
- Windows 10.

Afin de tester le comportement du package à l'installation, il est possible de le publier officieusement sur la plateforme testPyPI. Il faut alors se placer à la racine du package et lancer en ligne de commande l'instruction

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

On peut donc télécharger un package à partir d'index différents de PyPI, comme par exemple test-PyPI. Toutefois, les dépendances ne sont pas nécessairement répertoriées dans cet index. Cela induit dans ce cas une erreur au téléchargement et interdit l'installation du package. Il faut donc faire référence à PyPI pour le téléchargement des dépendances. Par exemple, l'installation de CRISPR-builder_TB à partir de testPyPI se fait à l'aide de l'instruction suivante

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url
https://pypi.org/simple/ crisprbuilder_tb
```

Attention à bien respecter l'espace entre `simple/` et `crisprbuilder_tb`.

L'instruction `--index -url` permet l'installation du package à partir de textPyPI alors que l'instruction `--extra -index -url` permet l'installation des dépendances à partir de PyPI.

Il peut arriver que testPyPI n'installe pas la dernière version. Dans ce cas, il convient de préciser la version souhaitée du package, par exemple pour la version 0.1.2, il faut écrire

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url
https://pypi.org/simple/ crisprbuilder_tb==0.1.2
```

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

Il est fréquent qu'il s'affiche à ce moment-là le message d'erreur suivant

```
  ERROR: Could not find a version that satisfies the requirement crisprbuilder_tb==0.1.2 (from
ERROR: No matching distribution found for crisprbuilder_tb==0.1.2
```

Dans ce cas, la répétition de l'instruction précédente résout le problème, sans raison apparente.

Une fois que l'installation est validée à partir de testPyPI et que les modifications nécessaires ont été effectuées dans le code du package, on peut publier le package sur PyPI et procéder aux tests sur la version proposée aux utilisateurs avec la commande

```
pip install crisprbuilder_tb
```

Lorsqu'une version antérieure de CRISPRbuilder_TB existe déjà sur l'ordinateur, il peut arriver que l'installation d'une version plus récente échoue. Il est possible de la mettre à niveau de la façon suivante

```
pip install --upgrade crisprbuilder_tb
```

En cas de difficulté rencontrée, on peut forcer la mise à niveau en supprimant d'abord le package *crisprbuilder_tb*

```
pip uninstall crisprbuilder_tb
pip install crisprbuilder_tb
```

### *4.0.1. L'utilisation de VirtualBox*

Afin d'effectuer les tests d'installation et de validation, nous créons artificiellement des plateformes de test vierges grâce à *VirtualBox*. Nous configurons systématiquement Python dans sa dernière version qui est la 3.8 et nous installons *Anaconda*, ce qui correspond sur Linux aux instructions suivantes

```
sudo apt-get install python 3.8.0
/usr/ A VERIFIER
apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2 libxrandr2 libxss1 libxcursor1 libxcc
bash ~/Téléchargements/Anaconda3/-2020.02-Linux-x86_64.sh
```

Nous commençons par télécharger des images *.iso* des différents systèmes d'exploitation, que nous clonons avant de commencer les tests, afin de conserver une version vierge pour pouvoir revenir en arrière en cas de besoin :

- pour Ubuntu 18.04 LTS, nous choisissons la version 64-bit Server Install Image car nous n'avons pas besoin d'interface graphique pour réaliser nos tests. La version de Python par défaut de ce système est 2.7.17. Il existe également une version 3.6.9 de Python.
- pour Ubuntu 20.04 LTS, nous choisissons la version 64-bit Server Install Image. La version de Python par défaut de ce système est 3.8.2.
- pour macOS 10.12 Sierra,
- pour Windows 10,

Nous commençons les tests en rajoutant dans la description de chaque système toutes les installations effectuées afin que le package fonctionne correctement.

Les premiers tests d'installation avec Ubuntu 18.04 LTS s'avèrent défaillants car nous n'avons pas précisé l'adresse de PyPI pour l'installation des dépendances, ce qui a entraîné des difficultés d'installation automatique des modules au regard des contraintes imposées par le fichier **poetry.lock**. Pour s'assurer que les dépendances soient bien installées, nous avons dans un premier temps proposé que l'utilisateur le fasse manuellement en suivant les consignes données dans la documentation et à l'aide de l'instruction `pip install nom_module`. Pour simplifier le travail de l'utilisateur, nous avons pensé inclure un fichier **requirements.txt** dans le package et demander à l'utilisateur d'exécuter `XXX`. Toutefois, cela suppose que l'utilisateur soit en mesure de trouver l'emplacement de son package dans son système avant d'utiliser le fichier **requirements.txt**, ce qui n'est pas nécessairement évident sur MacOS par exemple. Cela ne nous semble pas être conforme à l'esprit d'un package dont les dépendances doivent s'installer automatiquement grâce qu fichier **poetry.lock**. De ce fait, nous avons abandonné cette idée, et après avoir compris que les dépendances pouvaient être téléchargées sur PyPI, nous avons correctement effectué l'installation de CRISPRbuilder_TB. Malheureusement l'exécution de l'instruction

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

conduit au message d'erreur suivant

`XXX`

ce qui montre qu'un système vierge ne parvienne pas à exécuter *parallel-fastq-dump*, même si ce module a été préalablement installé par *Poetry*, probablement car il n'en connaît pas le chemin d'accès.

Pour que l'installation fonctionne correctement, il faut donc s'assurer que l'utilisateur dispose d'un version fonctionnelle de *parallel-fastq-dump* avant l'installation de CRISPRbuilder_TB, en la testant avec la commande

```
parallel-fastq-dump --sra-id SRR1219899 --threads 4 --outdir out/ --split-files --gzip
```

Notons que dans le cas où une version de *parallel-fastq-dump* existe déjà dans le système, *Poetry* ne cherche pas à en installer une nouvelle version `requirements already satisfied`.

On retient qu' il n'est donc pas nécessaire pour l'utilisateur d'installer manuellement les dépendances du projet, à l'exception de *parallel-fastq-dump*.

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**.

CHOIX DU NOM SUR PYPI A USAGE UNIQUE

L'installation du package par n'importe quel utilisateur se fait grâce à l'instruction

```
pip install crisprbuilder_tb
```

après s'être assuré que *pip* est bien configuré pour s'exécuter avec Python 3.

A RAJOUTER DANS LE NOTEBOOK Dans le cas contraire, il convient de préciser

```
pip3 install crisprbuilder_tb
```

A RAJOUTER DANS LE NOTEBOOK AVEC LES SPOLIGO ATTENTION A L'ESPACE AVANT CRISPRBUILDER LORS DE L'INSTALLATION A RAJOUTER AU NOTEBOOK

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 défini automatiquement par le système 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

```
['',
'\raisebox{-1ex}{\textasciitilde}/.pyenv/versions/3.6.5/lib/python36.zip',
'\raisebox{-1ex}{\textasciitilde}/.pyenv/versions/3.6.5/lib/python3.6',
'\raisebox{-1ex}{\textasciitilde}/.pyenv/versions/3.6.5/lib/python3.6/lib-dynload',
'\raisebox{-1ex}{\textasciitilde}/.local/lib/python3.6/site-packages',
'\raisebox{-1ex}{\textasciitilde}/.pyenv/versions/3.6.5/lib/python3.6/site-packages']
```

On pourrait donc utiliser `sys.path[3]` comme chemin permettant d'accéder au répertoire parent de **crisprbuilder_tb**. Toutefois, en fonction du système d'exploitation, le 4ème élément de cette liste pour

Lorsque l'installation est réussie

## 4.1. L'utilité de Anaconda

Une fois l'installation réussie, le premier test de fonctionnement avec l'instruction

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

fournit le bon résultat.

En revanche, l'instruction

```
python -m crisprbuilder_tb --collect ERR2704812
```

produit une erreur dont voici le résumé

```
File "~/.local/bin/parallel-fastq-dump" line 112, in <module> main()
p = subprocess.Popen(["sra-stat", "--meta", "--quick", sra_id], stdout=subprocess.PIPE)
File "/usr/lib/python3.8/subprocess.py", line 1364, in _execute_child raise child_exception_t
FileNotFoundError: [Errno2] No such file or directory: 'sra-stat': 'sra-stat'
```

Il semble au début qu'il faille installer le *SRA Toolkit*. Pourtant, après configuration du *Toolkit* suivant les recommandations du NCBI sur le site https://ncbi.github.io/sra-tools/install_config.html, le problème persiste.

Au cours des premiers tests d'installation, il est apparu qu'une fois les dépendances installées, elles n'étaient pas pour autant reconnues par un système vierge. Ainsi, l'erreur suivante s'affichait systématiquement

puis l'erreur suivante s'affichait également

Après avoir cherché du côté des chemins d'accès des différents fichiers support, il est apparu que le problème venait plutôt des variables d'environnement qui devaient être configurées. Ainsi, il est nécessaire que l'utilisateur ait convenablement répertorié les chemins d'accès vers ses fichiers exécutables et dans le cas présent les modules *parallel-fastq-dump* et *blastn*.

Renan Valieris sur son dépôt https://github.com/rvalieris/parallel-fastq-dump recommande l'utilisation de *bioconda* pour installer le package *parallel-fastq-dump*. En effet, nous nous sommes

rendus compte sur le système Ubuntu 20.04 LTS, que l'installation à partir de *pip install parallel-fastq-dump* se fait correctement mais le package se trouve à la fois dans XXX et dans XXX, ce qui empêche le bon fonctionnement de l'instruction de test suivante

```
parallel-fastq-dump -s ERR2704812
```

et lève l'erreur suivante

```
XXX
```

Après vérification que la variable d'environnement `$PATH` est bien configurée, le problème reste présent. Malgré le fait qu'il existe sûrement une solution à ce problème, l'installation de *parallel-fastq-dump* par la commande `conda install -c bioconda parallel-fastq-dump` le résout immédiatement.

il est donc nécessaire d'interroger la variable d'environnement `$PATH` avec la commande

```
echo $PATH
```

On doit obtenir un résultat du type

## 5. Les tests de validation

Les tests de validation 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

```
python -m crisprbuilder_tb --collect ERR2704808 ne parvient pas à trouver les modules ...
```

Les tests fonctionnels comprennent la validation de l'interface en ligne de commande. Il s'agit donc de vérifier que les lignes de commandes prévues pour l'exécution du code fournissent bien le résultat escompté. Pour cela, nous testons successivement les commandes

- `python -m crisprbuilder_tb --help` qui fournit bien les informations relatives aux options du package, `python -m crisprbuilder_tb --print 0` qui affiche bien la liste demandée,
- `python -m crisprbuilder_tb --collect ERR2704808`
- `python -m crisprbuilder_tb --change 0`
- `python -m crisprbuilder_tb --remove 0`

`python -m crisprbuilder_tb --add 0`

Nous considérons donc que l'utilisateur dispose déjà sur sa machine d'un SRA Toolkit fonctionnel, notamment des modules *parallel-fastq-dump*, *blast* et *blastn* produisant les résultats escomptés avant d'installer de CRISPRbuilder_TB.

### 5.1. Les problèmes liés aux chemins d'accès

L'incompatibilité des chemins d'accès entre les systèmes Posix et Windows nécessite la définition des chemins sans recourir aux caractères "/" ou "\". Par ailleurs, comme le package va s'installer à des emplacements différents en fonction du système d'exploitation et des environnements de travail installés sur la machine (*venv, anaconda, ...*), on ne peut pas définir d'adresse absolue mais privilégier plutôt des adresses relatives. Il est possible d'utiliser le module pahtlib, le module PurePath qui dépend du premier ou encore __file__. Ainsi, l'adresse **crisprbuilder_tb/REP/sequences** s'écrit :

- **pathlib.PurePath(crisprbuilder_tb.__path__[0], 'REP', 'sequences')**
- **os.path.join(path.dirname(__file__), 'REP', 'sequences)**

## 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 utilise lors de ses requêtes *fastq* et *blast* les serveurs du NCBI[3] et non pas ses propres serveurs. Si les serveurs de NCBI sont occupés, les réponses obtenues n'en seront que différées. CRISPRbuilder_TB n'est pas une application Web, l'utilisation simultanée de plusieurs ordinateurs exécutant le package n'a pas d'incidence sur celui-ci.

En revanche, l'optimisation des performances de traitement des données doit être pris en compte car les fichiers de type *.fasta* ont une taille comprise entre 1 et 3 Go, ce qui génère des délais importants lors de l'exécution du package.

### 6.1. subprocess.run et os.system

Au début de notre travail sur le code, nous avons transformé les lignes de commande Linux appelées par les méthodes Python **subprocess.run** et **os.system** en fonctions entièrement écrites en Python afin d'améliorer la portabilité de l'application, sans se limiter aux systèmes Linux. Nous avons constaté lors des premiers tests d'intégration un fort ralentissement du programme lorsque ces modifications étaient appliquées à l'ensemble du code, 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 imposera en revanche un changement de lignes de commandes 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 tester dans le programme si le système est Posix ou Windows, puis de conserver les lignes de commande pour les système Posix et d'utiliser des fonctions Python pour les systèmes Windows. Cela entraîne de ce fait un réel manque de performance dans le traitement des données pour Windows.

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

3. **NCBI :** National Center for Biotechnology Information

exécuté 10 fois sur un système Ubuntu 18.04 LTS avec un processus Intel Pentium de 1,6 GHz 64 bits 4 coeurs

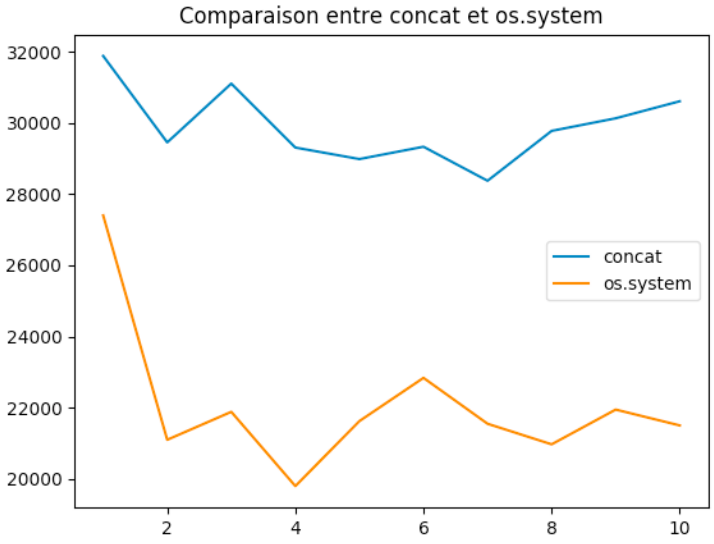
```
import time
import os

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 nom-

breuses minutes de retard sur l'ensemble du code lors de son exécution. Les temps moyens d'exécution du programme pour chaque système d'exploitation sont relevés dans la rubrique *Les tests de configuration*.

### 6.2. parallel-fastq-dump

Le package utilise les modules Python de téléchargement et traitement de données provenant du NCBI *fastq-dump* et *blastn*.

*fastq-dump* est un module archaïque permettant de transformer des données du format *.sra* au format *.fastq*. Il est contenu dans le *SRA Toolkit* du NCBI. Il peut s'avérer très lent en fonction de l'utilisation de la base de données en temps réel.

L'utilisation de plusieurs threads devrait permettre de diviser le travail en différentes tâches effectuées simultanément, et donc de gagner du temps. Malheureusement, Python n'accepte pas nativement le travail multi-threads, ce qui ne permet pas de tirer profit de systèmes multi-coeurs. C'est en fait une caractéristique de CPython. Le GIL [4] est un mutex qui n'autorise le travail que d'un unique thread à la fois. Il protège l'accès aux objets Python, pour empêcher de multiples threads d'exécuter du code Python simultanément. Ce verrou est rendu nécessaire parce que la gestion de la mémoire sous CPython n'est pas thread-safe. Le module *parallel-fastq-dump* résout partiellement ce problème de multi-threading et utilise l'implémentation CPython en incorporant du langage C dans son contenu. Il semble être à l'heure actuelle un bon compromis gratuit pour optimiser le traitement des données du format *.sra* au format *.fastq*. Notons que le *SRA Tookit* est installé automatiquement au moment de l'installation de *parallel-fastq-dump*.

REAL PYTHON ABOUT PYTHON THREADING

### 6.3. blast+

*blast* est également un outil du NCBI. Il permet de trouver des régions similaires entre plusieurs séquences génomiques. Il traite donc une quantité importante de données et ralentit de façon significative l'exécution du programme. Il compare des séquences de nucléotides ou de protéines à des séquences déjà présentes dans la base de données NCBI, puis calcule la signification statistique. *blast* existe sous plusieurs formes *blastx, tblastn, blastn, Primer-blast, smartblast, igblast, moleblast, ....* Notre package utilise *blastn* qui effectue des comparaisons entre nucléotides et dépend de la librairie *blast*. Les performances de *blast* sont optimisées avec la suite *blast+* qui utilise le Toolkit C++ du NCBI. Ainsi notre package utilise l'outil `makeblastdb` qui appartient à *blast+*.

## 7. 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.

TODO - CONFIG MINI

4. **GIL :** Global Interpreter Lock

# 8

# Conclusion

Ce stage m'a ouvert l'esprit vers le domaine de la bio-informatique, domaine intéressant qui a constitué un support scientifique très concret, me permettant de développer des compétences pratiques dans les domaines du développement et du test.

Bien que, a priori peu attiré par le travail de validation et de test de logiciels, j'ai pu me rendre compte rapidement de son importance et en apprécier l'efficacité, vu que les tests étaient omniprésents tout au long du stage.

Le temps passé à réfléchir aux nombreux dysfonctionnements de l'application, la frustration de comprendre tardivement que la solution provenait parfois d'éléments indépendants au code tels que l'environnement de développement ou la configuration de la variable $PATH, le souhait de bien comprendre les rouages d'un package Python ont fortement contribué à l'amélioration de mes connaissances en informatique.

En particulier, ce stage a mis l'accent sur les trois domaines suivants :

- la création d'un package et la gestion des dépendances,
- la nécessité d'un campagne de tests et de validation du code,
- la connaissance de notions clés concernant le séquençage du génome.

Ce package fournit des informations pratiques et utiles concernant des références de SRA. Il est amené à évoluer dans l'avenir, afin de propoer une reconstitution du CRISPR dans le cadre particulier de la tubercolose.

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

# Index

# Références


[1] Guyeux C, Sola C, Noûs C, Refrégier G. *CRISPRbuilder-TB : "CRISPR-Builder for tuberculosis". Exhaustive reconstruction of the CRISPR locus in Mycobacterium tuberculosis complex using SRA*. PLoS computational biology. 2020 ;submitted(PCOMPBIOL-S-20-00832-2)

[2] Guyeux C, Sola C, Refrégier G. *Exhaustive reconstruction of the CRISPR locus in M. tuberculosis complex using short reads BioRxiv*. 2019a. doi : https ://doi.org/10.1101/844746

[3] Coll F, Preston M, Guerra-Assuncao JA, Hill-Cawthorn G, Harris D, Perdigao J, et al. *PolyTB : A genomic variation map for Mycobacterium tuberculosis. Tuberculosis* (Edinb). 2014 ;94(3) :346-54(3) :346-54. doi : 10.1016/j.tube.2014.02.005. PubMed PMID : 24637013

[4] Stucki D, Brites D, Jeljeli L, Coscolla M, Liu Q, Trauner A, et al. *Mycobacterium tuberculosis lineage 4 comprises globally distributed and geographically restricted sublineages*. Nature genetics. 2016 ;48(12) :1535-43. doi : 10.1038/ng.3704. PubMed PMID : 27798628

[5] Palittapongarnpim P, Ajawatanawong P, Viratyosin W, Smittipat N, Disratthakit A, Mahasirimongkol S, et al. *Evidence for Host-Bacterial Co-evolution via Genome Sequence Analysis of 480 Thai Mycobacterium tuberculosis Lineage 1 Isolates*. Scientific reports. 2018 ;8(1) :11597. Epub 2018/08/04. doi : 10.1038/s41598-018-29986-3. PubMed PMID : 30072734 ; PubMed Central PMCID : PMCPMC6072702

[6] Shitikov E, Kolchenko S, Mokrousov I, Bespyatykh J, Ischenko D, Ilina E, et al. *Evolutionary pathway analysis and unified classification of East Asian lineage of Mycobacterium tuberculosis*. Scientific reports. 2017 ;7(1) :9227. doi : 10.1038/s41598-017-10018-5. PubMed PMID : 28835627 ; PubMed Central PMCID : PMCPMC5569047

[7] Kamerbeek J, Schouls L, Kolk A, van Agterveld M, van Soolingen D, Kuijper S, et al. *Simultaneous detection and strain differentiation of Mycobacterium tuberculosis for diagnosis and epidemiology*. J Clin Microbiol. 1997 ;35(4) :907-14. PubMed PMID : 9157152

[8] van Embden JDA, van Gorkom T, Kremer K, Jansen R, van der Zeijst BAM, Schouls LM. *Genetic variation and evolutionary origin of the Direct repeat locus of Mycobacterium tuberculosis complex bacteria*, J Bacteriol. 2000 ;182 :2393-401

[9] 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

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

[11] 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

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

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

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

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

[16] 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

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

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

[19] 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

[20] 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

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

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

[23] 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

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

[25] 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

[26] ,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

[27] 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